Interactive Visualization of Mobile Network Simulations & 3D Segmentation and Quantification of Brain Tumors

By
Jake Proctor

Senior Thesis

Advisors: Dr. K. R. Subramanian, Dr. T. A. Dahlberg, and Dr. J. P. Brockway

08 May 2002

Table of Contents

Acknowledgements

I would like to thank Dr. Subramanian, Dr. Dahlberg, and Dr. Brockway for their expertise advice and assistance over the course of this project. Their guidance and support have been an invaluable resource, without which this work could not have been completed.

Chapter 1

Introduction

Two constantly growing domains where much research is currently focused are congestion control in mobile networks and the segmentation and quantification of brain tumor volumes. The purpose of this report is to detail the work I have performed towards my senior project at the University of North Carolina at Charlotte, which relates to these areas of research. Over the course of one academic year, I worked on two separate projects. The first of which was related to the area of research surrounding congestion control in mobile networks and the visualization of survivability metrics for these networks. The second project involved segmenting tumors from datasets produced by magnetic resonance imaging (MRI) of the human brain and quantifying these tumors.

This report is divided into two major sections, each describing the details of one of the projects mentioned above. The first section provides a brief background about the problems faced with mobile networks and the current approaches to solving these problems. This section also describes Mobvis, an application developed by Dr. K. R. Subramanian and Dr. T. A. Dahlberg, which provides 3D visualization of real-time survivability metrics for mobile networks. The work I performed, which was focused on the modification and addition of features to Mobvis, is also described in this section.

The second major section of this report describes the work I performed on an application developed by Dr. K. R. Subramanian that provides a means to visualize 3D MRI datasets. This section provides a brief background of the problems surrounding accurate segmentation of tumors from human brain images and accurately computing the volume of these tumors. A discussion is included on an alternate implementation of the

MRI application using the Python programming language and the segmentation and volume computation features that have been added to the original implementation. Results of the segmentation and volume computation algorithms are portrayed through the presentation of three MRI datasets of the human brain.

There exist an opportunity for much future work to be performed using each of the applications mentioned above, thus each major section includes a sub-section detailing possible future features that could be added.

Chapter 2

Interactive Visualization of Mobile Network Simulations

2.1 Background Information and Problem description

A major challenge faced by mobile network service providers is to guarantee its customers an acceptable Quality of Service level. With the recent proliferation of mobile network use by both the public and the private sector, the demand placed on these networks is growing at an astronomical rate. Another factor that is complicating this problem even further is the merging of voice and data transmissions onto the same networks.

Users of these mobile networks expect to be able to user their mobile device whenever they like and have a guaranteed level of service provided. In addition, these users expect this level of service to be extended throughout their period of use on the network, regardless of changing conditions the network experiences on a constant basis. The networks generally experience bursts of demand for access and need to have some means of dealing with this demand under all possible network conditions.

Congestion on the mobile network can arise from numerous sources, some which are part of the actual network architecture, and others that are completely independent of the network itself. Examples of network congestion sources include failure of hardware or software the network is built upon, times or areas of peak demand, such as during rush hour or around traffic accidents, current weather conditions, and the topography of the surrounding landscape. These are all factors that must be considered and dealt with in developing a process for adaptively allocating network resources as required by the current state of all the present variables [2,3].

A mobile network is composed of several base stations, each of which covers some defined area, referred to as a 'cell'. As a user travels from one cell to another on the network, they must be transitioned from the base station currently handling their call, to the base station in the cell they are entering. A procedure must be in place to allocate resources for this transition, while not unnecessarily blocking potential new calls. The resources that must be allocated are the available channels each base station has available. It is important to note that these channels are not only limited by monetary factors, but there is a limited frequency spectrum for these channels to use, hence the need for adaptive algorithms to allocate resources according to the variable state of the network.

Dr. Dahlberg and Dr. Subramanian have worked towards developing Adaptive Admission Control algorithms to recognize the current state of the network and to dynamically allocate resources based upon the identified state, and accurately measure the performance of these algorithms. These algorithms must have the capability to react to multiple variables that are constantly changing, and sometimes changing at a dramatic rate. Dr. Dahlberg and Dr. Subramanian point out that there is a major tradeoff in developing these algorithms with distinguishing between real problems on the network in which the algorithm needs to address, from normal burst activity that the algorithm should ignore. Thus, major focus is placed on the sensitivity of the algorithm. Dr. Dahlberg and Dr. Subramanian experimented with four Adaptive Admission Control algorithms, which are described in detail in [1]. As a simple and brief description, these algorithms all monitor some defined, real-time metric and vary a guardband, referred to as δ, which defines the percentage of channels available for new-call requests and the percentage of channels set-aside for necessary handover requests. A handover request occurs when a user is traveling from one cell to another on the network and needs to switch to the base station of the cell they are entering, while a new-call request occurs when a new user attempts to gain access to the network. This allocation policy must be able to adapt to provide an optimal level of service to the customer under varying conditions such has a normal load versus a heavy load or under network failure conditions.

Numerous metrics are defined and explained in [1,2,3], two of which I will define here. These two are handover blocking rate (HBR) and new connection blocking rate (NBR). These were the main two metrics I used when testing the new features I added to Mobvis, so I will limit my description of metrics to these. HBR describes the percentage of handover requests denied which result in the termination of the connection. The following formula is defined in [1] for HBR:

$$\text{HBR} = (\Sigma \text{ handover requests denied})/(\Sigma \text{ handover requests})$$

The data is summed over a predefined period of time. NBR describes the percentage of new connection requests that are denied. The formula for NBR is given by:

$$\text{NBR} = (\Sigma \text{ new connection requests denied})/(\Sigma \text{ new connection requests})$$

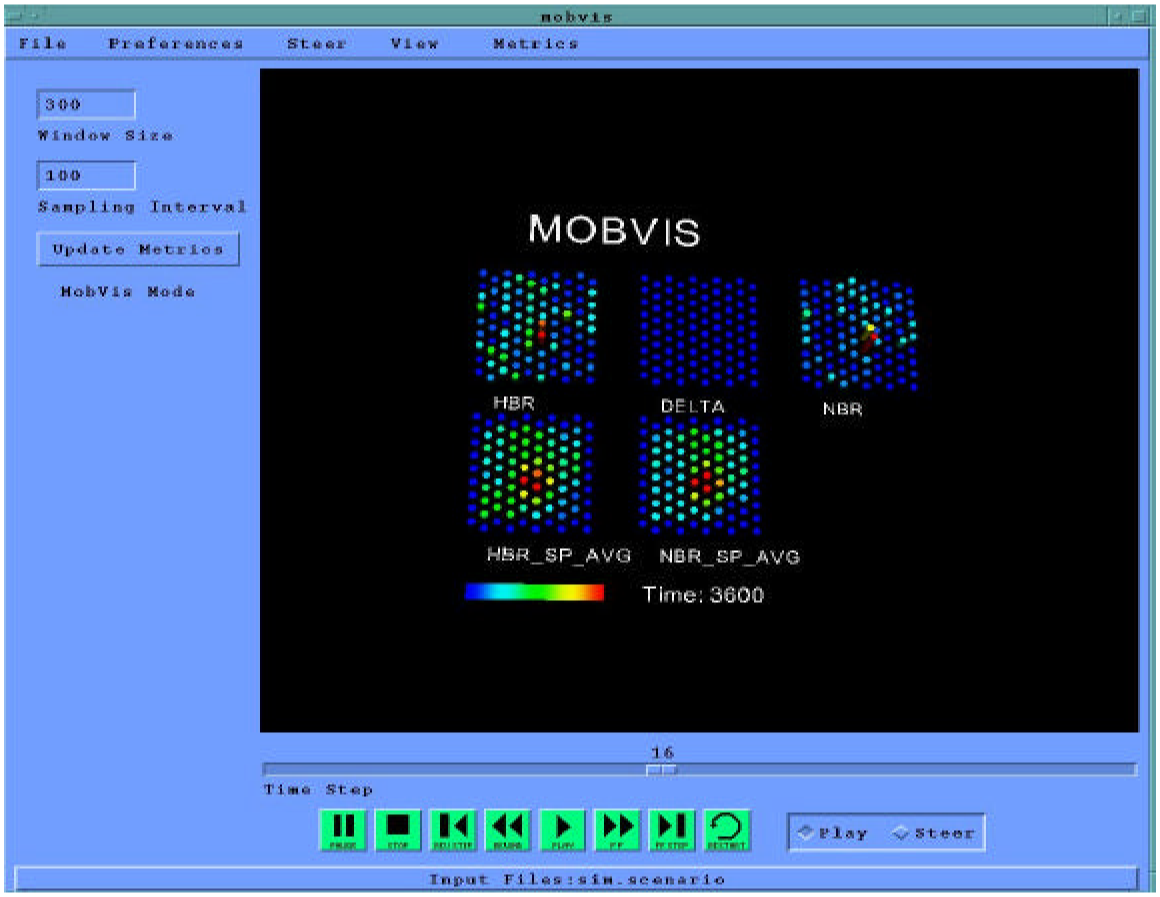
The four Adaptive Admission Control algorithms mentioned previously, examine HBR, its rate of change or derivative, NBR, and its rate of change, respectively. Once the simulations are run using these different algorithms, a means of examining the results of the allocation policy the algorithms invoke is needed. We need a method of comparing the effects of varying conditions, such as network failure or extreme demand, on each of the algorithms and how they handle such conditions. This analysis needs to be performed on both a spatial and temporal basis.

Furthermore, an Adaptive Admission Control algorithm must monitor the network and evaluate the system at each cell site, or base station, on the network. Given this fact, a simulation run over even a short period of time results in an explosion of data that needs to be quickly and accurately analyzed to determine the effectiveness of the algorithm under scrutiny. Traditionally such analysis has been performed through statistical methods and 2D plots of the data over time. Unfortunately these methods do not lend themselves well to the multivariate nature of the data we are attempting to analyze. Data visualization provides a powerful alternate method of analyzing the results from the network simulations on the required temporal and spatial bases [1,2,3].

2.2 Description of Mobvis

Mobvis, developed by Dr. Subramanian and Dr. Dahlberg, is an application that provides 3D visualization of the real-time metrics described in the previous section. The visualization tools are constructed using the Visualization Toolkit (VTK) [4], and the GUI is constructed using OSF/Motif. Mobvis displays the simulation runs in a spreadsheet style layout, allowing for comparison of a variable number of metrics from the different algorithms at once [2]. Spreadsheets have traditionally been extremely effective for interacting and manipulating numerical data. The benefits of using the spreadsheet layout for visualization of large and complex multidimensional datasets are presented in [5]. These benefits include a user's ability to view multiple visual representations of different datasets at once, perform actions on these visualizations, and visually compare the datasets. Such advantages prove very useful in the visualization of mobile network metrics from different algorithms over time. A screen shot of the application is provided below:

Figure 1: Mobvis Application



The original implementation of Mobvis provides three different types of visualizations. These are 1) color mapped planes, where metric values are mapped to a set of colors, 2) height fields (two types of height fields are available), where metric values are scaled and mapped as a height, and 3) parallel coordinates, where the metrics each are represented on their own vertical axes [1]. In Figure 1, above, one of the available height field visualizations is shown.

One of the additions I implemented was to provide a fourth visualization, which is an iso-surface of the entire time span over which the simulation is run. This feature allows the user to view areas where a chosen value occurs for the metric in question over the entire time span of the simulation run without needing to move through the simulation. More details on this feature are provided in the following section of the report.

Mobvis provides the ability to run the simulation and view the data through VCR style controls along the bottom of the application. The user may choose either the 'Play' or 'Steer' mode. When in 'Steer' mode, the actual simulation is being run. While in 'Play' mode, the user is simply moving through pre-computed sequences of the simulation. The user must first create a simulation by using 'Steer' mode and then may rewind/fast forward/play through that simulation using 'Play' mode.

VTK provides an interactor into the visualization window that allows the user to manipulate the visualizations through scaling, translation, and rotation. Users are able to compare the data from any visual perspective they prefer through this capability.

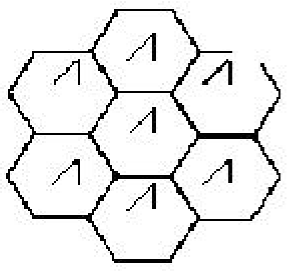
The use of data visualization proved extremely valuable to Dr. Dahlberg and Dr. Subramanian in their analysis of the algorithms under testing. Their results, including why their conclusions could not have been reached without the aid of data visualization are detailed in [1]. In the conclusion of their paper, they suggest several other possibilities of visualizing the data through different methods, which could prove even more useful in analyzing the Adaptive Admission Control algorithms. My work involved implementing some of these alternatives.

2.3 Modifications and Additions to Mobvis

One suggestion from Dr. Dahlberg and Dr. Subramanian was to implement a means of investigating the spatial relationships between the metric value in one cell with the metric value in its neighboring cells. This could perhaps provide insight into the effects of the algorithm being used on neighborhoods of cells, rather than isolated cells themselves.

As a potential solution to providing such a tool, I implemented a low pass spatial filter to be applied to the metric values over the entire region of cells. The simulation provides a region of 90 cells, in a hexagonal layout. The layout of the cells is shown below in Figure 2, reproduced from [1].

Figure 2: Layout of mobile network cells from [1].



As can be seen from above, each cell in the network will have six neighbors, one directly above, one directly below, and one positioned at each of four diagonal positions. The metrics, once computed for the simulation run, are stored in a three-dimensional array, which is indexed by 1) the metric being computed 2) the time step of computation and 3) the id of the particular cell the metric is computed for.

According to [6] the key requirement for such a low pass spatial filter is that all coefficients of the filter be positive, which can readily be achieved by making the coefficient for each cell equal to 1/X, where X is the number of cells the filter is applied to. In our scenario, there are seven cells to apply the filter to: the current cell, and its six neighbors.   Using this method, each cell receives a value equal to the average of its value with the values of is six neighbors. This is commonly referred to as neighborhood averaging.

The three-dimensional array containing the metric values described above is maintained in the mvisMetric class of Mobvis. I added the functionality of applying this spatial filter in the mvisMetric class along with the computation of the metrics themselves. The first step in implementing the averaging filter was to provide a means of computing the address of each cell's six neighbors. Even though the cells are offset in this hexagonal layout, finding the address of the six neighbors was fairly straightforward. Each cell's six neighbors are in exactly the same location relative to its own location, thus I was able to simply subtract/add some constants to each cells' address and I had the addresses of each cells' neighbors. The next step was to sum the values of the metrics in each of the seven cells and assign this value, divided by seven, to the current cell.

The metrics are computed one cell at a time for each time step. Since a cell's neighbors consist of three cells before it in the three-dimensional array and three cells after it in the three-dimensional array, the averaging could not be performed until all metric values were computed for that time step. Once the computation for the current time step is complete, the algorithm iterates through the three dimensional array, applying the filter to each cell, and then allows the metric computation module to proceed to the next time step. Figure 3 and Figure 4 display screen shots of the result of applying this low-pass filter to the HBR and NBR metrics. The two different figures are provided to portray the two types of height field visualizations Mobvis provides. In both figures, HBR is the metric displayed on the top row on the far left and NBR is the metric displayed on the top row on the far right. The result of the neighborhood averaging for HBR is shown in the bottom row on the left and the result of the neighborhood averaging for NBR is shown in the bottom row on the right.

Figure 3: Spatial Averaging of HBR & NBR – Using first type of Height Fields

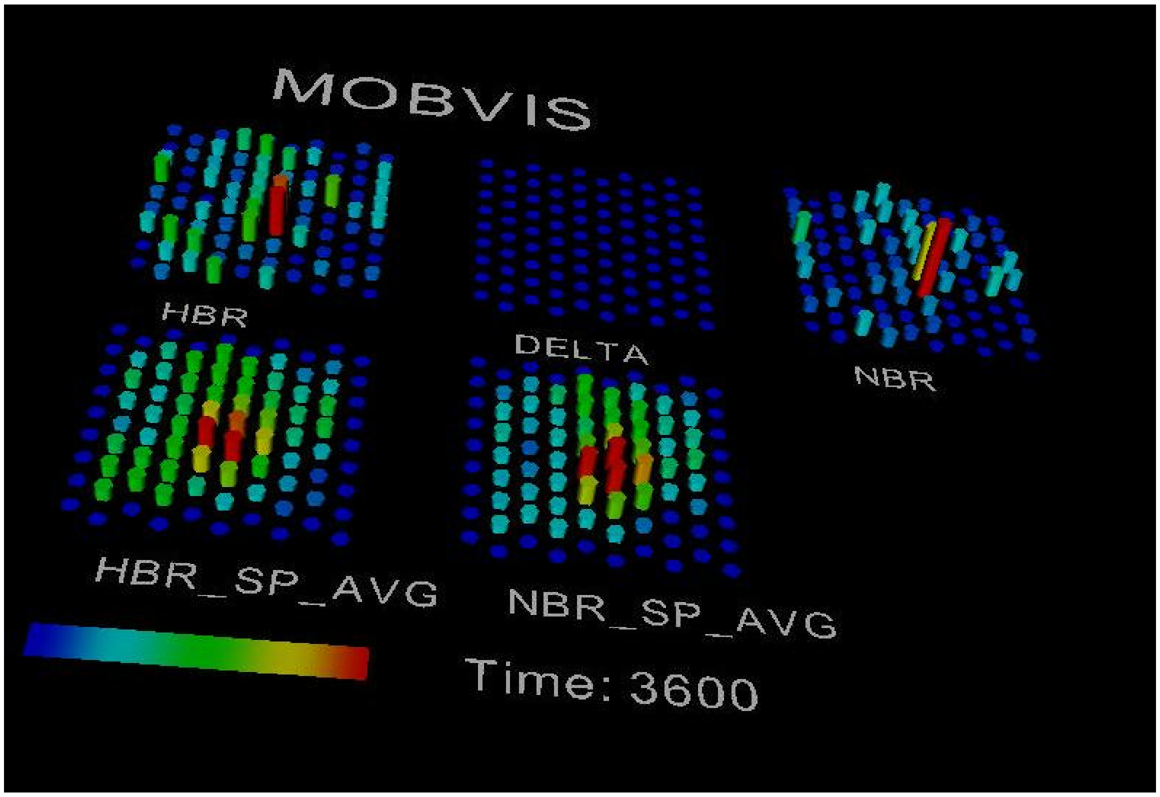


Figure 4: Spatial Averaging of HBR & NBR – Using second type of Height Fields

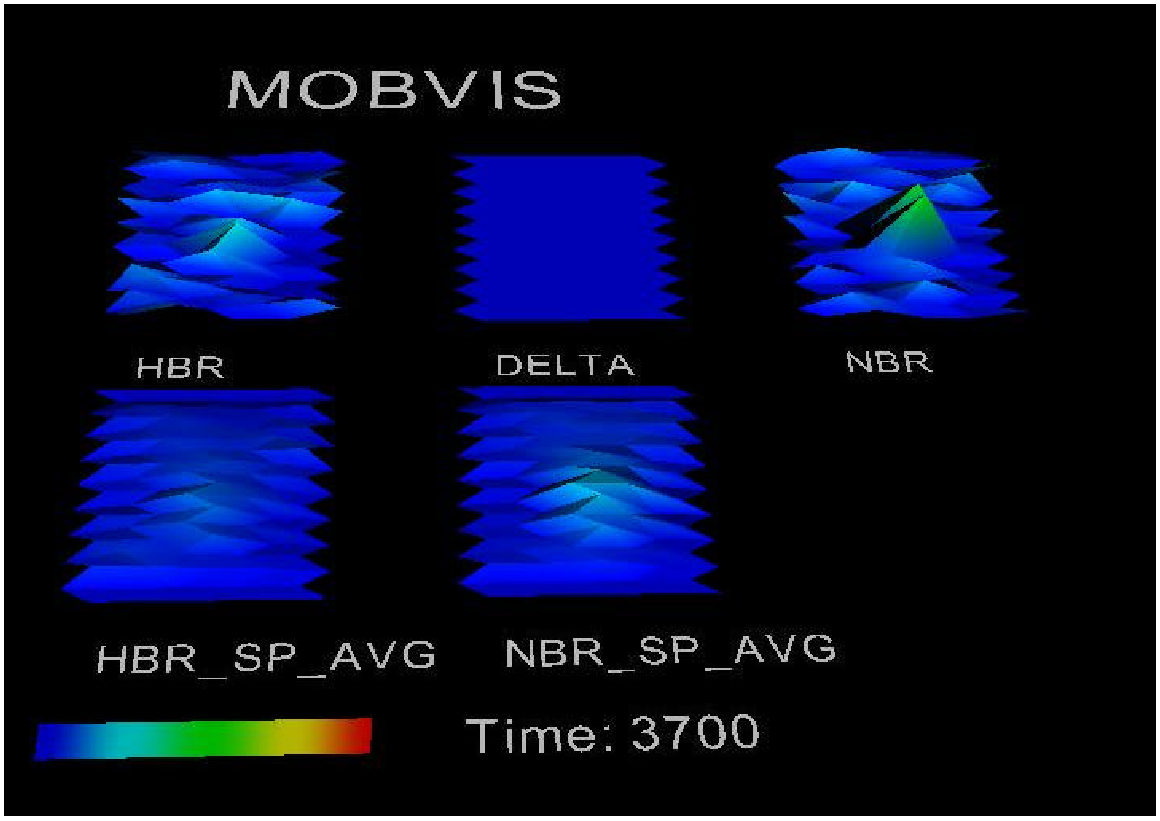


Figure 3 and Figure 4 clearly show the effect of neighborhood averaging on the metric values for each cell in the network. The result is what was expected, in that the smoothing of the values is evident and large peak values for the metrics have been spread out amongst the cells' neighbors.

Another feature I added to Mobvis, was a fourth type of visualization, which allows the user an alternate means of comparing a single metric over the entire time span of the simulation. This visualization type was achieved by creating an iso-surface from the value of a particular metric over the entire time the simulation was ran. The iso-surface visualization is now accessible from the ‘View’ menu as an additional visualization type.

When a user selects the iso-surface option, they are prompted to enter the metric in which they would like to see the iso-surface for, along with the value around which they would like to see the iso-surface constructed. This mandates that the user have an idea of what value they would like to look for. To meet this demand, a means for the user to query the value of the metric they were curious about was added. Mobvis already supported picking of the actors that represented each cell in the visualization. The functionality that needed adding was to have the cell id of the picked cell along with both the scaled and unscaled value of the metric displayed along the message bar at the bottom of the application. It was necessary to include the unscaled value, because this is the value actually stored in the three-dimensional array, thus if a user would like to view the iso-surface for a particular value, the unscaled value is what they must enter.

Once the user inputs the metric they are interested in, and the value the algorithm should construct the iso-surface around, the current visualization is removed from the visualization window and the iso-surface is displayed. Figure 5 shows the iso-surface constructed from the HBR metric using a non-scaled value of 0.15.

Figure 5: Iso-Surface Of HBR with unscaled value of 0.15.

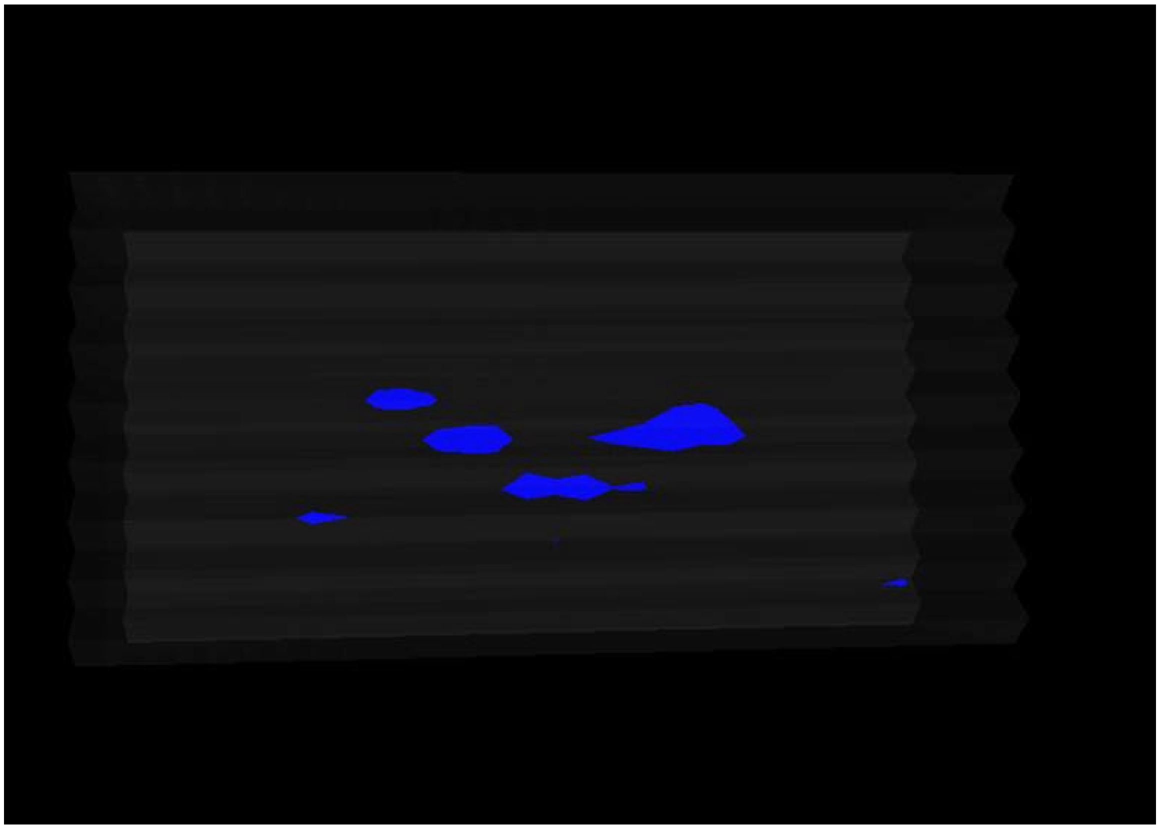


Figure 5 shows the occurrences where the value of HBR rose above 0.15. This image is taken from an orientation that begins with time 0 on the left, and runs to the end of the simulation on the right. The iso-surface can be manipulated in the same manner as the other visualizations, so the user can rotate the image to view the location of the cells where these iso-surfaces are occurring.

The iso-surface functionality has been added to the VisualizationPanel object of Mobvis, with related functions to call and support the procedure in the GUI object and the MvisVtk object. The metric values for the cells at each time step are input to a vtkStructuredGrid object, which is a dataset whose structure is topologically regular, but whose geometry is irregular [4]. The iso-surface is constructed using the marching cubes algorithm, which is described in [13]. Basically, the marching cubes algorithm creates a contour surface at a constant scalar value from the vtkStructuredGrid object.

The added functionality of having the ability to view a metric for all network cells over the entire time span of the simulation run provides a great advantage in a temporal analysis of the algorithm being used. Without this feature, the user must fast forward or rewind through the simulation run to visualize the changing values of the metrics as a result of the allocation policies put in place by the Adaptive Admission Control algorithm. The user can get an idea of how fast the system recovers from failures on the network or increased demand much quicker by simply viewing the iso-surface and examining how long the contour surfaces are, rather than moving, time-step by time-step, through the simulation run.

2.4 Future Work

With these new features added to Mobvis, there are still numerous additions and modifications that could be made to enhance the value this application has in the visualization of metrics from mobile network simulations. The neighborhood averaging functionality was added to Mobvis in a manner such that it can perform the averaging on any metric the user chooses. This function could be improved by implementing a more sophisticated low-pass filter method, perhaps by using a sampled Gaussian function to perform the averaging. In addition, other methods of determining the relationship of the metric values between cells should be explored. To add to the iso-surface feature, it would be useful to have the ability to view the iso-surface for multiple metrics at once. Currently, when the user is asked to input the metric to create the iso-surface for, they do not have the ability to input multiple metrics. Also, it may prove useful to be able to create multiple iso-surfaces at different values for a particular metric, perhaps distinguishable by varying the color of the iso-surface.

2.5 Mobile Visualization Summary

With mobile network technology improving at such a rapid pace, the need to analyze data created as a result of the policies implemented to manage the network will persist. Data visualization has proven itself a valuable tool in the analysis of data from this domain and will continue to play a prominent role in the advancement of this field.

The Mobvis application has proved itself a valuable tool in efficiently analyzing enormous amounts of data created from mobile network simulations. The additions and modifications described in section 2.3 extend the application's functionality and will increase researchers' ability to analyze the performance of Adaptive Admission Control algorithms.

Chapter 3

3D Segmentation and Quantification of Brain Tumors

3.1 Background Information and Problem Description

A revolution is currently occurring in the medical arena. The rate at which medical imaging technologies are being improved is accelerating rapidly. Imaging technologies such as computed tomography (CT), positron emission tomography (PET), magnetic resonance imaging (MRI), and ultrasound, are all allowing for large strides to be made in the detection and diagnosis of certain medical problems.

Medical imaging technologies allow physicians and scientist to non-invasively view the internal structures of the human body. Life-saving information can be gathered from simply using these technologies for the visualization and inspection of anatomical structures. In the instance of brain tumors, MRI images provide physicians with a means of detecting the existence of a tumor and visualizing the location of the tumor in two-dimensional planes. Imaging technologies are now being used in an even more profound sense, in that today these technologies have been expanded for the use of surgical planning and surgical simulation. If physicians can detect the shape of a brain tumor, its volume, and its location in a three-dimensional visualization of the brain, they could determine an optimal plan of approach prior to surgery. Having such detailed information prior to surgery would prove extremely useful for the physician, and would allow for reduction in damage to healthy tissue surrounding the tumor.

With medical imaging technologies being developed further everyday, and new uses for these technologies being realized, the need for analysis of the images these technologies produce is increasing. We need not only to extract the structures of interest from these images, but we need the capability to quantify and describe these structures. Simply trying to segment structures, such as tumors from a human brain, can prove a daunting task. Shapes of anatomical structures tend to be extremely irregular and complex, making the task of extracting an accurate representation of these structures very difficult. Once we have a geometric representation of a structure, we can then proceed to quantify this structure [7].

3.2 Objective of Project

Dr. Subramanian had previously developed an application that provided a means to visualize MRI datasets in both the three two-dimensional views (axial, coronal, sagittal) and a three-dimensional view constructed as a contour surface. The objective of this project was two-fold.

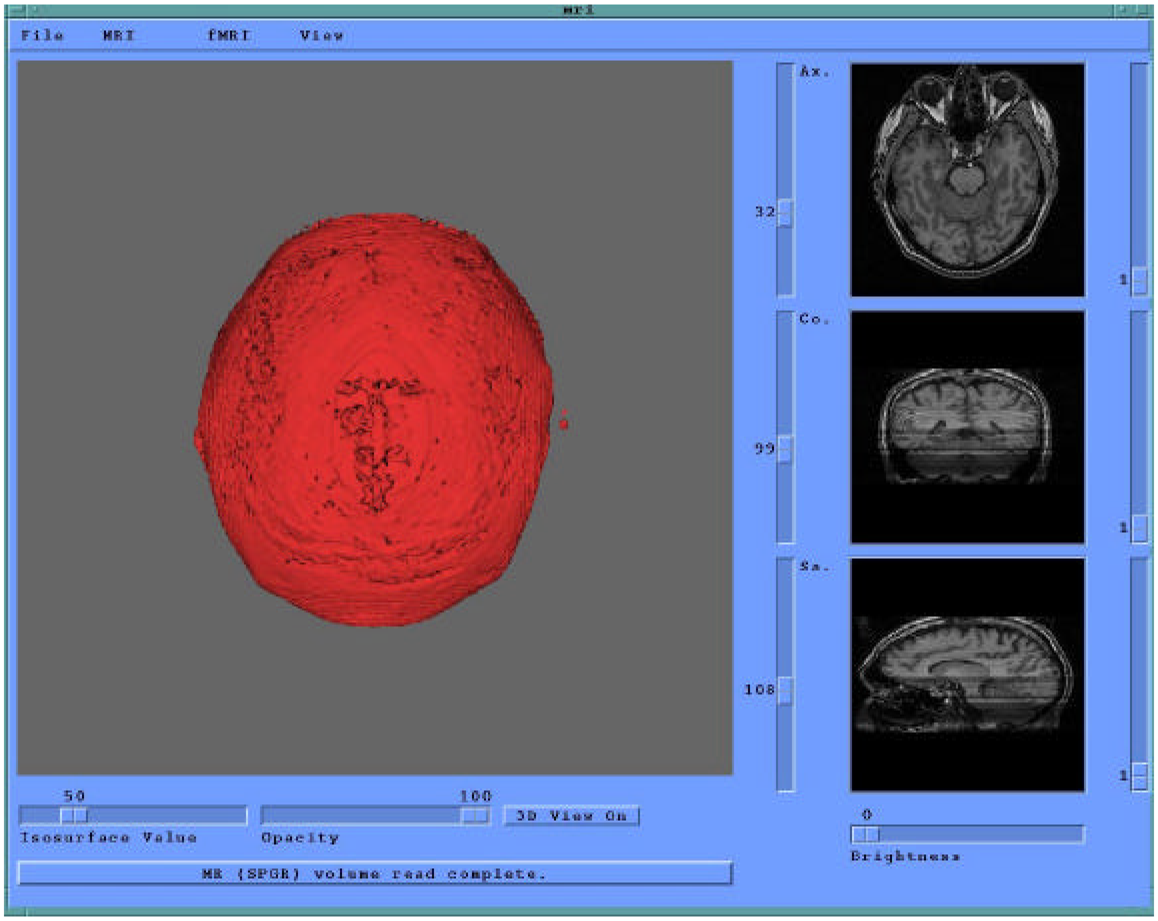
The first objective was to experiment with an alternate implementation of the current application. We were looking for an implementation that could provide the same functionality at the same speed, but that would be much simpler to maintain, extend, and port to other environments. We needed to have the capability of quickly adding interactive GUI features to the application for procedures such as interactive segmentation of tumors from the MRI datasets the application was designed to visualize.

The second objective was to actually extend the MRI application to provide the functionality of interactive 3D segmentation of tumor volumes from brain images. After segmenting the tumor from the healthy brain tissue, our goal was to provide an estimate of the volume of the tumor segmented. This application would provide physicians with a means to interactively view a 3D visualization of the human brain with the tumor segmented within that visualization. Making it a different color in the visualization makes the tumor distinct from the healthy brain tissue. Physicians could interactively move the visualization through rotation, scaling, or translation. While having this visualization before them, physicians would also be able to view an estimate for the volume of the tumor they are inspecting. As mentioned previously, this could provide a great advantage to the physician in pre-operative planning.

3.3 Description of current application

The application Dr. Subramanian developed allows a user to read MRI datasets into the application and construct visualizations of the datasets. The application was build to accept both the 3D .dset format, or a series of 2D images in the .MR format. The visualization tools are implemented through the Visualization Toolkit (VTK) [4] and the GUI is implemented using OSF/Motif. A screen shot of the application is provided below in Figure 6, with the mri2.dset dataset loaded into the application.

Figure 6: MRI application



MR (SPGR) volume read complete.

As can be seen from Figure 6, once a user chooses to open a dataset, the axial, coronal, and sagittal views are displayed in three windows on the right side of the application and a 3D view is constructed in a large window on the left. The user has the option to turn the 3D view on/off as necessary, as constructing the visualization requires a great deal of computation. The 3D visualization is constructed by forming a contour surface, which is achieved by using the marching cubes algorithm. The marching cubes algorithm is described and explained in detail in [13].

The user has the capability of scrolling through the entire dataset slice by slice at each one of the 2D views by moving the slider bars to the left of the viewing window. Also, if a user positions the mouse in one of the 2D viewing windows and clicks the left mouse button, the other two 2D windows automatically update to display the point chosen in the respective views. Scales are also provided to change the brightness of the 2D images to improve manual feature detection.

Moving the slider bar labeled ‘Isosurface Value’ under the 3D viewing window will change the iso-value at which the contour surface is created. The opacity of the 3D visualization of the human brain can be changed by changing the value of the slider bar labeled ‘Opacity’. VTK also provides the capability of interacting with the 3D visualization through scaling, rotation, or translation. This allows the user to gain a view of anatomical structures of interest from any perspective they like.

3.4 The Python Implementation

As stated in the introduction of this section, the first objective of this project was to experiment with an alternate implementation of the MRI application. The desire was to implement a version that provides equivalent functionality and efficiency, yet would be simpler to extend and simpler to port to other environments.

The programming language chosen to implement the new version was Python. Python is an interpreted and interactive objective-oriented programming language. Python is most often compared to languages such as Perl, Tcl, and Java. Python provides an excellent means of creating GUI’s through the use of the Tkinter module. Another advantage found with using Tkinter and Python is that they are extremely portable. Python runs on all major operating systems: UNIX, Linux, Windows, Mac, and OS/2.

This would be an enormous advantage over the current application built using the X/Motif libraries.

Python provides extremely quick turnaround in creating GUI applications. Developers can often make changes to a GUI and see the effect of those changes within a matter of minutes. With Motif, this is rarely possible. Programming the GUI with Motif is far more complicated, and of course with C++ code, it has to re-compiled each time changes are made. With Python being an interpreted language, re-compilation is unnecessary.

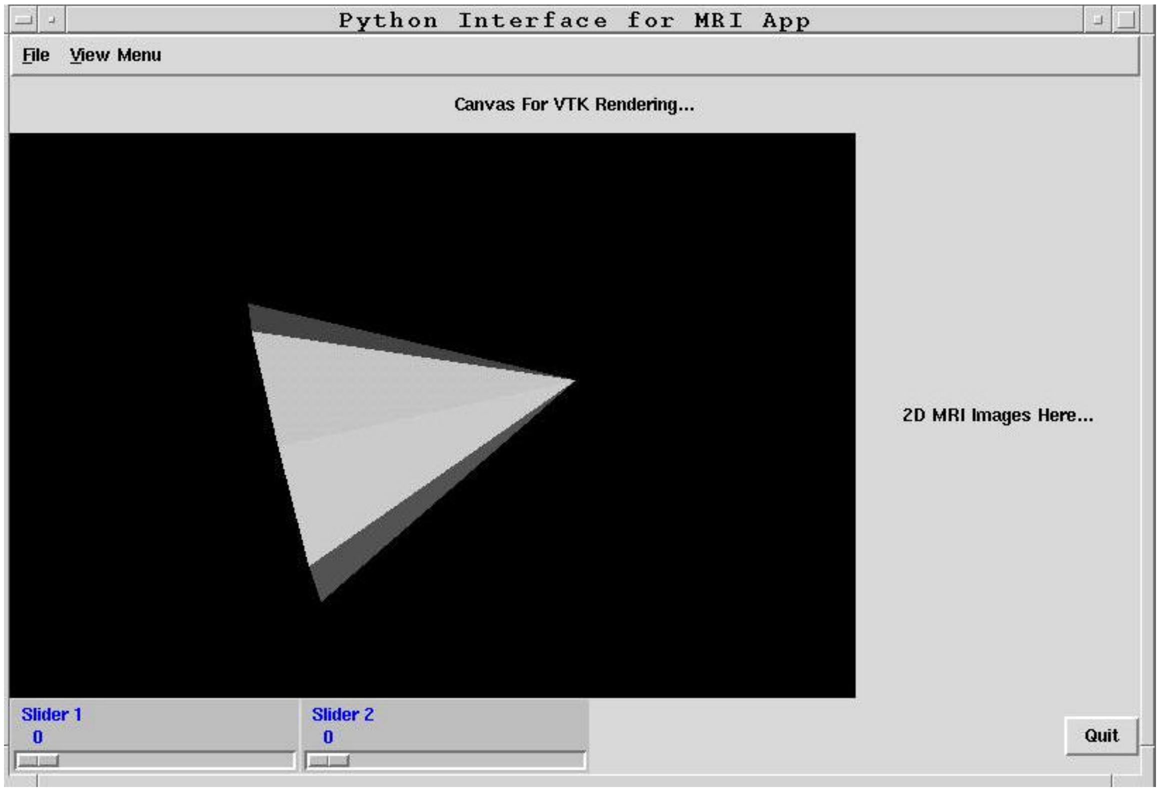
The one major weakness found with Python is that like all dynamic languages, it is not as fast and efficient as C++. In many domains this does not present a problem, as the difference would be indistinguishable [8]. But for the visualization of large 3D datasets, and to be able to interact with these visualizations, we need as much speed and efficiency as possible.

Thus, the planned design for the new version of the MRI application was to build the GUI using Python and Tkinter, and call C++ extensions from this GUI to do all computationally intensive tasks. With this plan in mind, I began reading books about Python and experimenting with simple GUI examples from these books.

With the Python implementation, we still needed to be able to use VTK to provide all of the visualization tools. Fortunately VTK provides extensions to be used with the Python programming language. In particular, the vtkTkRenderWindow module was most useful. This module provided exactly what we needed: a VTK window that could be embedded in a Tkinter GUI. From this point I was able to begin constructing a GUI as I learned more about the Python syntax, while including some VTK functionality as well.

Figure 7, below, shows one of the earliest GUIs I created using Python and Tkinter with a VTK window.

Figure 7: Python GUI 1



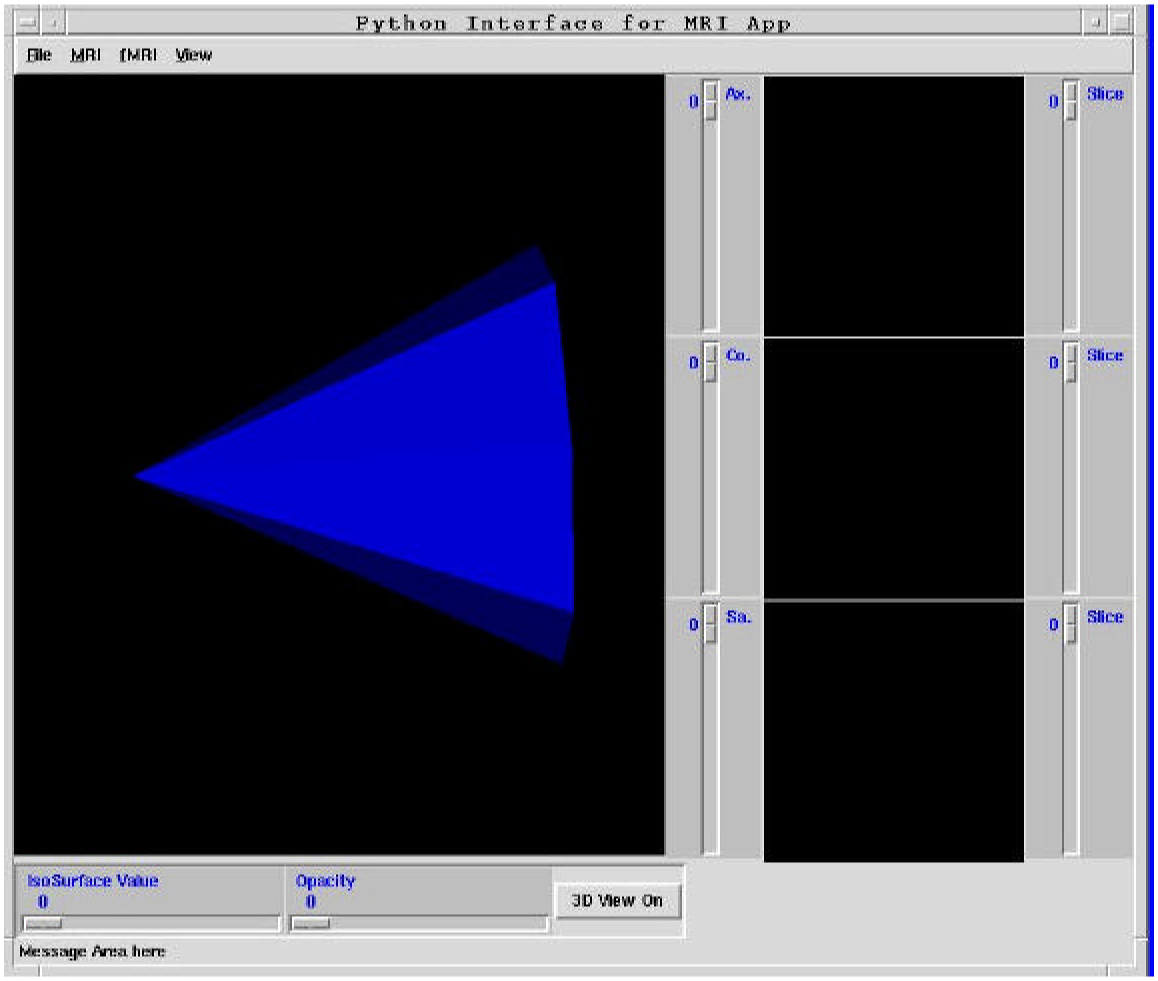
The VTK window is displaying a 3D cone, which can be rotated, scaled, or translated, just as the 3D visualization of the human brain could be manipulated in the original implementation. From this base GUI I began adding widgets building up to a GUI that would provide all of the same functionality as the original implementation.

In the original implementation of the MRI application, the large window for the 3D visualization was a VTK window, but the three small windows for the 2D views were OpenGL capable X windows. OpenGL was used to render these windows directly because this provided much faster redrawing and smoother transitions from slice to slice than VTK. In the original implementation, if the user moves the sliding scale for one of the 2D views, the new slice is drawn instantly; there is no delay whatsoever.

Python does not directly provide a means of creating OpenGL capable windows. I was able to find PyOpenGL online, which is a cross platform Python binding to OpenGL. We experimented briefly with PyOpenGL, but it turned out to not support the particular functions our application required. I proceeded with developing the GUI, using VTK windows for the three 2D views. At least in this case we would have to opportunity to see just how bad the performance could be for VTK from within the Python interface, perhaps it would not be unacceptable

Figure 8 below shows the completed Python GUI, with all four visualization windows being implemented as VTK windows.

Figure 8: Python GUI 2



Again, the 3D window is shown containing a 3D cone, created by using tools from VTK, which can be rotated, scaled, or translated. This GUI provides all the same functionality as the OSF/Motif GUI provided, with all the same menus, slider bars, and windows. The next step was to begin integrating the C++ functions from the original application to perform the computationally intensive tasks.

I began by extracting the function that reads in the MRI datasets from the mri object of the original application. I modified this function slightly so that I could use

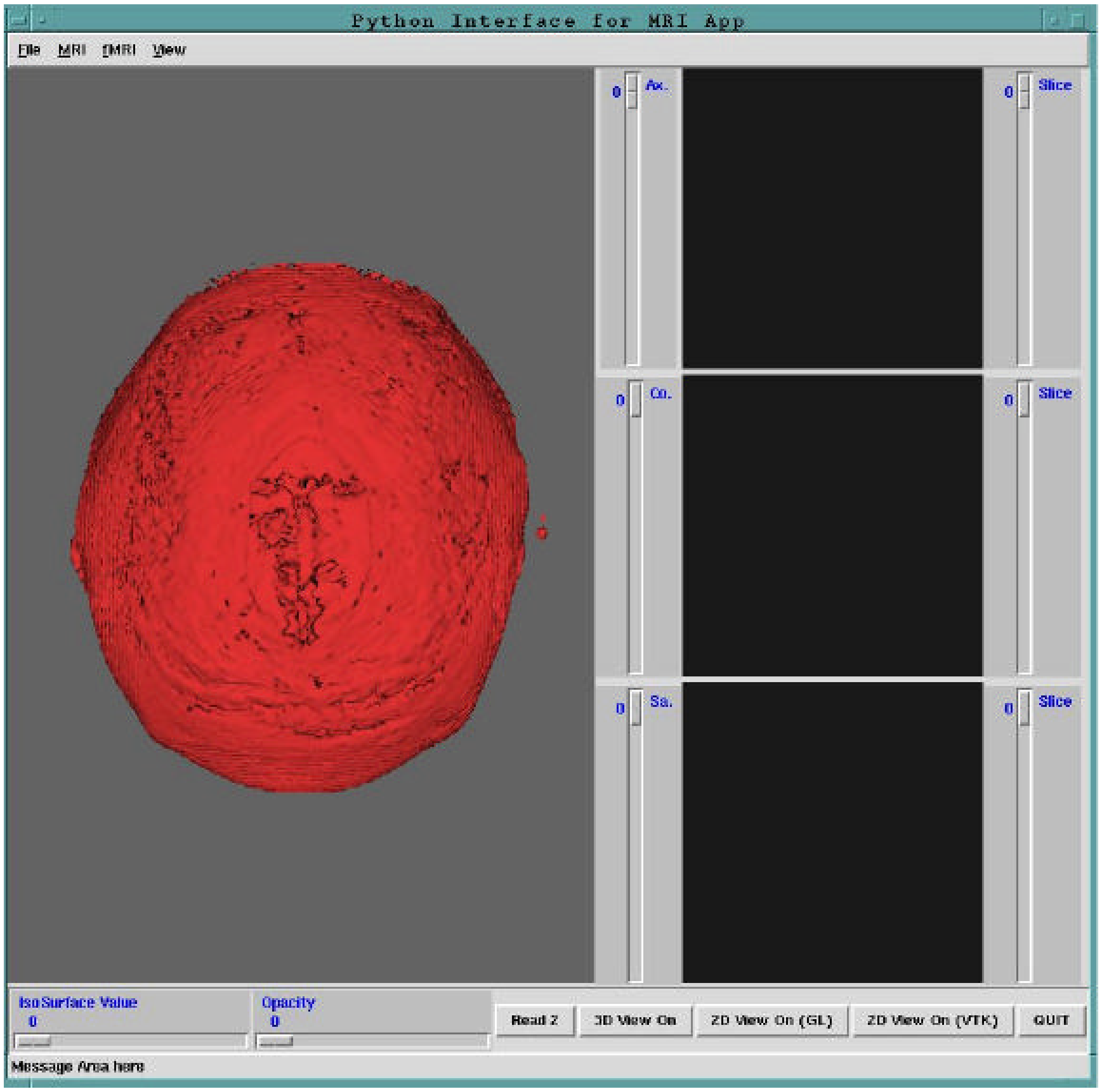
SWIG, which is a tool that wraps a C++ function to be called by a scripting language. SWIG is not limited to use only for Python, but will also wrap C++ functions to be used with a wide variety of scripting languages, such as Perl and Tcl/Tk, and even some non-scripting languages such as Java. To use swig, you simply have to create a .i input file that describes the C++ function that you would like to wrap. I created a shared module from the C++ function to read in the MRI datasets that could be called from the Python GUI. This is where the problems began with using Python and C++ extensions.

Python does not support pointers, so there was a problem with trying to return this array of values from the C++ function that read in the MRI dataset to the Python GUI for rendering. In both the Python and SWIG documentation, it is stated that SWIG should wrap the function in a manner that the array could be returned as a PyObject to the Python application. Unfortunately, after many attempts and many questions submitted to the Python and SWIG mailing lists, this solution never became a reality.

The work-around was to return each of the values read in from the MRI dataset one at a time to Python. This worked fine, but the dataset I was using for testing contained 6,129,456 values, and other datasets could be larger than this. Needless to say, reading this data in and passing it from the C++ function to Python one value at a time was very slow and inefficient. It took between two and three minutes to read in the entire dataset. The original implementation read the dataset into the application and created the vtkStucturedPoints object in less than fifteen seconds. With this we faced the first major drawback of using two different languages to implement this application. After further researching of the capabilities of the Python language, I was able to implement the functionality of reading the dataset in using Python alone. This solved the problem for the present moment. Even though this method was slower than the C++ function, it took less than one minute to read in the dataset, which was good enough to proceed. Yet the larger problem remained: How well would this implementation perform with the computationally intense task of constructing geometry for the visualizations? And if the performance were poor, how would we move the data efficiently between the Python application and C++ extension modules to perform the number crunching? Regardless, the system was functional at this point, and in hopes of a solution or some novel suggestion from the mailing list, we pressed on with the development. At least now we would have the opportunity to experiment with the visualizations in Python since the datasets had been read in.

The next step, now that I had the data into the Python portion of the application, was to begin constructing the 3D pipeline to create the contour surface for the 3D view of the brain. This process was fairly straightforward, since the pipeline was already available in the original implementation; I just had to rewrite the pipeline in the Python syntax. Figure 9, shown below, is a screen shot of the Python MRI application with the 3D contour surface of the human brain visible.

Figure 9: Python MRI Application with 3D view functional

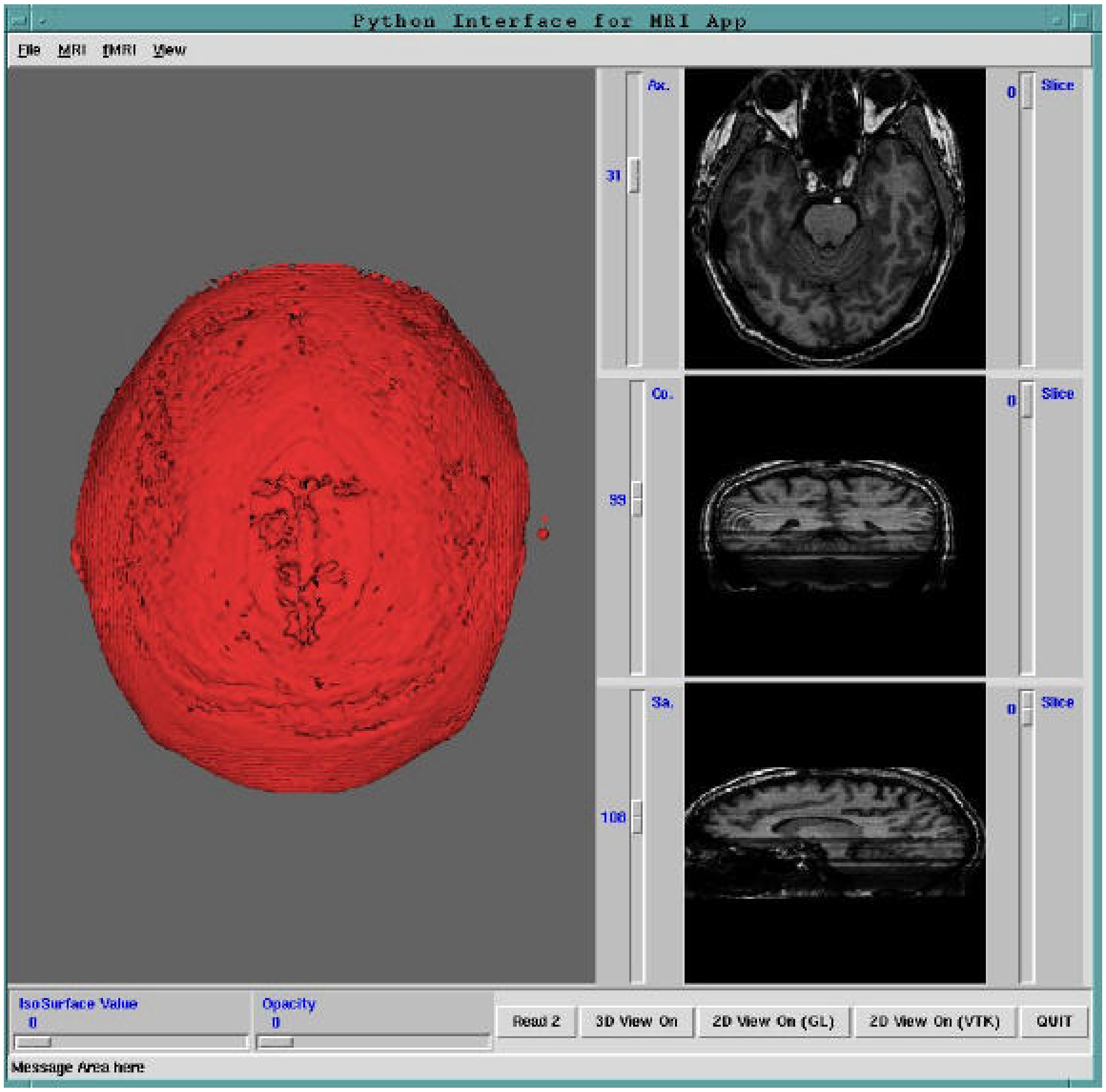


The contour surface is created using the marching cubes algorithm, identically as in the original implementation. This visualization provides all of the same functionality of being able to rotate, scale, and translate the image. Surprisingly enough, the speed at which Python handled the task of rendering this object was very good. There was no noticeable delay when manipulating the object and the performance was quite acceptable.

With Python performing this well with the 3D visualization, I proceeded to implement the 2D views using the vtkTkRenderWindow mentioned previously. To construct the 2D views using VTK, the values from the dataset for each slice were used as input to a vtkTexture object, which handles loading and binding of texture maps [4]. The texture map was then mapped onto a vtkPlanesource for rendering. Oddly enough, even when a lookup table was created to scale the values to produce a gray-scale coloring for the 2D images, VTK was rendering the texture maps in a bright orange-red. This was unacceptable, but fortunately, through posting this problem on the VTK mailing list, a solution was found. The problem was where the lookup table was being assigned in the 2D pipeline. Once this problem was fixed, the 2D views were functional. The process of adding the functionality of updating the 2D views with the appropriate slice according to either the movement of the mouse in one of the other two windows, or the movement of the slider scale, was another exercise in code conversion from C++ to Python. The necessary algorithm and functions were available in the original implementation; I had to simply implement the same methods using Python.

Figure 10, shown below, is a screen shot of the application with both the 3D view and the three 2D views functional. The dataset which is loaded into the application for this screenshot, is the mri2.dset dataset, which is the same dataset used in Figure 6 of the original MRI application. It can be clearly seen from a quick comparison of the two screenshots that the Python implementation results in identical visualizations of the dataset being examined.

Figure 10: Python MRI Application



Despite the visualizations produced being identical, there was a problem with the Python version. The performance when updating the 2D views was extremely poor. When a user moves the slider scales, or clicks the left mouse button in one of the other two 2D views, there is a very noticeable delay in the updating of the 2D view to the correct slice. Worse even, if a user drags the slider scale from the first slice to the last slice, the updating cannot keep up with the movement of the scale, hence the application drags, and the view jumps through only a few of the slices from the beginning of the dataset to the end. This is unacceptable, as the 2D views need to update very smoothly as the user drags the slider scale.

As a solution to speeding up the updating of the 2D views, I pre-computed an array for every possible slice of the dataset for each of the three 2D views. This solved the problem, as now the views would update very quickly and smoothly since the new data simply had to be assigned and rendered. No computation was necessary when the view needed to be updated. But, as with many solutions, this created a new problem. The time taken to pre-compute the arrays containing all of the slices of the dataset turned out to be absolutely unacceptable. This pre-processing step takes between four and five minutes. This time, added to the time it takes to read the dataset into the application, makes the use of this implementation unfeasible in a real-life scenario. No physician would be willing to wait this long for the application to complete its pre-processing step each time they need to use it.

Even though, at the present time, the Python implementation is not a feasible replacement for the current MRI application, the goal of experimenting with an alternate implementation has been achieved. Many valuable lessons were learned from building this application in Python, and groundwork has been set for further work. PyOpenGL is a new tool, and is under continuous development. In the very near future, the problems experienced with this tool may be remedied, and the Python implementation could become a substitute for the C++ implementation.

3.5 Tumor Segmentation and Quantification

The second objective of this project was to extend the current MRI application so that it would provide users with the ability to segment tumors from healthy brain tissue. Once the segmentation was complete, we wanted to provide a means to view a 3D visualization of the tumor within the brain, with the ability to interact with this visualization. Finally, we intended to provide a means of estimating the volume of the segmented tumor, and display this information to the user as they inspect the 3D construction.

Image segmentation is defined as the subdividing of an image into its constituent parts or objects. Developing an algorithm to automatically segment an image is one of the most difficult tasks in image analysis, yet is one of the most crucial to the success of the analysis [6]. In our system, we are not attempting automatic segmentation. The segmentation algorithm is designed to work from an input that will require the user to have some prior expertise in the field of interest. Since this application is being developed for the use of physicians, this prior expertise of the subject area is readily available.

Images produced from MRI scanning are gray-scale images, thus the datasets we are attempting to segment range in value from 0 – 255. Segmenting gray-scale images is typically approached with one of two different strategies: 1) Look for discontinuity in the image or 2) Look for similarity in the image. When searching an image for discontinuity, the goal is to segment the image according to rapid and drastic changes in the values of the pixels. This strategy lends itself well to point, line, and edge detection.

With the second strategy, the algorithms are typically based on some region growing, splitting and merging, or thresholding approach.

Edge detection is the most often used means of searching an image for meaningful discontinuities. An edge is defined as a boundary between regions of an image with distinct intensity properties. This method is most often used when the two regions separated by the boundary are reasonably homogenous, to the point that distinction could not be made between the two by comparing the intensity of each area. The basic strategy of most edge detection algorithms is to apply some local derivative operator to each region of the image to detect the regions where the intensities change rapidly. This technique will result in a set of pixels that define the boundaries of regions in the image. Unfortunately, this set of pixels is typically broken up due to noise or artifacts in the image. To complete the segmentation, a linking algorithm is usually applied to each pixel that underwent segmentation in order to define a continuous boundary between image regions. Brain tumors tend to have very irregular shapes with extremely jagged edges apparent in MRI datasets. Also, using the correct MRI protocol, the tumor pixels do not have similar intensities to the healthy brain tissue. For these reasons, we did not begin our segmentation attempts with edge detection strategies.

Region growing is a popular means of segmenting images on the basis of searching for regions of similar intensity. As one may expect from the name, this approach attempts to 'grow' a region from some starting seed point. This strategy constructs the region being segmented directly, rather than detecting edges and constructing regions based on these edges. Region growing typically produces better results images where distinct edges are not easily identifiable than the edge detection strategy described above [6]. Tumor volumes found in the MRI datasets we are attempting to segment fall into this category, thus implementing a simple region growing algorithm was the plan of action devised.

Region growing algorithms need an input, typically referred to as a ‘seed point’ to begin searching for points in the dataset with similar value. This input can be a single point or a group of points in the dataset. The algorithm must also have some defined way of measuring whether each data point it encounters is part of the region being segmented or not. The algorithm I implemented checks the value at points in the dataset to see if their intensity falls within some tolerance threshold of the region. This threshold is the second input required by the algorithm.

The algorithm begins with the seed point chosen by the user of the system and begins checking the point’s twenty-six neighbors, in 3D space, to see if their intensity value falls within the given threshold of its own value. This process continues recursively, considering the neighbors of each pixel that is added to the region. The function will return control to its calling function once no points defined to be in the region have neighbors not in the region with a value that falls within the give intensity threshold.

The addresses of a pixel’s neighbors, relative to its own address in the array, are computed to determine which pixels the algorithm should investigate. Connectivity of data points in the dataset is of crucial importance to the region-growing algorithm. Simply investigating the intensity of every data point in the array would not only be wasteful, but would yield vastly misleading results. We are only interested in data points that fall within the given intensity threshold and are connected to another point in the region grown from the seed point.

Two problems that arise with region growing algorithms can be the difficulty in selecting an appropriate seed point and the defining of a suitable threshold to determine whether a pixel should be added to the region or not. We are depending upon the user's ability to select the appropriate seed point to begin the algorithm. Once an appropriate seed point has been selected, the threshold can be selected on a trial and error basis by moving a slider scale to indicate the desired threshold. This is an ideal setup because it allows the user to vary the threshold and view how well the segmentation covers the region of the tumor. The user can then vary the threshold up or down, dependent upon the results of the segmentation.

Another issue that is often viewed as a problem with the region growing approach to image segmentation is that these algorithms have the tendency to grow holes in the segmented region. If data points exist within the region of the tumor that have intensity values outside the given threshold, the region growing algorithm will actually grow around this data point, leaving a hole in the tumor. While this can produce undesirable results, in our case this property actually proved very useful in one of the test cases presented later in this paper. The opportunity to exploit this feature became apparent when one of the data sets we tested the algorithm on contained a tumor with a cyst. The region-growing algorithm segmented the tumor, but grew around the cyst. This was not anticipated, but it turns out to be exactly what we need for accurate volume computation on the tumor since the volume of the cyst should not be included. It also provided the opportunity to segment the cyst from the tumor and compute its volume. This is discussed further in the presentation of the results.

To implement the region-growing algorithm, I modified the GUI object, the MRI object, and the MriVtk object. The GUI was modified to include a button for the user to click to run the segmentation procedure. A sliding scale was also added for the user to select the threshold the procedure should use. These two additions to the GUI were placed directly underneath the three 2D views. The 3D pipeline is constructed in the MriVtk object, which had to be altered to include an additional pipeline of objects for the 3D visualization of the brain tumor. This pipeline was created identical to the pipeline for the 3D visualization of the brain itself. The tumor will be represented as a contour surface created from the marching cubes algorithm. A vtkStructuredPoints object was also added to the MriVtk class, which maintains the data identifying which points in the dataset are part of the tumor region and which points are not. This vtkStructuredPoints object is created as the same size as the object containing the MRI dataset read into the application. With the segmentation information being maintained in this manner, two contour surfaces can be constructed, one from each of the vtkStructuredPoints objects, and the resulting visualization will place the segmented tumor value in the accurate location relative to the remaining portion of the brain. I distinguish between the two by coloring the tumor blue and the remaining portion of the brain red.

When a user clicks the 'Segment' button, the appropriate callbacks handle the request by calling the call_region_growing() function which has been added to the mri object. The MRI object is where the actual segmentation occurs. The call_region_growing() function queries the location of the slider scales for each of the 2D views and calculates the corresponding address in the array of the seed value. Thus the location of the sliding scales is the means in which a user can select the seed point. This can be achieved by either moving the sliding scales independently, or by choosing a view where the tumor is most identifiable and moving the mouse to the region of the tumor and clicking the left mouse button. This will automatically update the other two views to the appropriate slices.

Once the address is computed, the region_grow() function is called and sent the address of the seed point. This function has also been added to the mri object. This function extracts the vtkScalars array from the vtkStructuredPoints object containing the MRI dataset that has been read into the application. This function also queries the value from the threshold sliding scale that determines the threshold that the intensity of a data point must fall within to be included as part of the tumor. This function then calls the find_region() function, which is the actual recursive function that performs the segmentation. This function first checks to be sure the seed point is not on a boundary of the dataset and then calls the find_neighbors() function to determine the twenty-six neighbors that need to be investigated. Once this information is found, the function iterates through each of the neighboring data points identifying each as either part of the tumor or not. If the data point is determined to be part of the tumor, the function calls itself and the procedure repeats. If the data point is determined to be outside of the tumor, then the point is marked as visited but not tumor, and the iteration continues with the next neighbor.

After the recursion completes, control is returned to the region_grow() function which then iterates through the dataset and marks the segmented data points as either edge points or interior points. This computation is performed to aid in our volume computation discussed later.

Now the segmented tumor volume has been computed, and whenever the user chooses to turn the 3D visualization on, both the original contour surface of the brain and the contour surface of the segmented tumor will be constructed.

The final feature added to the MRI application was the quantification of the segmented tumor. Another button was added to the GUI of the application that allows the user to run the volume computation routine. The callback function associated with this button calls the CalcVolume() function. This function computes three estimates of the volume of the tumor. The first two estimates are computed by counting the number of voxels the tumor occupies. One estimate includes the edge voxels, while the other excludes the edge voxels. Once the number of voxels of the tumor was determined, this number is multiplied by the voxel size. The voxel size is maintained in the MRI object, and can be altered by the user through a menu option. The voxel size is maintained in cubic millimeters, and we convert to cubic centimeters to report the volume. The third volume estimate was computed by utilizing the vtkMassProperties object, which is supplied by VTK. This object accepts the output from the marching cubes filter and estimates the volume of the resulting object. The vtkMassProperties also provides a means of calculating the surface area of the object, so we calculate this and report it to the user as well. After experimenting with several data sets, and comparing the three volume estimates, it became apparent that the voxel count that excludes the edge voxels was a gross underestimation. Hence, this estimate was dropped, and only the remaining two volume estimates, along with the surface area, are reported on the message bar of the MRI application for the user to view.

3.6 Results

Dr. John Brockway at Presbyterian Hospital in Charlotte, NC, provided all MRI datasets used for experimentation and testing of the segmentation and quantification algorithms. Three of these datasets are presented in this section to display the results we obtained. Each case is identified by the name of the dataset.

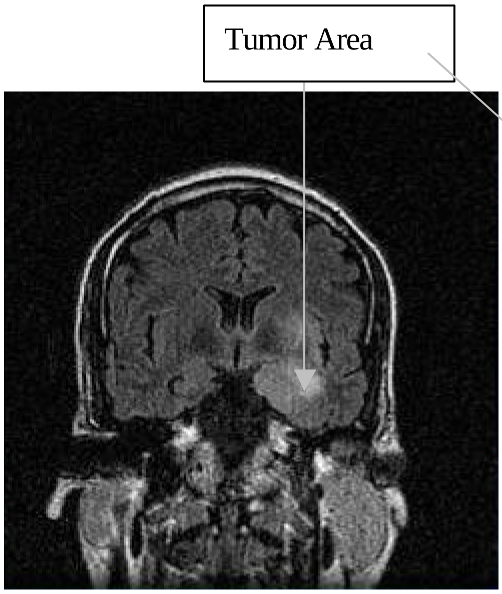
Case 1: jl1Flair.dset

The jl1Flair dataset was received in the .MR format, and there were 30 coronal slices provided. Using a previously developed Tcl script, the series of .MR images were first converted to the SGI .rgb format, and then into the 3D .dset format. The jl1Flair dataset slices were 3 mm thick and the field of view (FOV) was 200x200 mm. All flair datasets are taken as coronal views, as opposed to SPGR datasets, which are taken as axial views. The mri2.dset dataset that is displayed in Figure 6 and Figure 10 is an SPGR dataset. Since the flairs are taken from a different view, the location of the slices corresponding to each of the 2D views in the application is different. For this reason, all screenshots provided henceforth, show the coronal view in the top 2D window, the axial view in the middle 2D window, and the sagittal view in the bottom 2D window.

Figures 11, 12, and 13, show the coronal, axial, and sagittal views, respectively, of the jl1Flair dataset before segmentation. Upon inspecting the coronal view in Figure 11, a white area just to the right of the center of the image should be noticed. In flair datasets, this indicates a tumor mass. This will be the area we attempt to segment.

Figure 11: Coronal slice 15 of jl1Flair.dset

Figure 12: Axial slice 102 of jl1Flair.dset



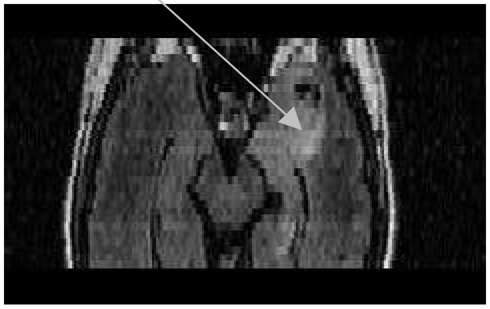
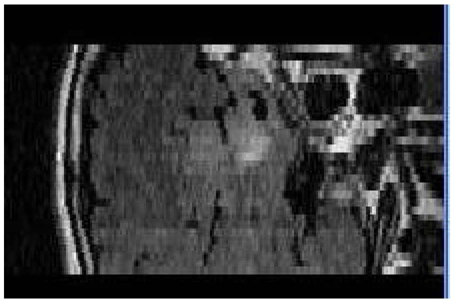


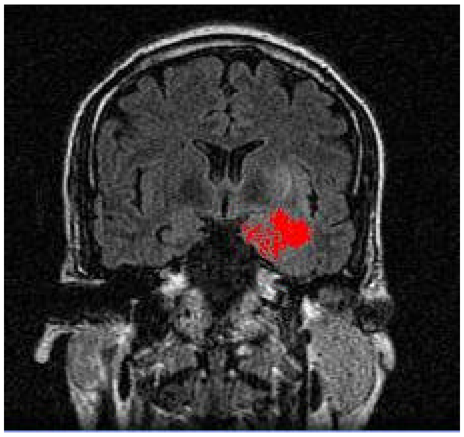
Figure 13: Sagittal slice 156 of jl1Flair.dset



Coronal, axial, and sagittal slices 15, 102, and 156, respectively, were chosen because this seemed to define the best point in the dataset to use as the seed point for the region growing algorithm. This point was decided upon by choosing a coronal slice with a good view of the tumor, and moving the mouse to what appeared to be the center of the tumor. The other two views were updated according to the location of the mouse in the coronal view. The segmentation was run several times, varying the value of the segmentation threshold, and the threshold that appeared to yield the best results was a value of 55. With these parameters set, the two volume estimates produced were 3.07 cm^3 and 2.54 cm^3. The first estimate comes from the voxel count, which includes the edge voxels, and the second estimate comes from the volume computed by the vtkMassProperties object. The surface area was computed as 28.27 cm^2. Figure 14, 15, and 16 show the same coronal, axial, and sagittal views of the jl1Flair dataset with the tumor segmented in red.

Figure 14: Coronal slice 15 of jl1Flair.dset

Figure 15: Axial slice 102 of jl1Flair.dset



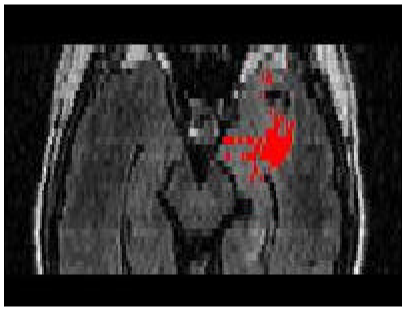
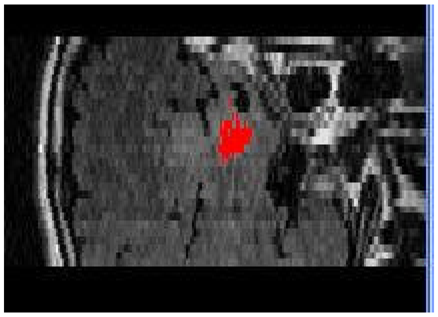


Figure 16: Sagittal slice 156 of jl1Flair.dset



Figures 17, 18, 19, and 20 show the 3D construction of the dataset. Figure 17 is a frontal coronal view of the dataset before segmentation. Figure 18 is the same view with the segmented tumor now shown in blue. Figure 19 shows the image rotated slightly counter-clockwise around the vertical axis. Figure 20 is a view from the bottom of the image.

These different perspectives of the image give the user a better understanding of the location of the tumor relative to the rest of the brain. An idea of the approximate shape and size can also be discerned from these images in conjunction with volume estimates the application reports.

Figure 17: Frontal view of 3D construction before segmentation – jl1Flair.set.

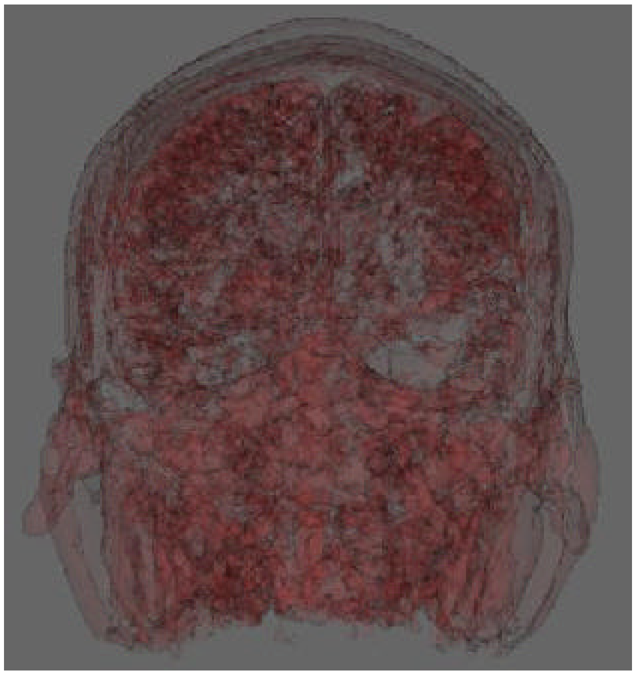


Figure 18: Frontal view of 3D construction, segmented tumor in blue – jl1Flair.dset.

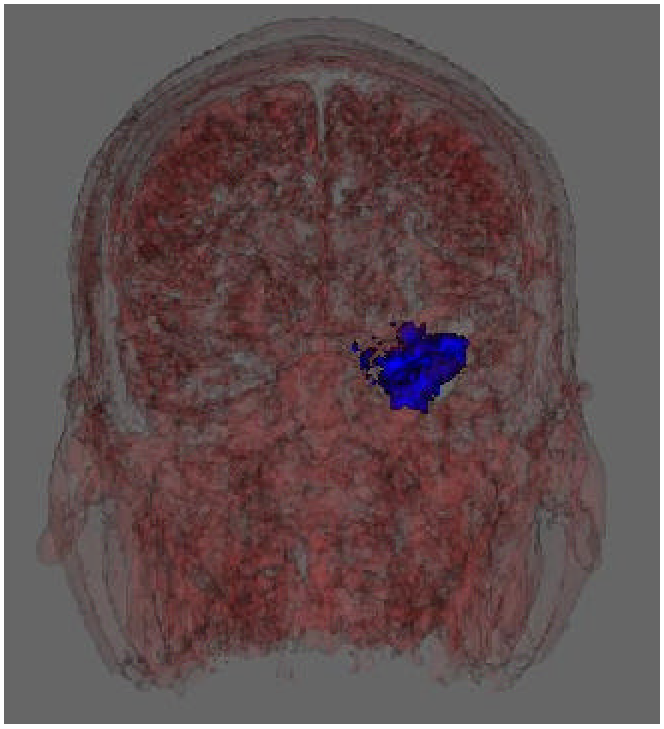


Figure 19: 3D construction slightly rotated around vertical axes – jl1Flair.dset.

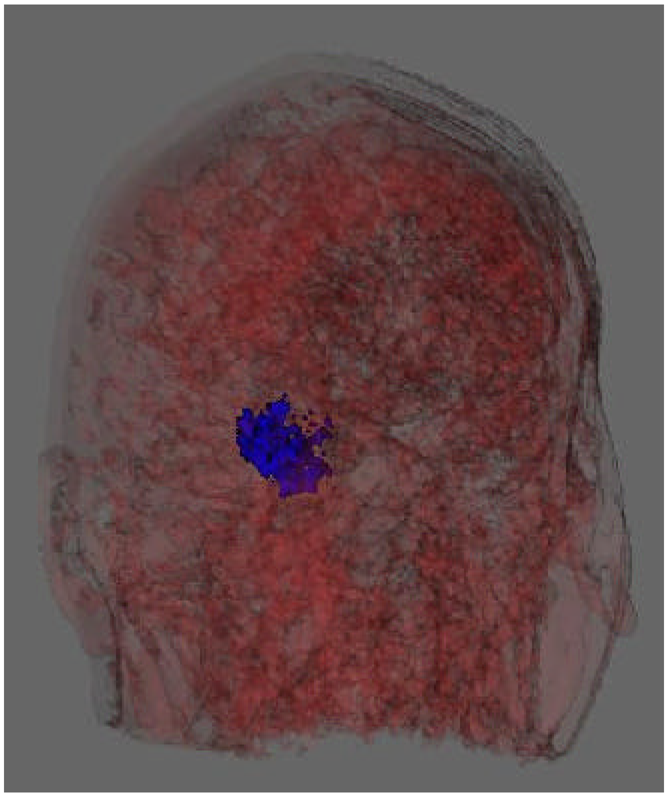
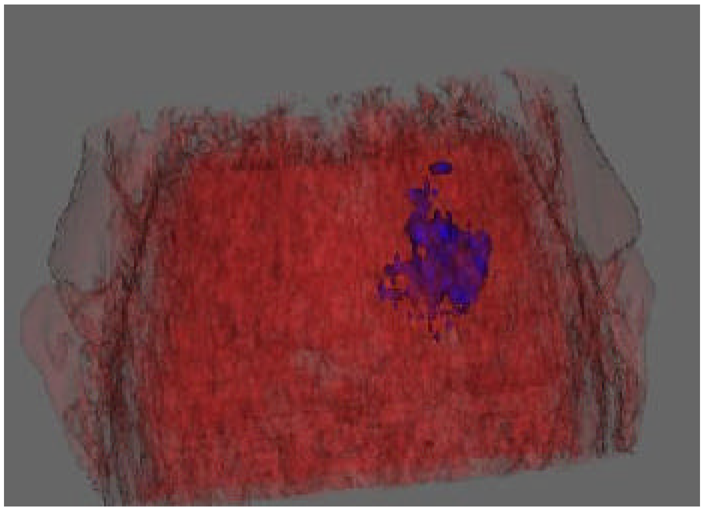


Figure 20: Bottom view of 3D construction – jl1Flair.dset



Case 2: Flair441.dset

The Flair441 dataset was received in JPEG format, and there were 24 coronal slices provided. The JPEG images were first converted to the PPM format using XV. The PPM images were then converted to the 3D .dset format using the previously developed ppm2dset3D converter. The FOV for this dataset was 220x220 mm and the thickness of each slice was 3mm.

The optimal seed point was defined by coronal slice 7, axial slice 146, and sagittal slice 161. Figure 21 shows these three slices, from top to bottom, before segmentation. The tumor mass is clearly identifiable in the upper right portion of the coronal image as a large white area. Figure 22 shows these same three slices after the segmentation has been completed, with the tumor identified in red.

The threshold value used for segmentation in the Flair441 dataset was 65. The volume estimates calculated were 26.83 cm^3 and 26.08 cm^3. The first estimate being the voxel count, including edge voxels, and the second estimate coming from the vtkMassProperties object. The surface area computed was 83.99 cm^2.

As would be expected from viewing the images, the volume of the tumor in this dataset is much larger than the volume of the tumor in the jl1Flair dataset. Figure 23 is an image of the 3D view of the dataset before segmentation and Figure 24 is the same view with the tumor segmented in blue. Figure 25 provides a side view of the dataset and Figure 26 shows a view from the top of the head.

Figure 21: Coronal slice 7, Axial slice 146, Sagittal slice 161 before segmentation – Flair441.dset

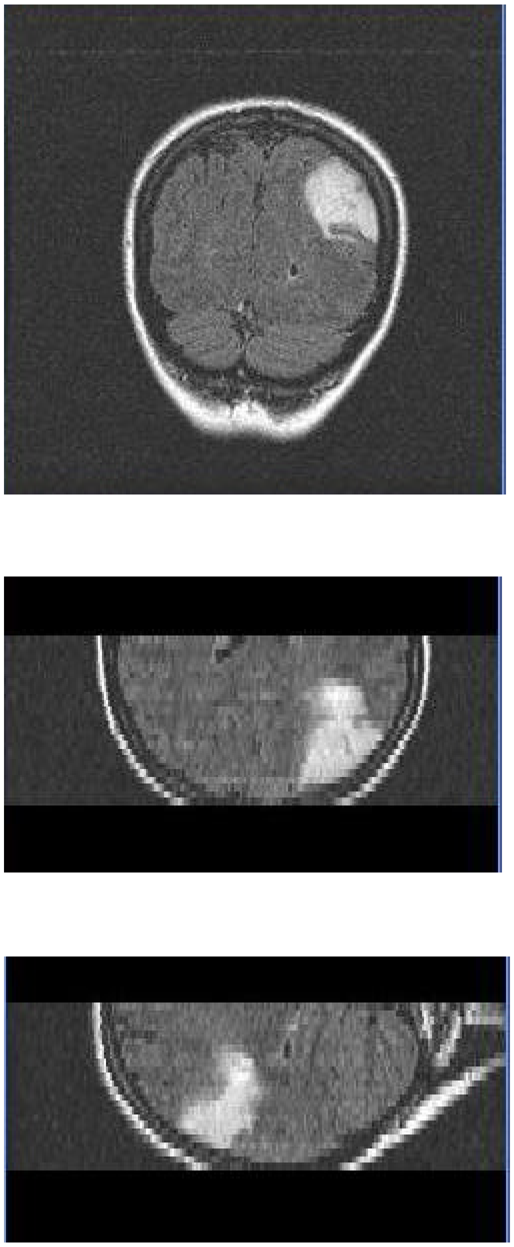


Figure 22: Coronal slice 7, Axial slice 146, Sagittal slice 161 after segmentation – Flair441.dset

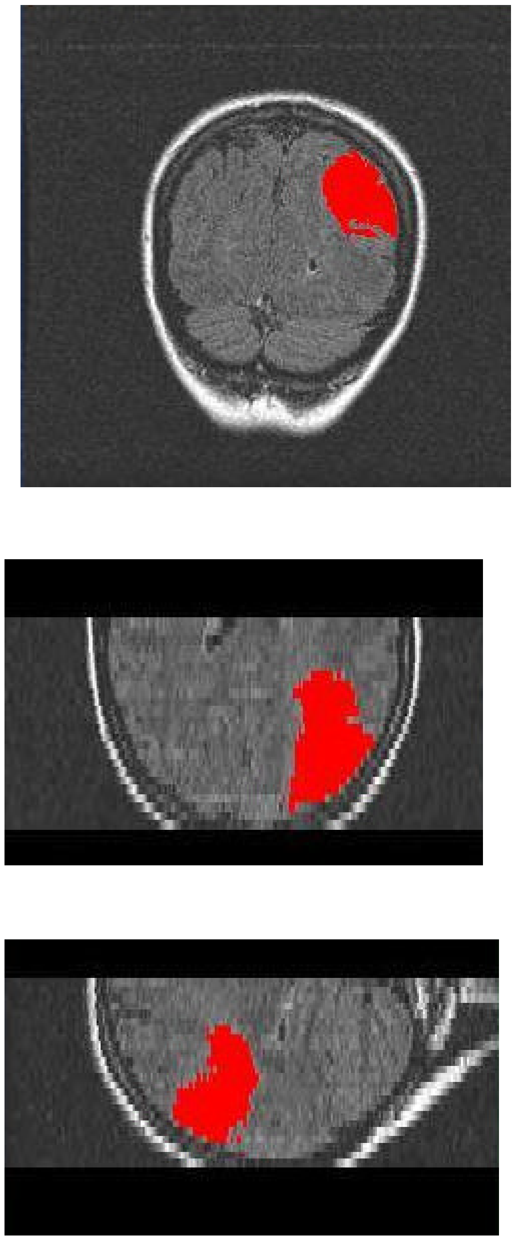


Figure 23: 3D construction before segmentation – Flair441.dset

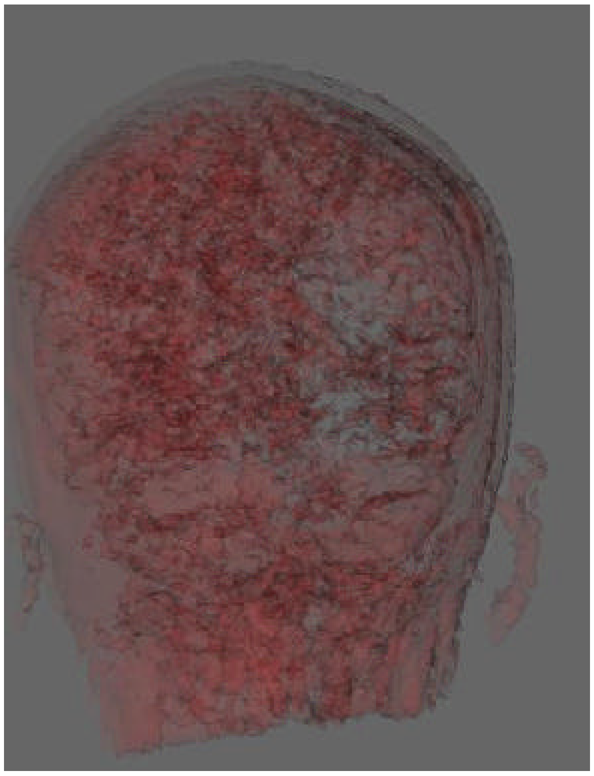


Figure 24: 3D construction with tumor segmented in blue – Flair441.dset

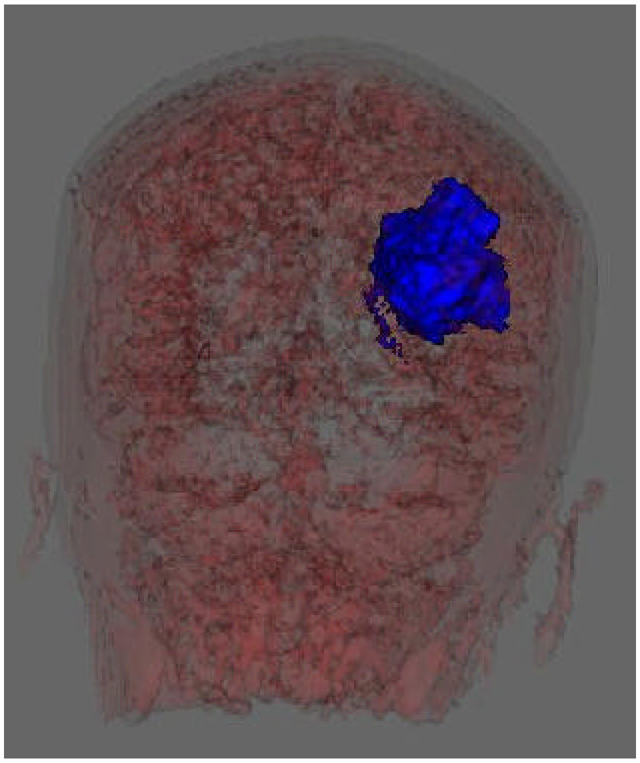


Figure 25: Side view 3D construction with segmented tumor – Flair441.dset

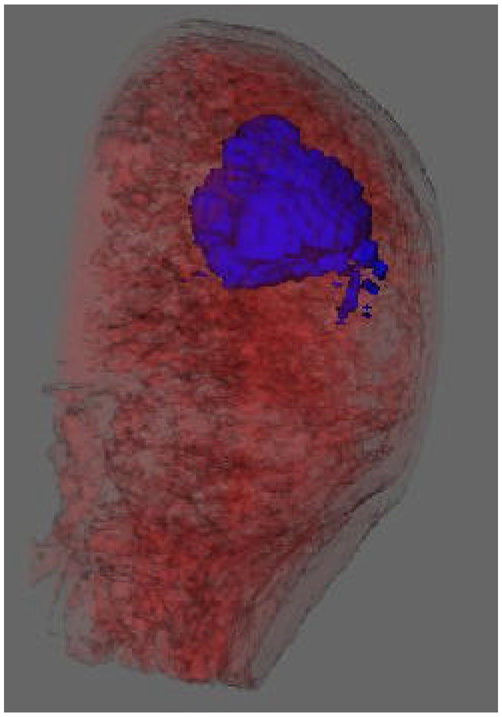
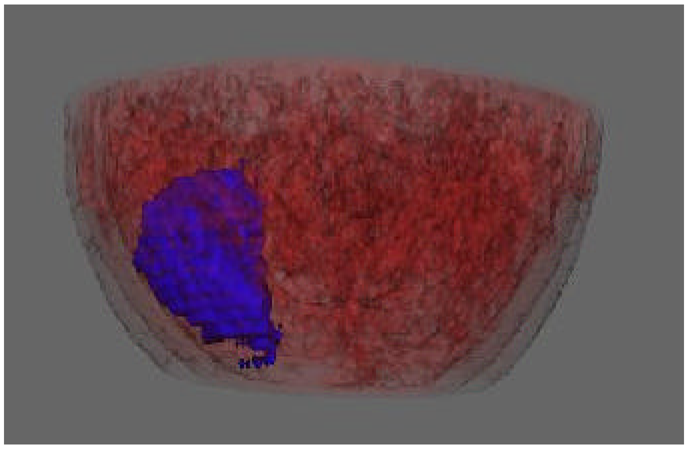


Figure 26: View from top of head of 3D construction – Flair441.dset



Case 3: s7Flair.dset

The s7Flair dataset was received in the .MR format and similar to the jl1Flair.dset, it was converted from a series of .MR's to a series of SGI .rgb's and then to the 3D .dset format. There were 18 coronal slices provided. The FOV for this dataset was 200x200 mm and the thickness of each slice was 3mm.

The optimal seed point was defined by coronal slice 14, axial slice 183, and sagittal slice 81. Figure 27 shows these three slices, from top to bottom before segmentation. The tumor mass can be seen in the top left portion of coronal slice 14. Figure 28 shows these same three slices after the segmentation has been completed, with the tumor identified in red.

The threshold value used for segmentation in the s7Flair dataset was 55. The volume estimates calculated were 29.81 cm^3 and 28.67 cm^3. The first estimate being the voxel count, including edge voxels, and the second estimate coming from the vtkMassProperties object. The surface area computed was 153.81 cm^2.

Figure 29 is an image of the 3D view of the dataset before segmentation and Figure 30 is the same view after the tumor has been segmented. Figure 31 provides a side view of the 3D visualization and Figure 32 shows the image from the top of the head. From inspection of these images, one can see that this tumor appears larger than either of the two previous tumors, which corresponds to the volume estimates we predict.

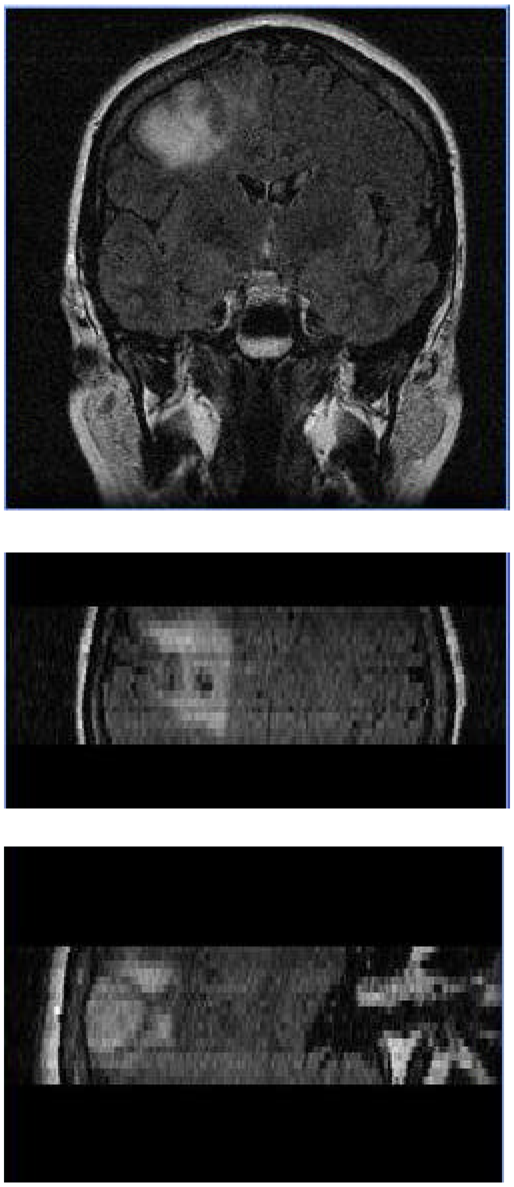


Figure 27: Coronal slice 14, Axial slice 183, Sagittal slice 81 before segmentation – Flair441.dset

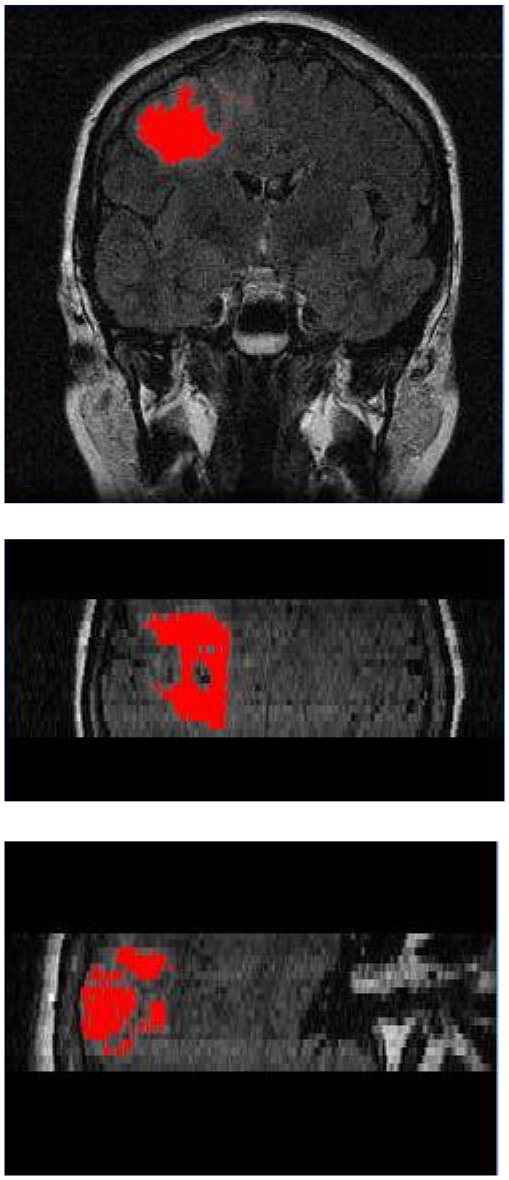


Figure 28: Coronal slice 14, Axial slice 183, Sagittal slice 81 after segmentation – Flair441.dset

Figure 29: 3D construction before segmentation – s7Flair.dset

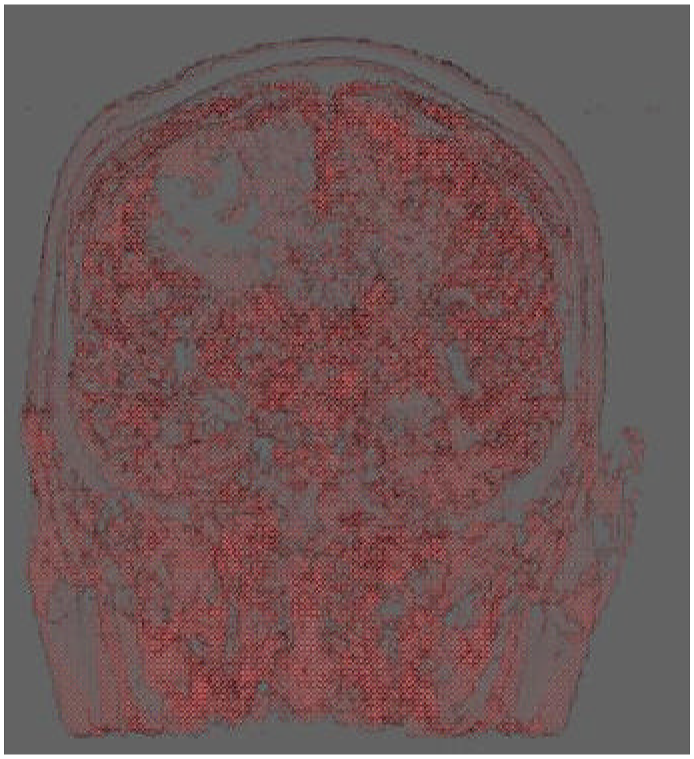


Figure 30: 3D construction with tumor segmented in blue – s7Flair.dset

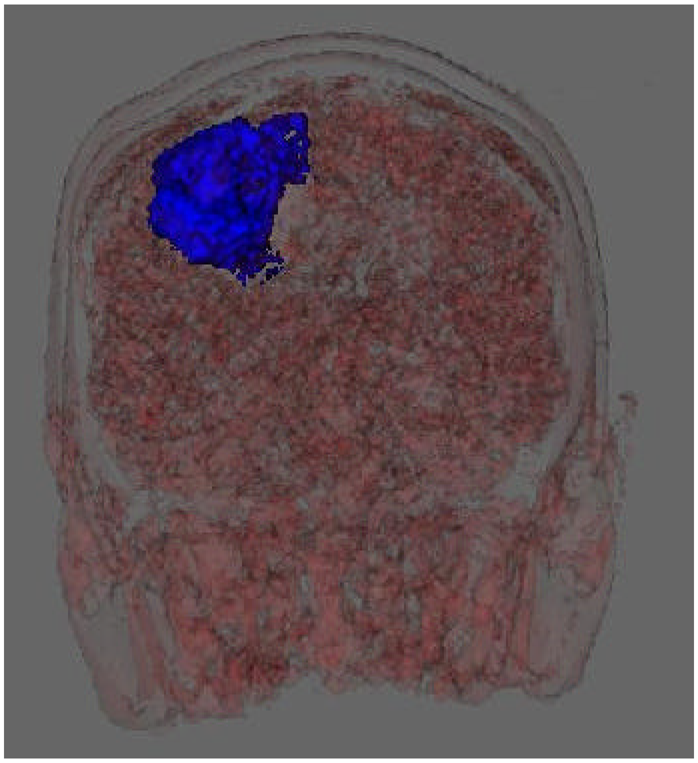


Figure 31: Side view 3D construction with segmented tumor – s7Flair.dset

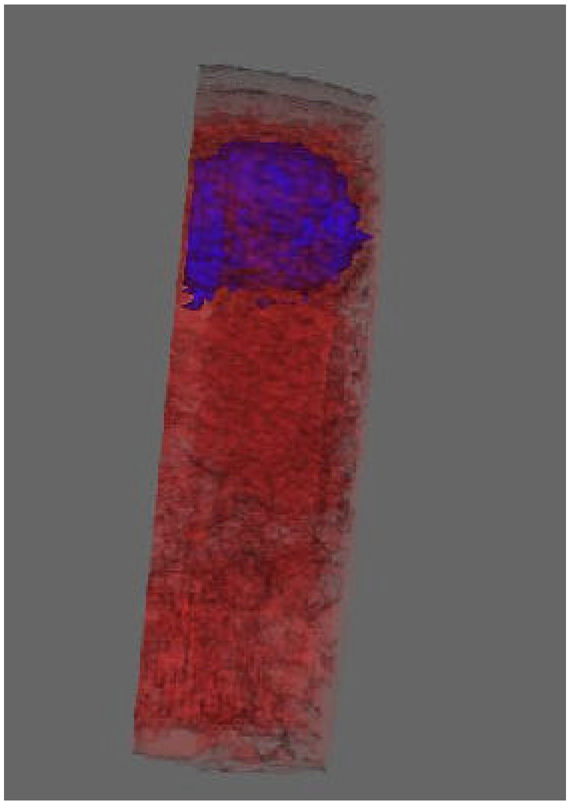
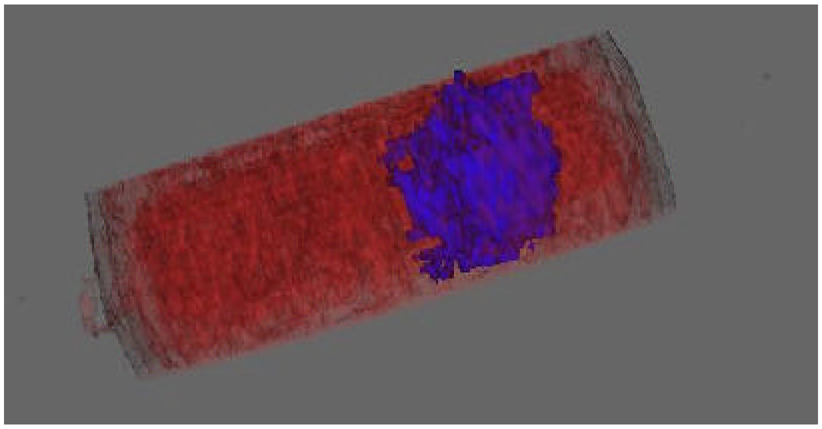


Figure 32: View from top of head of 3D construction – s7Flair.dset



Upon inspection of the axial slice of the s7Flair dataset in Figure 28, we noticed the algorithm had left a whole in the middle of the tumor. This area in the center of the tumor has a much lower intensity value from the other portions of the tumor. When these results were shown to Dr. Brockway, he informed us that this was actually a cyst in the middle of the tumor and requested that we go one step further and segment the cyst from the tumor.

The best seed point for the cyst was defined by coronal slice 8, axial slice 188, and sagittal slice 101. The threshold value used for this segmentation was 17. The volume calculation for the cyst yielded estimates of 0.71 cm^3 and 0.54 cm^3. The first estimate being the voxel count, including edge voxels, and the second estimate coming from the vtkMassProperties object. The surface area computed was 7.52 cm^2.

Figure 33 shows the cyst segmented in the three 2D slices described above and Figure 34 shows the 3D visualization with the cyst segmented in blue.

Figure 33: Coronal slice 8, Axial Slice 188, Sagittal slice 101, with cyst segmented in red – s7Flair.dset

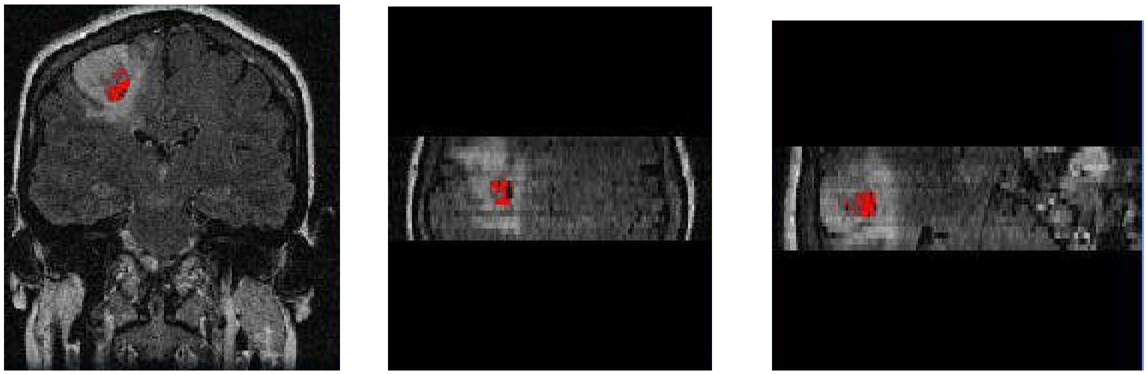


Figure 34: 3D construction with segmented cyst in blue – s7Flair.dset

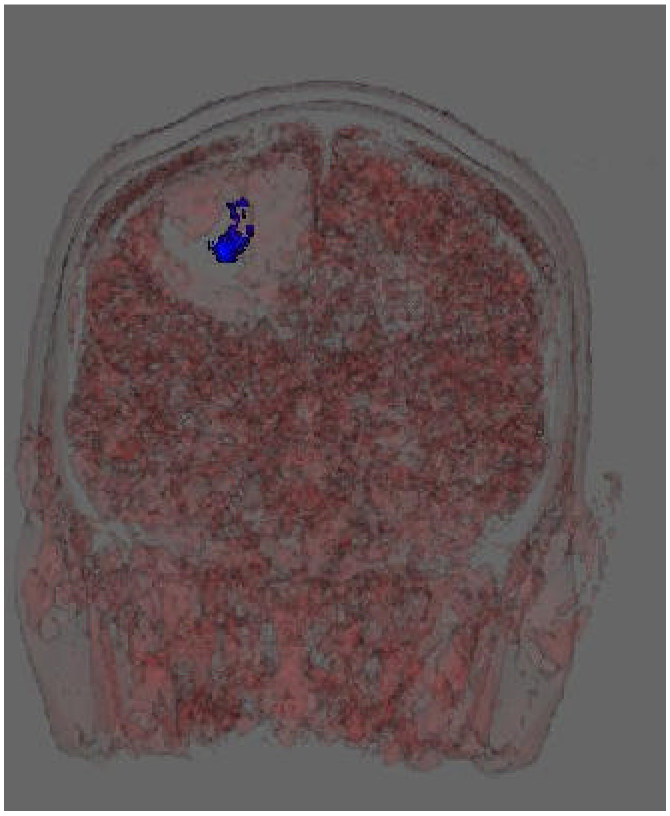


Figure 35 provides a table to summarize our results. Each of the three datasets described above are listed with their estimated volume and surface area. The cyst from the s7Flair dataset is also included.

Figure 35: Table summarizing results

Dataset	**Volume (cm^3)** **From Voxel Count**	**Volume (cm^3)** **From vtkMassproperties**	**Surface** **Area (cm^2)**
jl1Flair.dset	3.07 cm^3	2.54 cm^3	28.27 cm^2
Flair441.dset	26.83 cm^3	26.08 cm^3	83.99 cm^2
s7Flair.dset	29.81 cm^3	28.67 cm^3	153.81 cm^2
Cyst from s7Flair.dset	0.71 cm^3	0.54 cm^3	7.52 cm^2

3.7 Conclusions and Future Work

With the MRI application being extended to segment tumors and calculate estimated volume of these tumors, there exist great opportunity for extending this work even further. One immediate need realized from case 3 described in the results section, is to have the capability to perform more than one level of segmentation. It would prove useful to be able to segment objects, such as a cyst, from the tumor volume that has been segmented from the brain, and display both segmented objects at once. These objects could be made distinguishable by rendering each in a different color.

Further work should also be performed in refining the segmentation algorithm and perhaps replacing it with a more sophisticated method of segmentation. One possibility would be to combine the current region growing approach with an edge detection strategy. A combination of the two is often used in order to yield more acceptable segmentation results [6]. Another possibility would be to experiment with the Implicit

Snakes approach described in [12], which could provide a means to progressively determine the surface of the tumor. This could result in more accurate edge detection method, which would in turn increase the accuracy of the volume computation.

As this application evolves and continues to be improved upon, there is great potential for its use as a tool for physicians. With accurate segmentation and quantification of tumor volumes, physicians would be able to better prepare their strategy before performing surgery, and could save a life where otherwise would not have been possible

Chapter 4

Summary

Mobile networking and medical image analysis are both exciting and interesting research domains. Data visualization techniques have proved themselves invaluable in aiding in analysis in both of these research areas. There are many interesting problems waiting to be solved in each field, all of which data visualization would be a useful tool.

Over the course of the past academic year, Mobvis and the MRI application have both been extended to offer features found useful in the areas of research they support. Through working with these applications, I had the opportunity to learn numerous valuable lessons about software development and design in general. In addition, working with these applications provided the opportunity to further my knowledge in data visualization techniques, and grow my interest in this area.

References

[1] T.A. Dahlberg, K.R. Subramanian, ``Visualization of Mobile Network Simulations", *Simulation, The Society for Computer Simulation and Modeling*, Vol 77(3-4), pp. 128-140,Sept/Oct. 2001.

[2] K.R. Subramanian, T. Dahlberg, ``Congestion Control in Mobile Networks", *Proceedings of Information Visualization 2000 (INFOVIS 20000) Late Breaking Hot Topics,* Oct 9-10, 2000, Salt Lake City, UT, IEEE Computer Society.

[3] T.Dahlberg, K.R. Subramanian, ``Visualization of Real-Time Survivability Metrics for Mobile Networks", *ACM SWIM 2000, Third ACM International Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, Boston, MA., August 2000.

[4] Shroeder, W., Martin, K. and Lorensen, B. *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics,* Prentice Hall, Inc., Second Edition, 1998.

[5] Ed Huai-hsin, John Riedl, Phillip Barry, and Joseph Konstan, "Principles for Information Visualization Spreadsheets", *IEEE Computer Graphics and Applications,* July/August 1998, pp. 30-38.

[6] Gonzalez, Rafael C., Woods, Richard E. *Digital Image Processing,* Addison-Wesley Publishing Company, 1993.

[7] Tim McInerney, Demetri Terzopoulos, "Deformable Models in Medical Image Analysis", *Medical Image Analysis,* 1996.

[8] Lutz, Mark. *Programming Python, 2nd Edition,* O'Reilly, 2001.

[9] Beazley, David M. *Python Essential Reference,* New Riders Publishing, 2000.

[10] Heller, Dan, Ferguson, Paula M., *Motif Programming Manual for OSF/Motif Release 1.2, Volume Six A,* O'Reilly & Associates, 1994.

[11] Fountain, Antony, Ferguson, Paula, *Motif Reference Manual for Motif 2.1 Volume Six B,* O'Reilly & Associates, 2000.

[12] T.S. Yoo, K.R. Subramanian, ``Implicit Snakes: Active Constrained Implicit Surfaces", *Medical Image Computing and Computer-Assisted Intervention (MICCAI2001),* October 14-17, 2001, Utrecht, Netherlands.

[13] William E. Lorensen, Harvey E. Cline, “Marching Cubes: A High Resolution 3D Surface Construction Algorithm”, *Computer Graphics,* Volume 21, Number 4, July 1987.