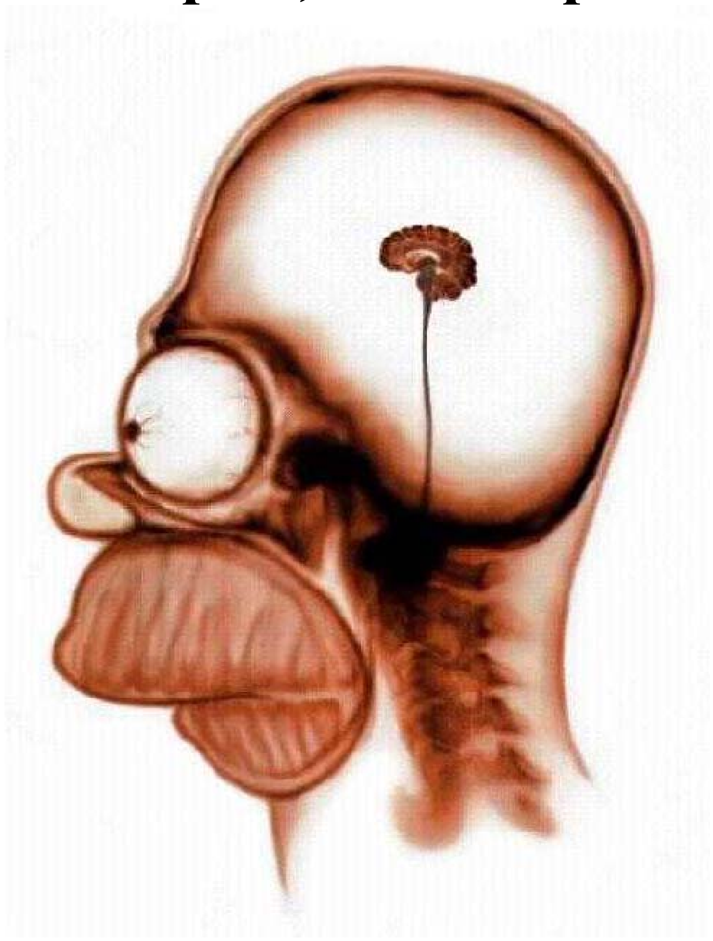
18-511, Spring 2003
Group 13, Final Report



Color Your Head
(MRI Segmentation)

Jerry Tsai (jtsai@andrew.cmu.edu)
Yuchun Wang (yuchunw@andrew.cmu.edu)
Jeff Yang (jayang@andrew.cmu.edu)

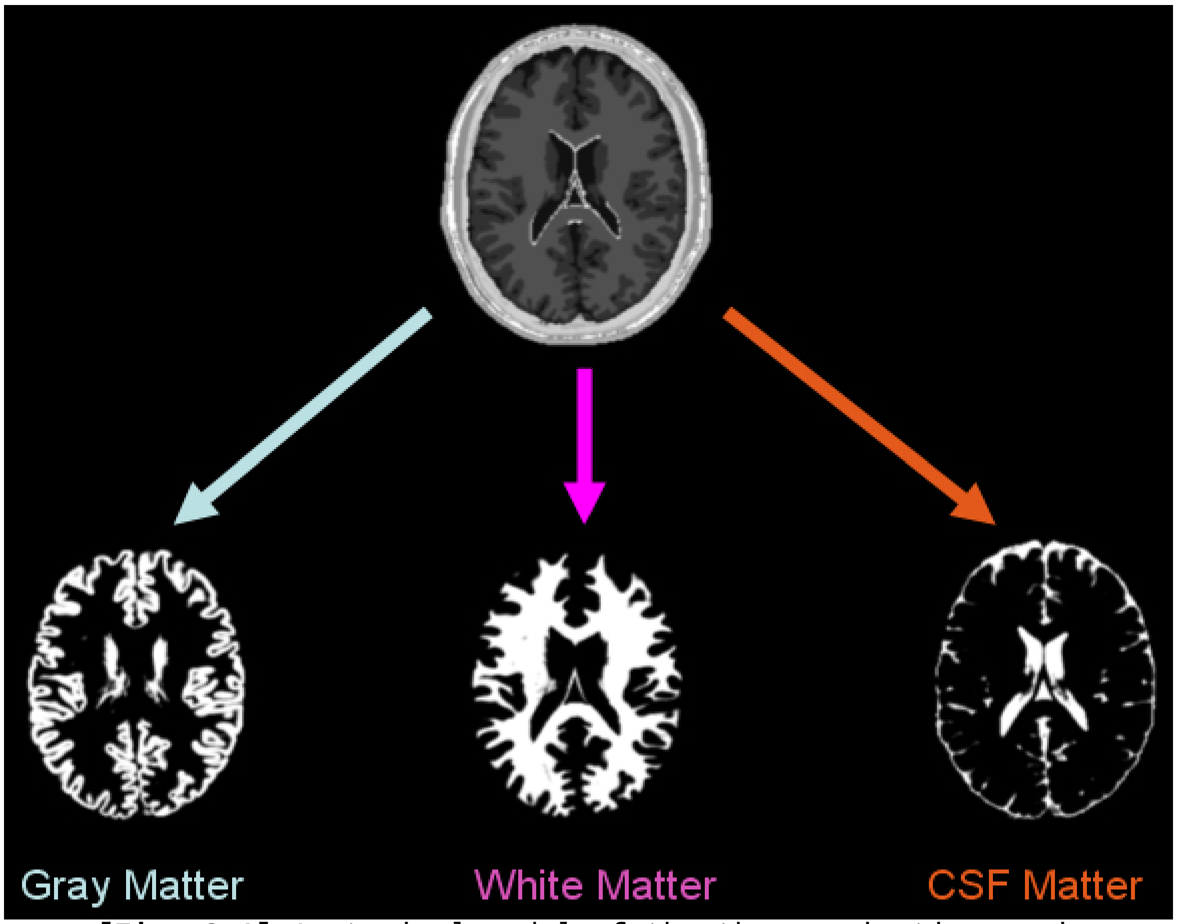
May 5th, 2003

Table of Contents

1. Introduction

The problem we are addressing is accurate detection of physiological structures, specifically the composition and structure of the human brain, in MRI (Magnetic Resonance Imaging) scans. This is generally called "segmentation" because we are programmatically identifying several regions of interest in a target image. Several techniques we will utilize include linear correlation, filtering, error correction, thresholding, and finally classification. We will be using preprocessed images of the head: radiologists would typically look at these images. Many of the relevant software packages available to the public are the result of academic research, and we believe that we can contribute to automated segmentation of the brain.

2. Project Goal



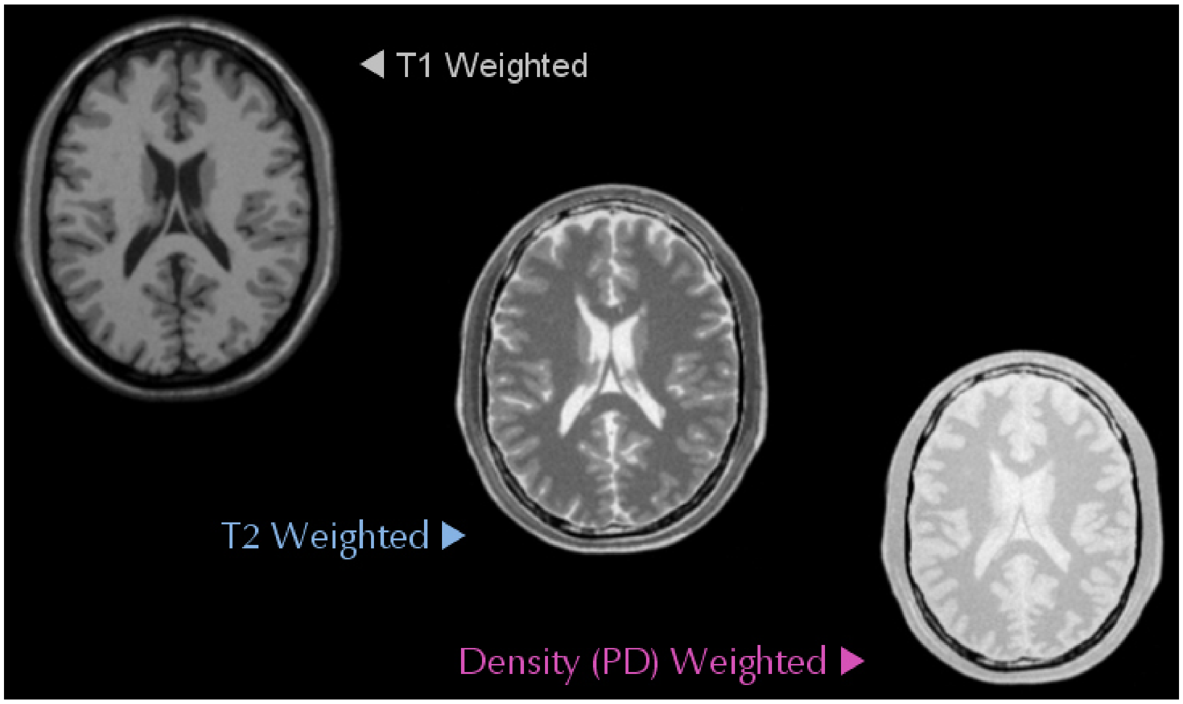
[Fig. 2.1] Anatomical model of the three main tissues in the brain [12]

3. Background on MRI

In MRI, the spin properties of nuclei are exploited by placing them in a high-energy magnetic field. They then align along this field and will respond to orthogonal RF pulses at a specific resonance, the Larmor frequency, given a certain magnetic field strength (usually 1.5 Tesla). The Larmor frequency for a proton (^1H) is about 64 MHz in a 1.5 Tesla field. For ^{13}C, the frequency is about 16 MHz. Combine this with decay constants called T_1 and T_2 (related to spin-lattice and spin-spin interactions in the body), and we can read out signal intensity according to the atomic composition of the material. The signal intensity can be naively characterized by the equation:

$$SI = N * \exp(-T_E/T_2) * [1 - \exp(-T_R/T_1)]$$

Where N is the proton density of the unit area, TE and TR are defined by the MRI operator, T_1 is the spin-lattice constant, and T_2 is the spin-spin constant. The signal received due to the atomic composition of a region corresponds directly to macroscopic structures such as fatty tissue, blood, gray matter, and other structures. The actual recorded image needs to be Fourier transformed to become recognizable. Also, magnetic field inhomogeneities, distortion due to movement of the head, and background noises may be present in the transformed images, but there are many post-processing techniques to clean and enhance the image. Our objective is to segment these images of the brain into distinct categories such as white matter, grey matter, and cerebrospinal fluid.



[Fig. 3.1] Images of the same slice of brain under different time weighting. [12]

4. Prior Work

4.1 Prior 18-551 Work

The group did not find any prior 18-551 projected related to MRI segmentation, thus the group had to search through the web for suitable algorithms.

4.2 Other Groups Work

The idea of MRI segmentation is not new to the field of MR Imaging thus several commercialized softwares are readily available for use. Although these softwares are out there the actual algorithms being used are undocumented on their websites. The group was fortunate to find a paper by M. Stella Atkins and Blair T. Mackiewich, titled *Fully Automated Hybrid Segmentation of the Brain*, which describes the fully automated method of segmenting MRI brain images. From the Atkins' paper the group was able to learn step-by-step methods to segment the brain [1]. It calls for the removal of the background using an intensity histogram, then a mask of the intracranial boundary is created to extract only the brain, and final segmentation is applied. This became the foundation of the group's algorithm.

There were actually several segmentation technique suggest in the Atkins' paper. One method calls for a combination of statistical classification of brain tissues. This method reduces the effect of radio frequency inhomogeneity, but requires some interaction to provide tissue training pixels and computationally expensive. Another method suggested was the atlas method, but one manual segmentation of the brain is required. The last method briefly discussed is a probabilistic atlas method for automatic segmentation of gross anatomical structures of the brain. Through the web the group was able to find a clustering algorithm by Dan Pelleg and Andrew Moore, *Accelerating Exact k-means Algorithms with Geometric Reasoning*, which can be used in the segmentation of the brain. Pelleg uses a *kd*-tree data structure, with sufficient statistics stored at eat node, to reduce the large number of nearest-neighbor queries. An analysis of the geometry of the cluster centers reduces the computation.

4.3 Available Code

There were no prior related 551 projects and no available code from the web the group can use so most of the C code was written from scratch. The group used the majority of the code from lab 3 as a frame to write the program. There were only two functions found on the web. One function is QuickSort [3], which the group used to perform the median filter, and the other code was for *k*-means clustering [2]

5. Dataset

5.1 From Where

The group found three reliable options to get the dataset from, which the group has analyzed their pros and cons:

1) **Center of Cognitive Brain Imaging @ CMU [13]**
   - *Pros*
     i) Unlimited access to a large inventory of MRI images
     ii) Images of the brain are of the horizontal cross-section
   - *Cons*
     i) Only contain T1 and T2 images
     ii) T2 weighted images are functional images.
     iii) Image files are Big Endian (Need to reverse the bytes if we want to use it)

2) **Internet Brain Segmentation Repository (IBSR) [14]**
   - *Pros*
     i) Wide variety of MRI brain images
     ii) 18 datasets, 126 images in total
     iii) Contains both images of the normal brain and brain with multiple sclerosis
   - *Cons*
     i) Images of the brain are of the vertical cross-section (our filters might not be able to precisely extract out the brain)

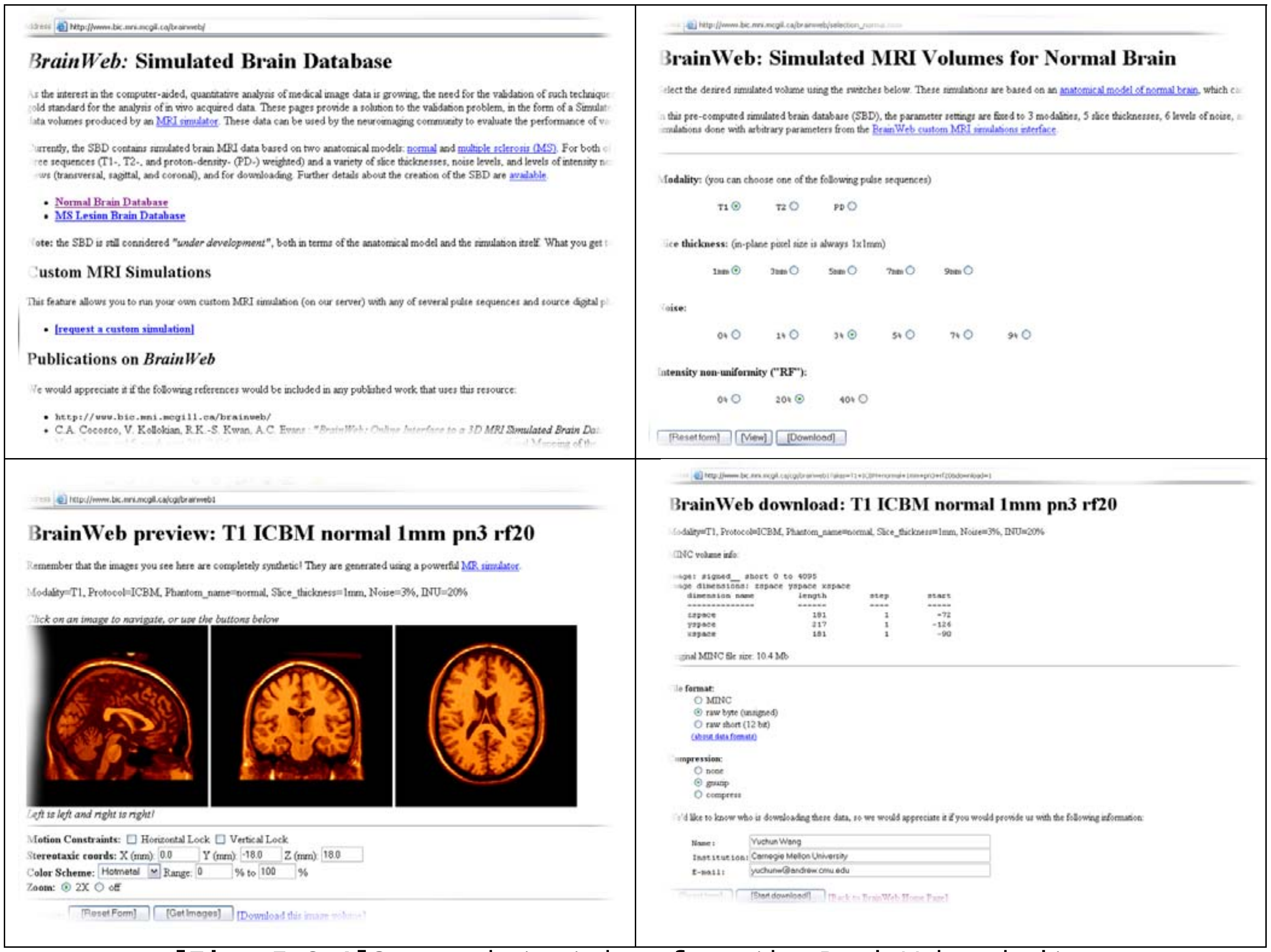
3) ***BrainWeb:* Simulated Brain Database [12]**
   - *Pros*
     i) Realistic MRI images
     ii) Specifically designed to evaluate the performance of various image analysis methods in a setting where the truth is known
     iii) Provides corresponding anatomical model (Fig, 2.1, Allows the group to check the accuracy of the algorithm)
     iv) Simulates both normal brain and brain with multiple sclerosis
     v) Provides correct orientation of the brain cross-section
     vi) Have a selection of T1, T2, and density weighted images
     vii) Choice of noise level
     viii) Do not have to mess around with the file
   - *Cons*
     i) Not actual laboratory MRI brain scans.

   The group's choice was to use the dataset from BrainWeb due to its advantages over the other options.

5.2 Training Set / Test Set

The dataset from BrainWeb comes in "full 3-dimensional data volumes [that] have been simulated using three sequences (T1-, T2-, and proton-density- (PD-) weighted) and a variety of slice thicknesses, noise levels, and levels of intensity non-uniformity" [12]. Each slice is a grayscale brain image of 181 x 217 pixels, with each pixels being 16 bits (unsigned char). Twelve slices in total were extracted from a normal brain simulation and ms brain simulation. The normal brain dataset consists of slice #100 to #120 with 5-slice intervals, and the ms brain dataset consists of slice #90, #95, #97, and #100 - #120 with 5-slice intervals.

In the training set we used slice #100 and #120 for both the normal brain and ms brain. The other slices were used for the test set.



http://www.bic.mni.mcgill.ca/brainweb/

BrainWeb: Simulated Brain Database

s the interest in the computer-aided, quantitative analysis of medical image data is growing, the need for the validation of such technique old standard for the analysis of in vivo acquired data. These pages provide a solution to the validation problem, in the form of a Simulat ata volumes produced by an MRI simulator. These data can be used by the neuroimaging community to evaluate the performance of v

urrently, the SBD contains simulated brain MRI data based on two anatomical models: normal and multiple sclerosis (MS). For both ree sequences (T1-, T2-, and proton-density- (PD-) weighted) and a variety of slice thicknesses, noise levels, and levels of intensity n ws (transversal, sagittal, and coronal), and for downloading. Further details about the creation of the SBD are available.

- Normal Brain Database
- MS Lesion Brain Database

ote: the SBD is still considered *"under development"*, both in terms of the anatomical model and the simulation itself. What you get

ustom MRI Simulations

This feature allows you to run your own custom MRI simulation (on our server) with any of several pulse sequences and source digital p

- [request a custom simulation]

Publications on *BrainWeb*

e would appreciate it if the following references would be included in any published work that uses this resource:

- `http://www.bic.mni.mcgill.ca/brainweb/`
- C.A. Cocosco, V. Kollokian, R.K.-S. Kwan, A.C. Evans : *"BrainWeb: Online Interface to a 3D MRI Simulated Brain Da*

http://www.bic.mni.mcgill.ca/brainweb/selection_normal

BrainWeb: Simulated MRI Volumes for Normal Brain

elect the desired simulated volume using the switches below. These simulations are based on an anatomical model of normal brain, which c

this pre-computed simulated brain database (SBD), the parameter settings are fixed to 3 modalities, 5 slice thicknesses, 6 levels of noise, mulations done with arbitrary parameters from the BrainWeb custom MRI simulations interface.

odality: (you can choose one of the following pulse sequences)

[✓ T1] T2 PD

ice thickness: (in-plane pixel size is always 1x1mm)

[✓ 1mm] 3mm 5mm 7mm 9mm

oise:

0% 1% [✓ 3%] 5% 7% 9%

ntensity non-uniformity ("RF"):

0% [✓ 20%] 40%

http://www.bic.mni.mcgill.ca/cgi/brainweb1

BrainWeb preview: T1 ICBM normal 1mm pn3 rf20

emember that the images you see here are completely synthetic! They are generated using a powerful MR simulator.

Modality=T1, Protocol=ICBM, Phantom_name=normal, Slice_thickness=1mm, Noise=3%, INU=20%

lick on an image to navigate, or use the buttons below

Left is left and right is right!

Motion Constraints: ☐ Horizontal Lock ☐ Vertical Lock
Stereotaxic coords: X (mm): 0.0 Y (mm): -18.0 Z (mm): 18.0
Color Scheme: Hotmetal Range: 0 % to 100 %
Zoom: [✓ 2X] off

BrainWeb download: T1 ICBM normal 1mm pn3 rf20

odality=T1, Protocol=ICBM, Phantom_name=normal, Slice_thickness=1mm, Noise=3%, INU=20%

INC volume info:

```
age: signed__ short 0 to 4095
age dimensions: zspace yspace xspace
   dimension name        length      step      start
   --------------        ------      ----      -----
   zspace                   181         1        -72
   yspace                   217         1       -126
   xspace                   181         1        -90
```

ginal MINC file size: 10.4 Mb

le format:
MINC
[✓ raw byte (unsigned)]
raw short (12 bit)
(about data formats)

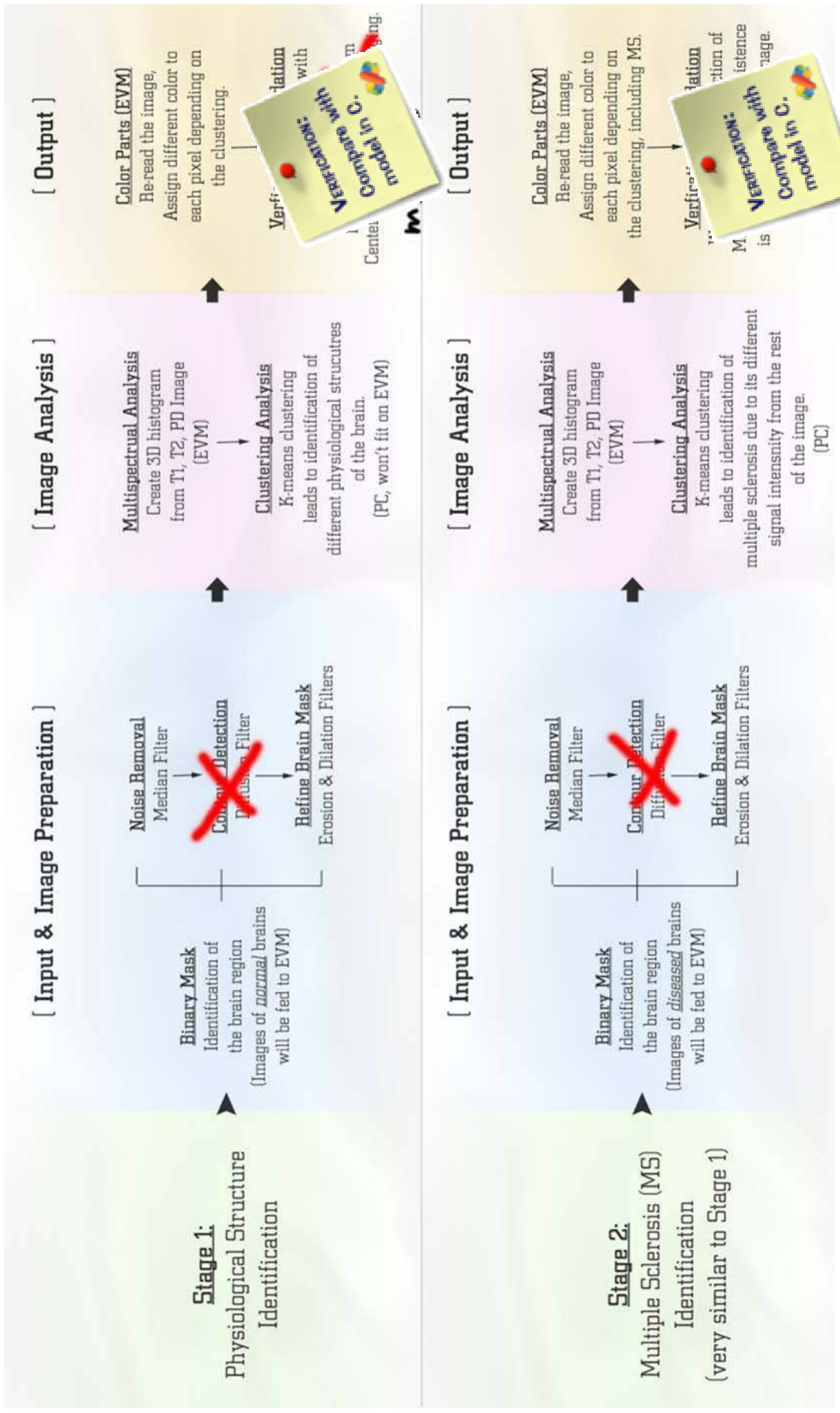
ompression:
none
[✓ gzip]
compress

'd like to know who is downloading these data, so we would appreciate it if you would provide us with the following information:

Name:	Yuchun Wang
Institution:	Carnegie Mellon University
E-mail:	yuchunw@andrew.cmu.edu

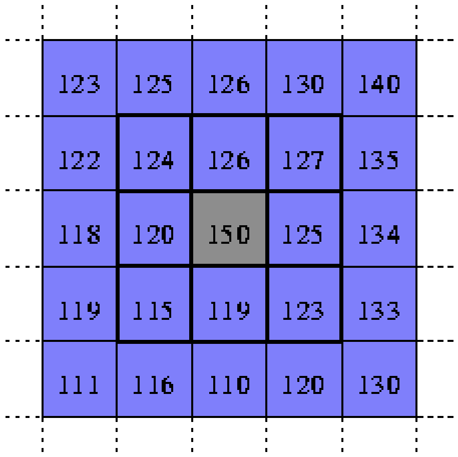
[Fig. 5.2.1]Screenshots taken from the BrainWeb website.

6. Algorithms


Stage 1: Physiological Structure Identification
[Input & Image Preparation]
Binary Mask
Identification of the brain region
(Images of normal brains will be fed to EVM)
Noise Removal
Median Filter
Contour Detection
Diffusion Filter
Refine Brain Mask
Erosion & Dilation Filters
[Image Analysis]
Multispectral Analysis
Create 3D histogram from T1, T2, PD Image (EVM)
Clustering Analysis
K-means clustering leads to identification of different physiological strucutres of the brain.
(PC, won't fit on EVM)
[Output]
Color Parts (EVM)
Re-read the image, Assign different color to each pixel depending on the clustering.
Verification: Compare with model in C.
Stage 2: Multiple Sclerosis (MS) Identification (very similar to Stage 1)
[Input & Image Preparation]
Binary Mask
Identification of the brain region
(Images of diseased brains will be fed to EVM)
Noise Removal
Median Filter
Contour Detection
Diffusion Filter
Refine Brain Mask
Erosion & Dilation Filters
[Image Analysis]
Multispectral Analysis
Create 3D histogram from T1, T2, PD Image (EVM)
Clustering Analysis
K-means clustering leads to identification of multiple sclerosis due to its different signal intensity from the rest of the image.
(PC)
[Output]
Color Parts (EVM)
Re-read the image, Assign different color to each pixel depending on the clustering, including MS.
Verification: Compare with model in C.

[Fig. 6.1.2] Data flow graph of our project.

6.1.1 Median Filter (5x5) on T1,T2,PD `do_comp3x5(), median_filter(), quicksort(args)`



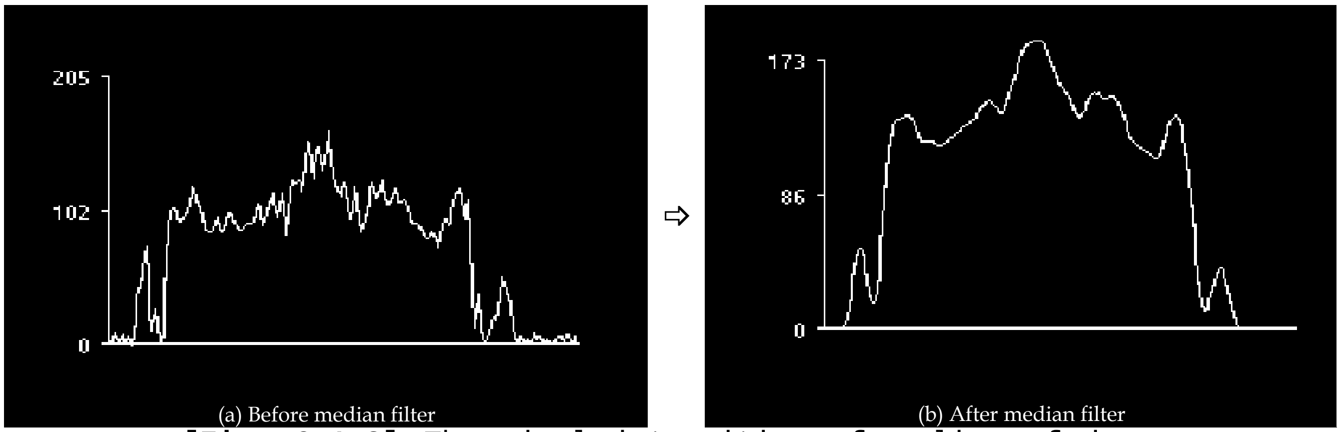
123	125	126	130	140
122	124	126	127	135
118	120	150	125	134
119	115	119	123	133
111	116	110	120	130

```
Neighbourhood values:
110,111,115,116,118,
119,119,120,120,122,
123,123,124,125,125,
126,126,127,130,130,
133,134,135,140,150


Median Value: 124
```

[Fig.6.1.1]

As illustrated in `Fig.6.1.1`, the median filter reads in 5x5 matrix at a time from the input image, quicksort the values in ascending order and replace the center pixel with the median value.

By doing this we 'smooth out' the values across the image and it's now easier to find the threshold to discard the skull from the image. We'll call this image the 'blurred image'.



(a) Before median filter (b) After median filter

[Fig. 6.1.2] The pixel intensities of a line of image before and after applying median filter.

6.1.2 Thresholding on PD (create mask to remove the skull) `extractBrain(args)`

Note: Here we are applying the thresholding algorithm on the PD image to create the mask for brain extraction. PD is specifically chosen because our training set shows that it produces the most accurate mask compared to T1 and T2.

(1) Extract one row from the blurred image. (`extractLine(args)`)
(2) Look for threshold from left: (`findT(args)`)
  a. Find the first peak (`max`) above a pre-determined threshold (`maxT=58`) between the left and the center of the row.
  b. Find the absolute minimum (`min`) within the next 20 pixels from `max`.
     The decision to limit the search within 20 pixels was through trial and error. We found that this would produce the most accurate detection of the skull.
  c. Zero-pad the original input image from the left up to `min`. This would mask out the skull along with the background in the left half of the row. (`zeroPad(args)`)
(3) Apply (2) from right. This would mask out the skull along with the background in the right half of the row.
(4) Do the same thing throughout the image (top to bottom).
(5) Extract one column from the image and do (2)-(4) vertically. (`extractLineV(args), zeroPadV(args)`)

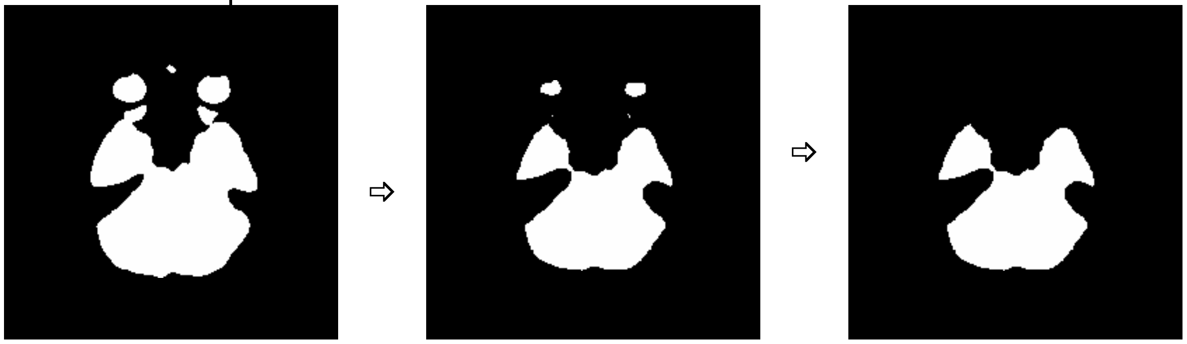
6.1.3 Erosion & Dilation (improve mask to remove remaining skull)

(1) Erosion: (`erode(args)`)

a. Apply 15x15 filter on the whole image: (`test_erosion(args)`)

```
0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,
0,0,0,0,0,0,1,1,1,0,0,0,0,0,0
0,0,0,0,0,1,1,1,1,1,0,0,0,0,0,
0,0,0,0,1,1,1,1,1,1,1,0,0,0,0,
0,0,0,1,1,1,1,1,1,1,1,1,0,0,0,
0,0,1,1,1,1,1,1,1,1,1,1,1,0,0,
0,1,1,1,1,1,1,1,1,1,1,1,1,1,0,
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
0,1,1,1,1,1,1,1,1,1,1,1,1,1,0,
0,0,1,1,1,1,1,1,1,1,1,1,1,0,0,
0,0,0,1,1,1,1,1,1,1,1,1,0,0,0,
0,0,0,0,1,1,1,1,1,1,1,0,0,0,0,
0,0,0,0,0,1,1,1,1,1,0,0,0,0,0,
0,0,0,0,0,0,1,1,1,0,0,0,0,0,0,
0,0,0,0,0,0,0,1,0,0,0,0,0,0,0
```

Example:


(a) Before erosion ⇨ (b) Eroding … ⇨ (c) After erosion

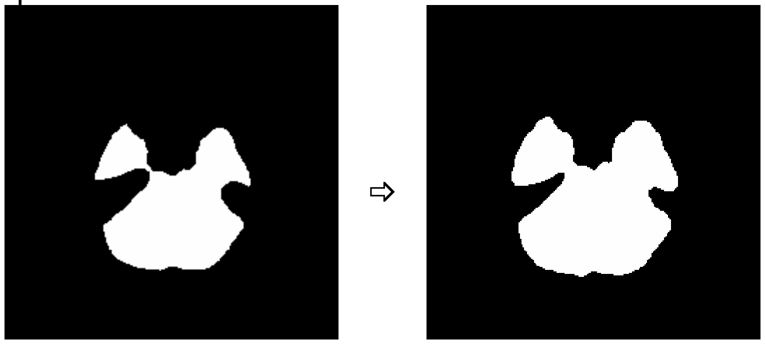
[Fig. 6.1.3.1]

(2) Dilation: (`dilate(args)`)

a. Apply 15x15 filter on the eroded image: (`test_dilation(args)`)

```
0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,
0,0,0,0,0,0,1,1,1,0,0,0,0,0,0,
0,0,0,0,0,1,1,1,1,1,0,0,0,0,0,
0,0,0,0,1,1,1,1,1,1,1,0,0,0,0,
0,0,0,1,1,1,1,1,1,1,1,1,0,0,0,
0,0,1,1,1,1,1,1,1,1,1,1,1,0,0,
0,1,1,1,1,1,1,1,1,1,1,1,1,1,0,
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
0,1,1,1,1,1,1,1,1,1,1,1,1,1,0,
0,0,1,1,1,1,1,1,1,1,1,1,1,0,0,
0,0,0,1,1,1,1,1,1,1,1,1,0,0,0,
0,0,0,0,1,1,1,1,1,1,1,0,0,0,0,
0,0,0,0,0,1,1,1,1,1,0,0,0,0,0,
0,0,0,0,0,0,1,1,1,0,0,0,0,0,0,
0,0,0,0,0,0,0,1,0,0,0,0,0,0,0
```

`Example:`



(a) Before dilation (b) After dilation

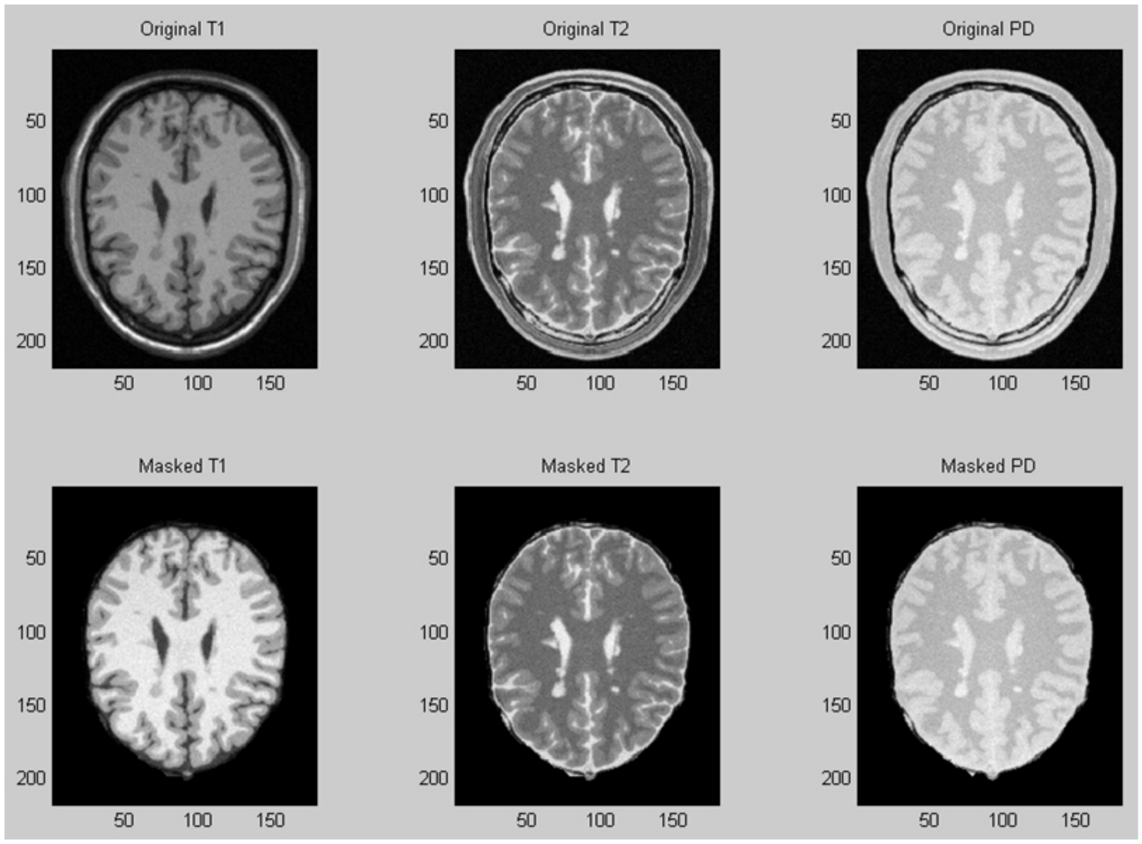
[Fig. 6.3.2]

As we can see from above, by eroding and dilating the image, we are able to remove parts of the image that are not essential to the segmentation of the brain. In particular, we are interested in removing the little bits and pieces of the skull that was not removed by the thresholding algorithm.

6.1.4 Applying Mask to T1 & T2 `createmask(args)`

Here we are applying the mask that was created from analyzing and processing the PD image with thresholding, erosion and dilation to T1 and T2 to mask out the skull for both images. After this process, our T1, T2, and PD images have their skulls removed and are ready for segmentation.

(1) Apply mask to T1 (`createmask(t1)`)
(2) Apply mask to T2 (`createmask(t2)`)
(3) Results:



[Fig.6.1.4.1]

6.2 Image Analysis `makeState(args), kmeans clustering`

The T1, T2, and PD-weighted images form the basis of our analysis. Since each image describes different physical properties of the particular brain slice, we can describe any pixel in the brain by its T1 intensity, T2 intensity, and PD intensity.

Ideally, pixels with identical or similar T1, T2, and PD intensities are the same material. Therefore, we plot each pixel in x-y-z space, where the x-axis represents a pixel's T1 intensity, the y-axis represents the pixel's T2 intensity, and the z-axis represents the pixel's PD intensity. Then, we find pixels that are similar to each other via the k-means (least mean distance) algorithm. This algorithm creates "clusters" of pixels with similar T1-T2-PD intensities, effectively identifying pixels in the brain that are made of the same material.

The code we used to perform k-means clustering was provided by Andrew Moore and Dan Pelleg [j]. The code is an implementation of x-means, a more dynamic version of k-means clustering. It actually decides how many clusters to create, but we found that the algorithm would create the maximum number of clusters we allowed. In our project, we allowed a maximum of 12 clusters. This gave us enough resolution to distinguish between all four materials (white matter, gray matter, cerebrospinal fluid, and MS).

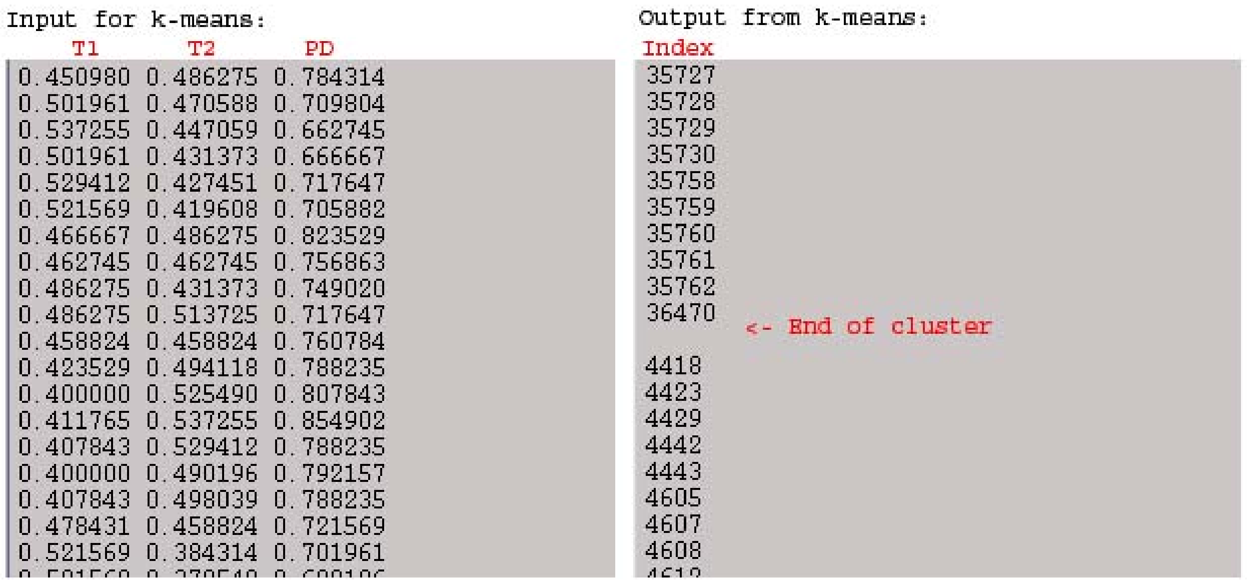
6.2.1 Space-state Matrix `makeState(args)`

The input to the k-means algorithm is a list of T1-T2-PD points. This is represented by a space-state matrix (sstate) with interlaced T1, T2, and PD values. The T1, T2, and PD-weighted images are read, masked, and stored into the matrix on the EVM-side. The function (makeState) is called to store the masked image into the space-state matrix. The 39277x3 matrix was written to a text file for clustering. The algorithm for makeState is:

- Let offset = 0 (for T1), 1 (for T2), or 2 (for PD)
- For each index j of the masked image array
- Place the value into coordinate (j, offset) of the space-state matrix

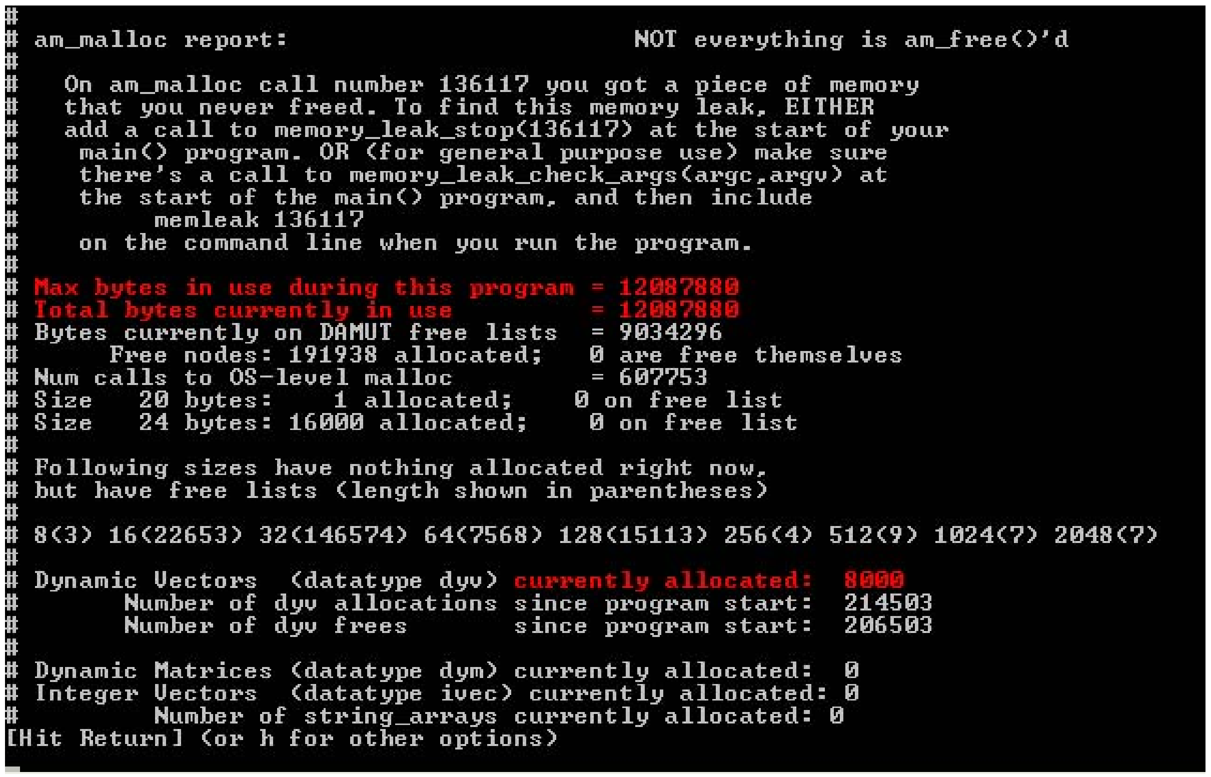
6.2.2 Kmeans Clustering

The clustering algorithm used a kd-tree to sort and cluster the points provided by the file containing the space-state matrix (sstate). This algorithm ran in an external executable and returned a file that contained the matrix indexes for each cluster. The output is then sent to the EVM. The motivation for running k-means separately is the performance: memory and CPU utilization are extreme. The running time for a three-dimensional 39277-point clustering is approximately 6 minutes on a SunBlade 100. A three-dimensional 39277-point graph is not available (Matlab froze on multiple occasions). For comparison, the memory utilization for two-dimensional 8000-point data is shown in Figure 6.2.2. The clustering illustration for the same data is shown in Figure 6.2.3.



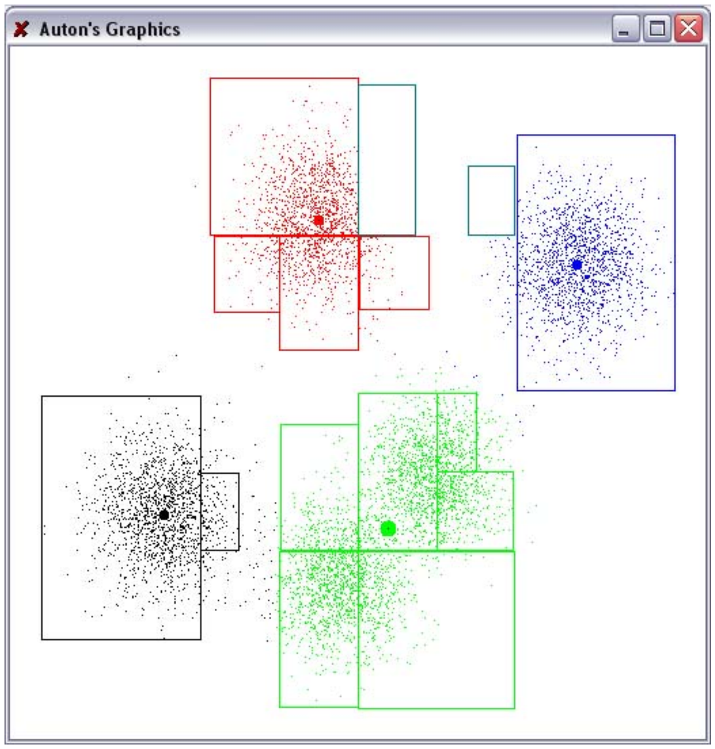
```
Input for k-means:                          Output from k-means:
    T1       T2       PD                    Index
0.450980 0.486275 0.784314                  35727
0.501961 0.470588 0.709804                  35728
0.537255 0.447059 0.662745                  35729
0.501961 0.431373 0.666667                  35730
0.529412 0.427451 0.717647                  35758
0.521569 0.419608 0.705882                  35759
0.466667 0.486275 0.823529                  35760
0.462745 0.462745 0.756863                  35761
0.486275 0.431373 0.749020                  35762
0.486275 0.513725 0.717647                  36470   <- End of cluster
0.458824 0.458824 0.760784
0.423529 0.494118 0.788235                  4418
0.400000 0.525490 0.807843                  4423
0.411765 0.537255 0.854902                  4429
0.407843 0.529412 0.788235                  4442
0.400000 0.490196 0.792157                  4443
0.407843 0.498039 0.788235                  4605
0.478431 0.458824 0.721569                  4607
0.521569 0.384314 0.701961                  4608
```

[Figure 6.2.1] The input is a three-dimensional list of intensities normalized from 0-255 to values between 0 and 1. The line number of each input data point represents the index of that pixel. The output is a list of indexes. These indexes are grouped by cluster, so we know which pixels are in the same cluster.



```
#
# am_malloc report:                      NOT everything is am_free()'d
#
#    On am_malloc call number 136117 you got a piece of memory
#    that you never freed. To find this memory leak, EITHER
#    add a call to memory_leak_stop(136117) at the start of your
#     main() program. OR (for general purpose use) make sure
#     there's a call to memory_leak_check_args(argc,argv) at
#     the start of the main() program, and then include
#          memleak 136117
#     on the command line when you run the program.
#
# Max bytes in use during this program = 12087880
# Total bytes currently in use         = 12087880
# Bytes currently on DAMUT free lists  = 9034296
#      Free nodes: 191938 allocated;   0 are free themselves
# Num calls to OS-level malloc         = 607753
# Size   20 bytes:     1 allocated;    0 on free list
# Size   24 bytes: 16000 allocated;    0 on free list
#
# Following sizes have nothing allocated right now,
# but have free lists (length shown in parentheses)
#
# 8(3) 16(22653) 32(146574) 64(7568) 128(15113) 256(4) 512(9) 1024(7) 2048(7)
#
# Dynamic Vectors  (datatype dyv) currently allocated:  8000
#       Number of dyv allocations since program start:  214503
#       Number of dyv frees       since program start:  206503
#
# Dynamic Matrices (datatype dym) currently allocated:  0
# Integer Vectors  (datatype ivec) currently allocated: 0
#          Number of string_arrays currently allocated: 0
[Hit Return] (or h for other options)
```

[Figure 6.2.2] The memory requirements for running a two-dimensional, 8000-point dataset is over 12 MB. By looking at the scaling of this kmeans implementation, we see that we cannot run a three-dimensional 39277-point dataset on the EVM.



[Figure 6.2.3] Screenshot of the *k*-means clustering.

6.3 Output `colorframe(), getColor(args)`

After kmeans has created its output file, we read the data line by line. If the character buffer has a number, we convert it to an int using atoi() and send it to the EVM. As we read the data, we also keep track of the ends of each cluster, denoted by a single endline character in the buffer. This list of cluster ends is also sent to the EVM.

On the EVM side, we receive the clustered data (color[]) and the cluster ends (clusterends[]) and create an RGB interlaced 39277x3 image.

- For index j from 0 to 39277
- Let pixel_idx = color[j] (the index of the pixel to color)
- Select color c from an array of colors
- Write color c into the interlaced frame in RGB format.
- If we see that pixel_idx is the last value in a cluster, increment color c

After we have created the colored image on the EVM, we send it to the PC to match each color to a material. We use a utility function getColor(…) to find a specific color. This function is passed the color c and the colored image from the EVM. It also gets a pointer to the output frame. The output frame is a binary image. The function getColor():

- For index j from 0 to 39277
- If coloredimage[j] is the color c, OR the output[j] is already 1, then write 1
- Else write 0.

The reason we have the OR condition will become apparent: The output frame represents our extracted materials. If there are 12 clusters and only 4 materials we are looking for, we will have several clusters describing a single material. Therefore, we want to perform a logical OR between the outputs of several clusters to get an image that represents a single material.

It is VERY important that the EVM and PC side have the same colors, otherwise we cannot match correctly. The colors we used:

```
          background      pinks       greens         yellow      blues      red
int colorR[12] = {0  , 255,255,196,   64 ,0  ,128,   255,    0  ,0  ,0  ,   255};
int colorG[12] = {0  , 192,128,128,   255,255,255,   255,    0  ,128,0  ,   0  };
int colorB[12] = {128, 255,255,255,   64 ,0  ,0  ,   0  ,    192,255,255,   0  };
```

6.3.1 Dynamic Assignment `calcAccuracy(args), finder(), getMaterials()`

After we have extracted a single cluster, we want to match it with a certain material. The BrainWeb database provides us with separate anatomical models of white matter, gray matter, CSF, and MS. We can superimpose these models on our extracted cluster and count the number of matching pixels (cluster[j] = model[j]) and the number of false positives (cluster[j] = 1, but model[j] = 0). A value of 1 means there is material present, and a 0 means there is none of the material present. These two statistics allow us to "score" a cluster against a specific material.

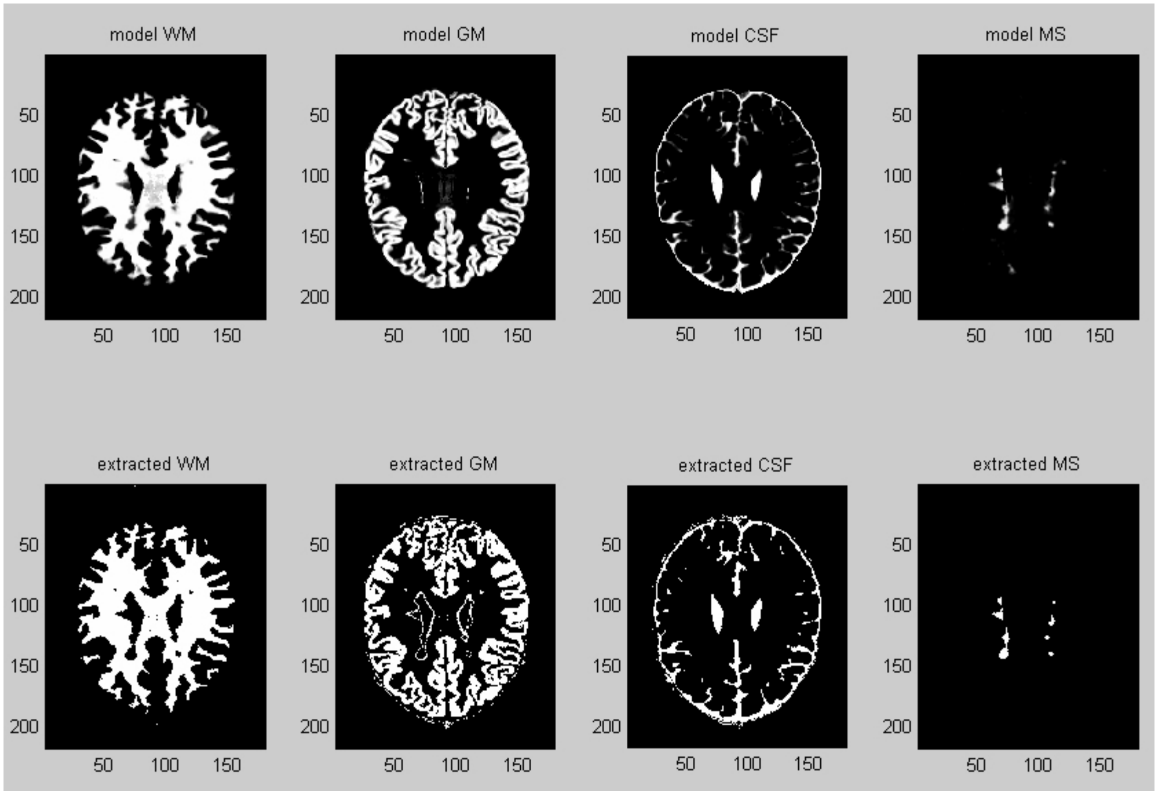
The `calcAccuracy(…)` function takes a binary image and a filename for a model (i.e. "c:\model_csf.raw") and a pointer to a struct containing fields for accuracy and false positives. It then counts the number of times the cluster "misses", meaning the value at input[j] = 0 when model[j] = 1. The accuracy statistic = 1 - (miss count / total pixels equal to 1 in the model). The false positive statistic = (false positive count / total pixels equal to 1 in the cluster).

The function `finder()` contains the algorithm that identifies clusters as a specific material. The function uses `calcAccuracy(…)` multiple times to find the accuracy and false positive statistics for some cluster[] versus the white matter model, cluster[] versus the gray matter model, cluster[] versus the CSF model, and in the case of MS, cluster[] versus the MS model. The material that has the highest accuracy and a false positive statistic below 40% is chosen. This cluster is then added to a list of clusters that represent the chosen material.

For detecting MS, we made a concession: since there were relatively few pixels of MS compared to the other materials, it was harder to segment using k-means. We allowed the false positive statistic to reach 98%, as long as the accuracy (hit rate) was high. This allowed us to find POSSIBLE areas with MS, albeit with a high rate of false positives.

6.3.2 Verification & Validation

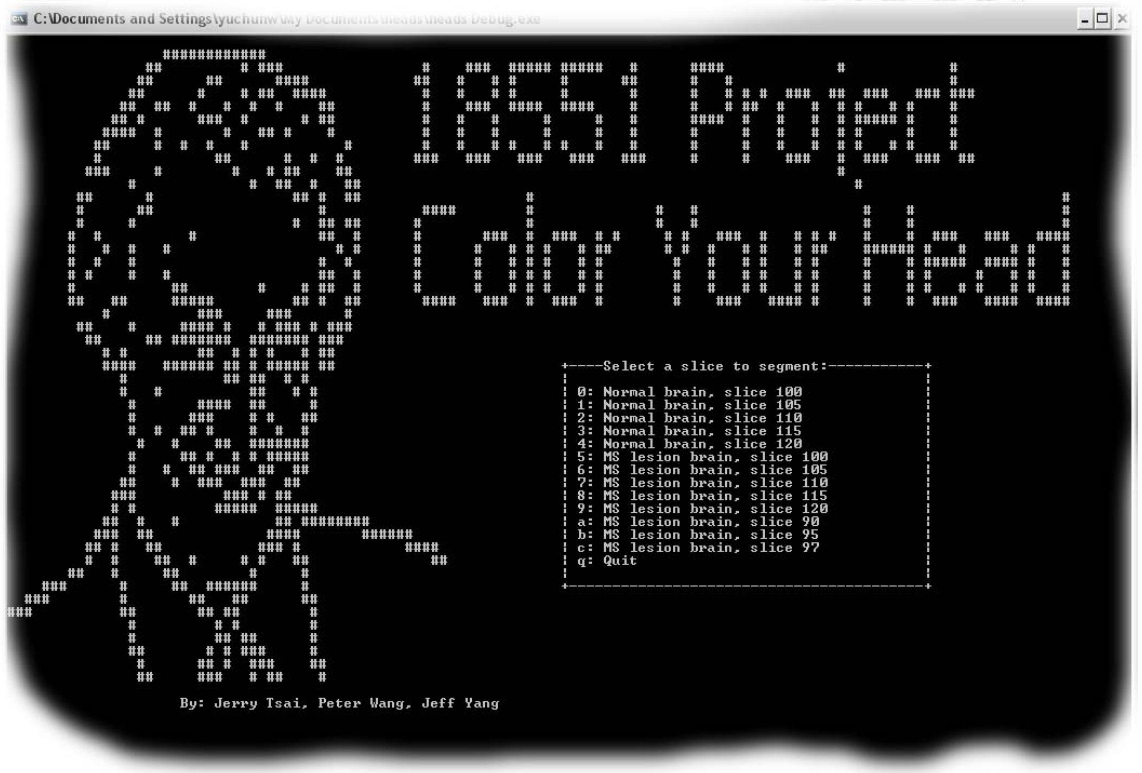
After we have identified which clusters represent each material, we want to output our results so we can display them in Matlab. We use the logical OR operation built into `calcAccuracy(…)` and extract all clusters that describe white matter. The resulting frame is written to a raw file and cumulative accuracy and false-positive statistics are displayed. This process is repeated for gray matter, CSF and MS. The results for slice 100 of an MS-diseased brain are seen in Figure 6.3.1:



[Fig 6.3.1] *Top row*: Models provided by BrainWeb.
Bottom row: Extracted clusters.

7. Demo / Results

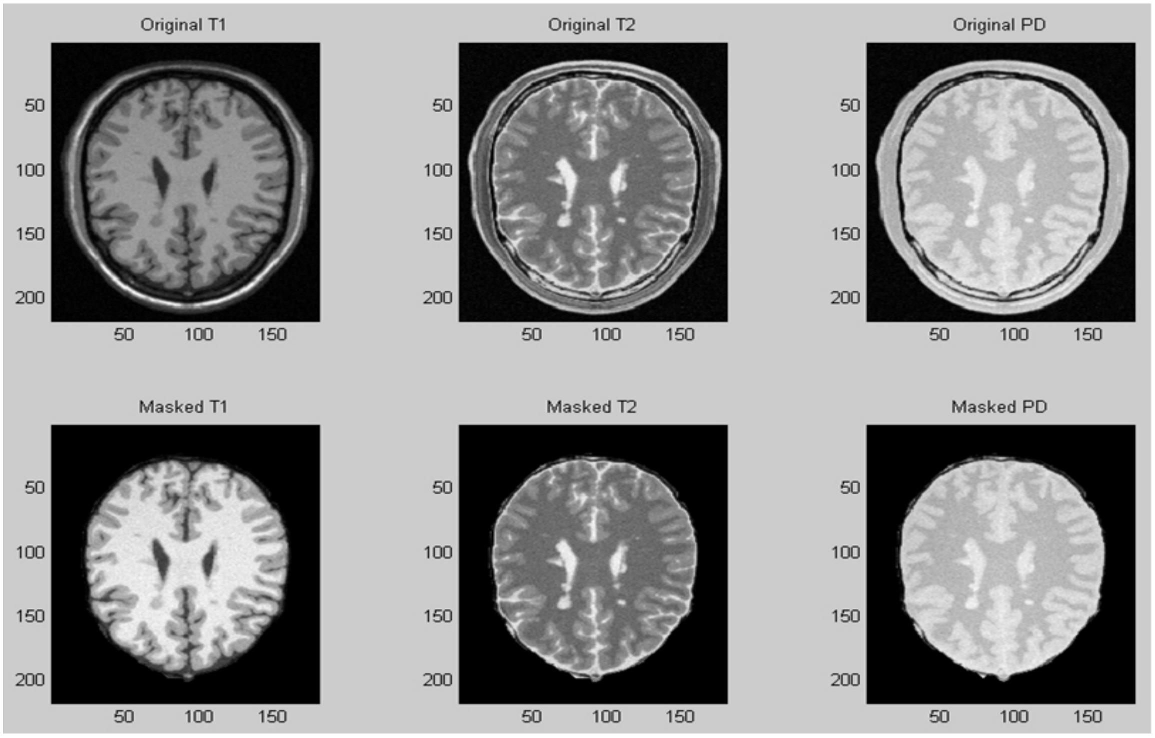
7.1 Demo Menu Screenshot



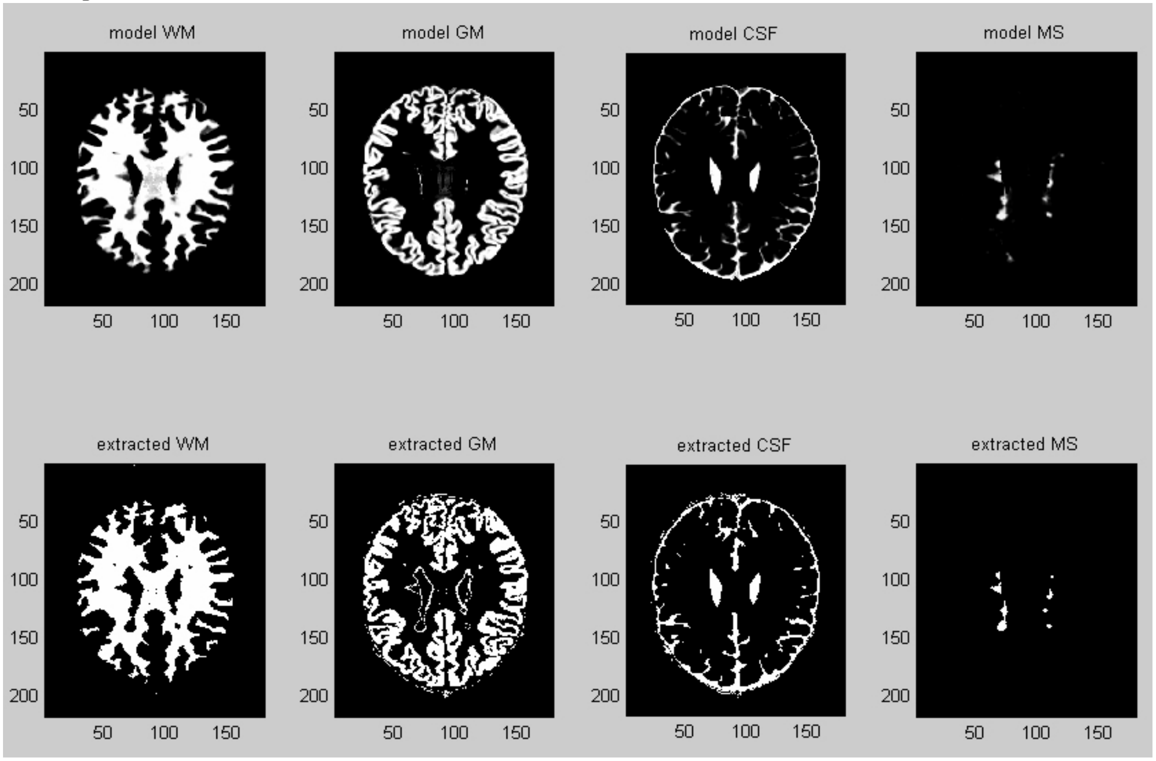
```
+----Select a slice to segment:------------+
|                                           |
| 0: Normal brain, slice 100                |
| 1: Normal brain, slice 105                |
| 2: Normal brain, slice 110                |
| 3: Normal brain, slice 115                |
| 4: Normal brain, slice 120                |
| 5: MS lesion brain, slice 100             |
| 6: MS lesion brain, slice 105             |
| 7: MS lesion brain, slice 110             |
| 8: MS lesion brain, slice 115             |
| 9: MS lesion brain, slice 120             |
| a: MS lesion brain, slice 90              |
| b: MS lesion brain, slice 95              |
| c: MS lesion brain, slice 97              |
| q: Quit                                   |
|                                           |
+-------------------------------------------+

By: Jerry Tsai, Peter Wang, Jeff Yang
```

[Fig 7.1] Screenshot of the program menu.

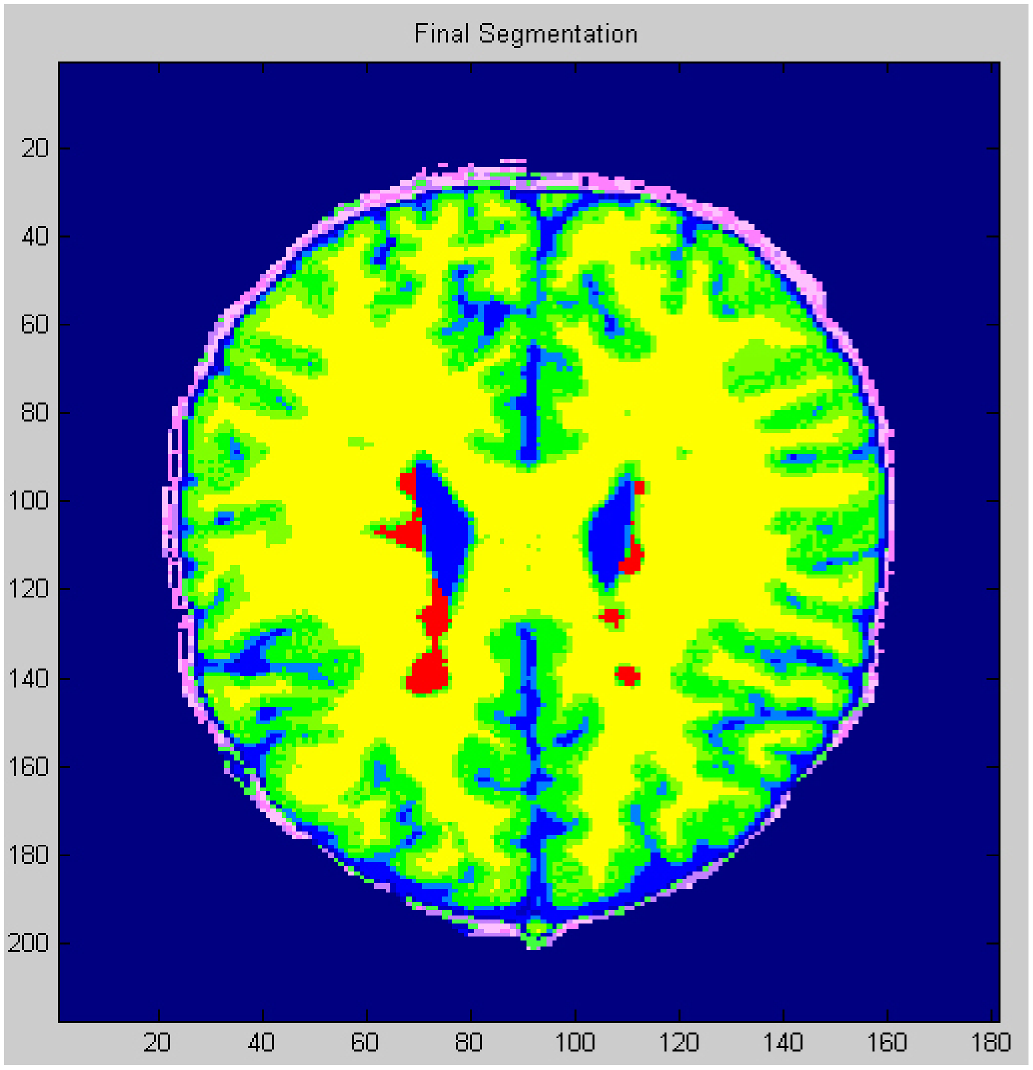
7.2 Demo Output



[Fig 7.2.1] Screenshot of the brain before and after extraction



[Fig 7.2.2] Screenshot of the comparison between anatomical models of the brain to the segmented brain by the program.



[Fig 7.2.3] Screenshot of the segmented brain by the program.

7.3 Accuracy and False Positives

For the Normal Brain test set:

Material	min – avg – max		min – max	
White matter:	84% - 92% - 97%	accuracy;	0% - 9%	false positives
Gray matter:	68% - 78% - 91%	accuracy;	3% - 12%	false positives
CSF:	62% - 88% - 98%	accuracy;	6% - 26%	false positives

For the MS-Diseased Brain test set:

Material	min – avg – max		min – max	
White matter:	84% - 92% - 97%	accuracy;	0% - 9%	false positives
Gray matter:	68% - 78% - 91%	accuracy;	3% - 12%	false positives
CSF:	62% - 88% - 98%	accuracy;	6% - 26%	false positives
MS:	0% - 22% - 65%	accuracy;	94%-98%	false positives

7.4 Possible Improvements

There are several areas we could improve in our project. The three main areas are code optimization, porting kmeans, and MS extraction.

Code & k-means:

- Currently, it takes about 5 seconds to create the mask (1. median filtering, 2. thresholding, 3. dilate/erode filtering). This could be lowered through memory access optimizations in all three operations.
- The median filter code uses I/O paging as in lab 3's `do_comp3()` function. This incurs the normal transfer penalty of 15 cycles per word. Our group attempted to use DMA transfers with paging, but was not successful. Using DMA incurs a 15 cycle first-transfer penalty, but only 1-cycle per word afterwards.
- The thresholding process does not implement any optimizations. We did not attempt to page memory or use DMA.
- The erosion and dilation filters do not implement any optimizations at all, using straight for-loops. However, paging may become nasty because of our large 15x15 kernel.
- The k-means code we used was memory intensive. Had we found another k-means implementation that used less memory, we could have ported it to the EVM.
- The code size could be reduced with parameterized functions both on the PC side and EVM side.

MS segmentation (extraction):

- The number of clusters can be increased to increase resolution in T1-T2-PD space (and thus extract the MS), but this would put more stress on the extraction of the other materials (more clusters per material).
- Alternatively, MS could be morphologically extracted by size or position after clustering (it usually appears as an island in white matter).

8. Performances

8.1 Memory Utilization

The total program on the EVM side took 131,742 bytes.
Total EVM malloc'd memory = 392,750 bytes:

- Frame = 39,277 bytes
- Result = Tempresult = 78,544 bytes
- Mask = 39,277 bytes
- Space-state = 235,632 bytes
- Clusterends = 20 bytes

K-means clustering requires 12 to 90 MB of RAM, too large for the EVM, so it was performed on PC.

8.2 Profile Results

Function	Cycles
create_mask	2,556,567
colorFrame	8,917,981
extracBrain	7,277,391
do_comp3x5	170.3 M
quickSort	830
findT	1043
zero_pad	778

9. Conclusion

We have shown that the segmentation of the brain and its component materials is reasonably efficient using a relatively simple process. Clustering is the main driver of segmentation after the brain has been extracted from an MRI scan. The preparation of the image is successfully performed by a single pass of a noise-reducing median filter, and a smart thresholding algorithm and morphological filters can refine a brain mask. For normal brains, segmentation is highly successful (near 90% average accuracy). However, the physical properties of MS are less suited to clustering and segmentation only averages around 22%. In the future, further post-clustering processing may extract MS more successfully using morphological filtering techniques.

10. References

[1] M. Stella Atkins and Blair T. Mackiewich, “Fully Automated Hybrid Segmentation of the Brain”
http://fas.sfu.ca/pub/cs/stella/book/word/Atkins3.doc

[2] Dan Pelleg and Andrew Moore, “Accelerating Exact *k*-means Algorithms with Geometric Reasoning”
http://www-2.cs.cmu.edu/~dpelleg/kmeans.html 2003

[3] QuickSort
www.sum-it.nl/QuickSort.java

[4] “MRIcro software Guide”
http://www.psychology.nottingham.ac.uk/staff/cr1/mricro.html

[5] “ISMRM: Magnetic Resonance Sites of the World Wide Web”,
http://www.ismrm.org/mr_sites.htm

[6] “MRIMAIN2”
http://www.cis.rit.edu/htbooks/mri/inside.htm 2002

[7] “MRI Segmentation”
http://www.fmrib.ox.ac.uk/~yongyue/mriseg.html 2001

[8] Yi Tao, William I. Grosky, Lucia Zamorano, Zhaowei Jiang, and Jianxing Gong, “Segmentation and representation of lesions in the MRI brain images”, www.cs.wayne.edu/~grosky/Papers/99SPIEMedical.pdf

[9] “MDL:MEM:Spring 98: Digital Signal Processing”,
http://www.cs.cmu.edu/~cil/v-images.html 2002

[10] “Imaging Science Research Division”
http://hsc.usf.edu/COM/radiology/isrd/Flash_Site/Research/Demos/demo3.html

[11] K. Krishnan and M. S. Atkins, “Segmentation of Multiple Sclerosis Lesions IN MRI-An Image Analysis Approach”
http://css.sfu.ca/sites/mcl/admin/user_resources/spie98.pdf

[12] “BrainWeb: Simulated Brain Database”
http://www.bic.mni.mcgill.ca/brainweb/ 2003

[13] “Center of Cognitive Brain Imaging”
http://coglab.psy.cmu.edu/

[14] “IBSR”
http://www.cma.mgh.harvard.edu/ibsr/ 2002

11. Program Code

11.1 filterpc.c

```
/****************************************************/
/* 18-551 Group 13 Project, Spring 2003              */
/* filterpc.c: Communication and file I/O routines   */
/*              for PC-side of "Color Your Head"     */
/* Authors: Jerry Tsai, Yuchun Wang, Jeffrey Yang     */
/* Based on code by <pwb@andrew.cmu.edu>             */
/****************************************************/

#include <stdio.h>
#include <windows.h>
#include "evm6xdll.h"

/* Image size */
#define X_SIZE 181
#define Y_SIZE 217
#define F_SIZE X_SIZE*Y_SIZE
#define HPIF_SIZE (F_SIZE+3)

#undef LOAD_FILE
#define LOAD_FILE    /* Uncomment to automatically load program */

// Filenames
#define INFILE "c:\\g13\\Project\\Images\\dataset\\t1_"
#define INFILE_T2 "c:\\g13\\Project\\Images\\dataset\\t2_"
#define INFILE_PD "c:\\g13\\Project\\Images\\dataset\\pd_"
#define INFILE_CLUSTER "c:\\g13\\Project\\Images\\dataset\\k551_"
//MS models
#define MODEL_MS "c:\\g13\\Project\\Images\\dataset\\model_ms_"
#define MODEL_CSF "c:\\g13\\Project\\Images\\dataset\\model_csf_"
#define MODEL_WM "c:\\g13\\Project\\Images\\dataset\\model_wm_"
#define MODEL_GM "c:\\g13\\Project\\Images\\dataset\\model_gm_"

//outputs
#define OUTFILE "c:\\g13\\Project\\Images\\result_t1.raw"
#define OUTFILE_T2 "c:\\g13\\Project\\Images\\result_t2.raw"
#define OUTFILE_PD "c:\\g13\\Project\\Images\\result_pd.raw"
#define OUTFILE_CLUSTER "c:\\g13\\Project\\Images\\result_final.raw"
```

11.1 filterpc.c

```
#define OUTFILE_ORIGINAL_T1 "c:\\g13\\Project\\Images\\original_t1.raw"
#define OUTFILE_ORIGINAL_T2 "c:\\g13\\Project\\Images\\original_t2.raw"
#define OUTFILE_ORIGINAL_PD "c:\\g13\\Project\\Images\\original_pd.raw"

//materials (individual extraction)
#define OUTFILE_CSF "c:\\g13\\Project\\Images\\extract_csf.raw"
#define OUTFILE_WM "c:\\g13\\Project\\Images\\extract_wm.raw"
#define OUTFILE_GM "c:\\g13\\Project\\Images\\extract_gm.raw"
#define OUTFILE_MS "c:\\g13\\Project\\Images\\extract_ms.raw"
#define OUTFILE_MODEL_CSF "c:\\g13\\Project\\Images\\model_csf.raw"
#define OUTFILE_MODEL_WM "c:\\g13\\Project\\Images\\model_wm.raw"
#define OUTFILE_MODEL_GM "c:\\g13\\Project\\Images\\model_gm.raw"
#define OUTFILE_MODEL_MS "c:\\g13\\Project\\Images\\model_ms.raw"
#define NORMALBRAIN 0
#define MSBRAIN 1
#define EVM_FILE "c:\\g13\\lab3\\echo.out"

/* Structure for holding transfer information */
struct transfer_s {
        void *buffer;
        unsigned long size;
        unsigned long command;
};
typedef struct {
        int clustercount;
        float accuracy;
        float falsepositive;
        char material;
}score;
/* Function prototypes */
#ifdef LOAD_FILE
int load_file(HANDLE hBd, LPVOID hHpi);
#endif
int wait_request(struct transfer_s *ts);
int send_data(struct transfer_s *ts, void *local_buf);
int get_data(struct transfer_s *ts, void *local_buf);
void hpi_write_word(ULONG addr, ULONG data);
void getMaterialsMS();
void getMaterialsNormal();
void getColor(short int *input, short int *output, int index);
void calcAccuracy(score *rval, short int *experiment, const char *modelFileName);
void finder(int braintype);
```

11.1 filterpc.c

```
/* Global vars */
unsigned char frame[F_SIZE];                    // Generic Input
short int result[F_SIZE];                       // Generic Output
short int lineresult[X_SIZE];                   // Line Output (debug)
short int kmeans_result[3*F_SIZE];              // Output for Kmeans
unsigned short int clusterframe[F_SIZE];        // Input for coloring
short int cluster_result[3*F_SIZE];             // Final output

char sinfile[50];
char sinfile_t2[50];
char sinfile_pd[50];
char sinfile_cluster[50];
char smodel_ms[50];
char smodel_csf[50];
char smodel_wm[50];
char smodel_gm[50];

int wm[5];
int gm[5];
int csf[5];
int ms[5];
int wmlen = 0, gmlen = 0, csflen = 0, mslen = 0;
score sorter[4];

HANDLE hBd  = NULL;
LPVOID hHpi = NULL;
HANDLE h_event;

// Write output for the T1-weighted images
void data_to_file(int number)
{
        FILE *outfile;
        char fname[255]
        sprintf(fname, "%s", OUTFILE);
        outfile=fopen(fname, "wb");
        fwrite(result, F_SIZE, sizeof(short int), outfile);
        fclose(outfile);
}
// Write output for the T2-weighted images
void data_to_file_t2(int number)
{
        FILE *outfile;
        char fname[255];
        sprintf(fname, "%s", OUTFILE_T2);
        outfile=fopen(fname, "wb");
        fwrite(result, F_SIZE, sizeof(short int), outfile);
        fclose(outfile);
}
// Write output for the PD-weighted images
void data_to_file_pd(int number)
{
        FILE *outfile;
        char fname[255];

        sprintf(fname, "%s", OUTFILE_PD);
        outfile=fopen(fname, "wb");
        fwrite(result, F_SIZE, sizeof(short int), outfile);
        fclose(outfile);
}
// DEBUG: Write a line of the output image
void data_to_file_line()
{
        FILE *outfile;
        char fname[255];
        int i;
        sprintf(fname, "c:\\g13\\Project\\Images\\lineresult.txt");
        outfile=fopen(fname, "wb");
        for(i = 0; i < X_SIZE; i++) {
                fprintf(outfile,"%i\n",lineresult[i]);
        }
        fclose(outfile);
}

// Write the input file for kmeans processing
void kmeans_to_file()
{
        FILE *outfile;
        char fname[255];
        int i,j;

        sprintf(fname, "c:\\g13\\Project\\Images\\kmeans.ds");
        outfile=fopen(fname, "wb");
        fprintf(outfile,"#num_rows = 39277\n");
        fprintf(outfile,"#num_cols = 3\n");
        fprintf(outfile,"#num_classes = 6\n");
//      fprintf(outfile,"#seed = 1244\n");
        fprintf(outfile,"#sigma = 0.055\n");
        fprintf(outfile,"\nx0 x1 x2\n");

        for(i = 0; i < F_SIZE; i++) {
                j = 3*i;
                fprintf(outfile,"%1.6f %1.6f %1.6f\n", kmeans_result[j]/255.,kmeans_result[j+1]/255.,kmeans_result[j+2]/255.);
        }
        fclose(outfile);
}

// Write the final clustered image
void data_to_file_cluster()
{
        FILE *outfile;
        char fname[255];

        sprintf(fname, OUTFILE_CLUSTER);
        outfile=fopen(fname, "wb");
        fwrite(cluster_result, 3*F_SIZE, sizeof(short int), outfile);
        fclose(outfile);
}

//outputs the different materials
void data_to_file_materials(const char* filename)
{
        FILE *outfile;
        char fname[255];

        sprintf(fname, "%s", filename);
        outfile=fopen(fname, "wb");
        fwrite(clusterframe, F_SIZE, sizeof(short int), outfile);
        fclose(outfile);
}

/* ==============================MAIN======================================*/
int main(int argc, char **argv) {
        int program_exit=0, i=0, j=0, k=0, slicenum, braintype;
        char s_buffer[80];
        unsigned int clusternum[1];
        unsigned short int *clusterends;
        unsigned char temp[1024];
        struct transfer_s ts;
        FILE *infile, *outfile;

printf("\n");
printf("              ############                     \n");
printf("            ##         # ###           #    ###  ##### #####   #      ####             #            #\n");
printf("           ##      ##      ####       ##   #   # #     #      ##      #   #                         #\n");
printf("          ##      #    # ##  ####      #   #   # #     #       #      #   # # #  ###  ##  ###   ### ###\n");
printf("         ##  ##  #   #  #  #    ##     #    ###  ####  ####    #      #   # ##  #   #  # #   # #    #\n");
printf("        ## #      ###  #      # ##     #   #   #     #     #   #      ####  #   #   #  # ##### #    #\n");
printf("       ####  #       #   ## #    #     #   #   #     #     #   #      #     #   #   #  # #     #    #\n");
printf("      ##     #  #  #   #           #   #   #   # #   # #   #   #      #     #   #   #  # #   # #    #\n");
printf("      #             ##      #  #  #   ###   ###   ###   ###   ###     #     #    ###   #  ###   ###  ##\n");
printf("     ###     #        #   # ##    ##                                                   #\n");
printf("          #             #  ##  #   ##                                                 # \n");
printf("    ##      #                ## #  ##              #                                                             #\n");
printf("    #      ##                   #      ####        #              #   #                   #    #                 #\n");
printf("    #     #                  #  ## ## #            #              #   #                   #    #                 #\n");
printf("   #  #          #              ##  # #       ###  #  ###  # #     # #   ###  #   # # #   #    #  ###   ###   ####\n");
printf("   #   #  #   #                   # # #      #   # # #   # ##      # #  #   # #   # ##    ###### #   #     # #   #\n");
printf("   #  #   #                     #  #  #      #   # # #   # #        #   #   # #   # #     #    # #####  #### #   #\n");
printf("   # #    #   #                 ##  # #      #   # # #   # #        #   #   # #   # #     #    # #     #   # #   #\n");
printf("   #          ##        #    # ##  # #      #   # # #   # #        #   #   # #   # #     #    # #   # #   # #   #\n");
printf("   ##   ##     #####         ## #  ##  ####   ###  #  ###  #        #    ###   #### #     #    #  ###   ####  ####\n");
printf("       #          ###     ###      #           \n");
printf("    ##    #     #### #   # ### # ###           \n");
printf("     ##     ## #######  ####### ###            \n");
printf("       # #        ##  # # #   # ##             \n");
printf("       ####   ###### ## # ##### ##             \t\t");printf("+----Select a slice to segment:-----------+\n");
printf("         #           ## ##  # #                \t\t");printf("|                                         |\n");
printf("         #   #          ##   # #               \t\t");printf("| 0: Normal brain, slice 100              |\n");
printf("          #       ####  ##     #               \t\t");printf("| 1: Normal brain, slice 105              |\n");
printf("          #      ###    # #   ##               \t\t");printf("| 2: Normal brain, slice 110              |\n");
printf("          #  #  ## #    #  #  #                \t\t");printf("| 3: Normal brain, slice 115              |\n");
printf("           #   #    ##  #######                \t\t");printf("| 4: Normal brain, slice 120              |\n");
printf("          #     ## #  # # #####                \t\t");printf("| 5: MS lesion brain, slice 100           |\n");
printf("          #   #  ## ###  ##  ##                \t\t");printf("| 6: MS lesion brain, slice 105           |\n");
printf("         ##    #  ###  ###  ##                 \t\t");printf("| 7: MS lesion brain, slice 110           |\n");
printf("        ###          ### # ##                  \t\t");printf("| 8: MS lesion brain, slice 115           |\n");
printf("        # #         #####  #####               \t\t");printf("| 9: MS lesion brain, slice 120           |\n");
printf("       ##  #   #           ## ########         \t\t");printf("| a: MS lesion brain, slice 90            |\n");
printf("      ###  ##             ####       ######    \t\t");printf("| b: MS lesion brain, slice 95            |\n");
printf("     ## #   ##           ### #            #### \t\t");printf("| c: MS lesion brain, slice 97            |\n");
printf("     # #     ##  #     # #   ##              ##\t\t");printf("| q: Quit                                 |\n");
printf("   ##   #     ##        #     #                \t\t");printf("|                                         |\n");
printf("###      #     ##  ######     #                \t\t");printf("+-----------------------------------------+\n");
printf("#        #       ##   ##      ##               \n");
printf("###      ##       ## ##        #               \n");
```

11.1 filterpc.c

```
printf("          #         # #        #               \n");
printf("          #         ## ##      #               \n");
printf("          ##       # # ###     #               \n");
printf("           #      ## #  ###    ##              \n");
printf("           ##     ###   # ##    #              \n");
printf("\n\t\t    By: Jerry Tsai, Peter Wang, Jeff Yang\n\n\n");

        getchar(); getchar();
        if(argc > 1) {
                clusternum[0] = atoi(argv[1]); // get the number of clusters from user
        } else {
                clusternum[0] = 12;                           // else default to 12 clusters
        }
        s_buffer[0] = '0';


                // Open the board
                #ifdef LOAD_FILE
                        hBd = evm6x_open(0, 1);
                #else
                        hBd = evm6x_open(0, 0);
                #endif

                if ( hBd == INVALID_HANDLE_VALUE ) {
                        fprintf(stderr, "Couldn't open board\n");
                        exit(1);
                }
                fprintf(stderr, "Opened connection to board...\n");

                // Also open connection to HPI
            hHpi = evm6x_hpi_open(hBd);
            if ( hHpi == NULL ) {
                        printf("NULL\n");
                        exit(4);
                }

                #ifdef LOAD_FILE
                        load_file(hBd, hHpi);
                        fprintf(stderr, "Loaded program...\n");
                //      while(1);
                #else
                        // Set DMA AUX priority greater than CPU priority, so we don't lock the PCI bus
                    hpi_write_word(0x01840070 /*Addr*/, 0x00000010 /*Data*/);
                // Reset the mailboxes and FIFO
                hpi_write_word(0x0170003C, 0x0e000000);
        #endif

        // Setup a windows event so we don't have to poll for incoming messages
        evm6x_clear_message_event(hBd);
        sprintf( s_buffer, "%s%d",EVM6X_GLOBAL_MESSAGE_EVENT_BASE_NAME, 0);
        h_event = OpenEvent( SYNCHRONIZE, FALSE, s_buffer );

        // get user input
        scanf("%c",temp);
        printf("=============================================\n");
        printf("You chose: %c\n", temp[0]);

        if(temp[0] >= '0' &&  temp[0] <= 'c') {

        switch(temp[0]) {
                case '0': slicenum = 100; braintype = 0; break;
                case '1': slicenum = 105; braintype = 0; break;
                case '2': slicenum = 110; braintype = 0; break;
                case '3': slicenum = 115; braintype = 0; break;
                case '4': slicenum = 120; braintype = 0; break;
                case '5': slicenum = 100; braintype = 1; break;
                case '6': slicenum = 105; braintype = 1; break;
                case '7': slicenum = 110; braintype = 1; break;
                case '8': slicenum = 115; braintype = 1; break;
                case '9': slicenum = 120; braintype = 1; break;
                case 'a': slicenum = 90; braintype = 1; break;
                case 'b': slicenum = 95; braintype = 1; break;
                case 'c': slicenum = 97; braintype = 1; break;
        }

        // append the slice and braintype to the input filenames
        if(braintype == NORMALBRAIN) {
                sprintf(sinfile,"%s%i.raw",INFILE,slicenum);
                sprintf(sinfile_t2,"%s%i.raw",INFILE_T2,slicenum);
                sprintf(sinfile_pd,"%s%i.raw",INFILE_PD,slicenum);
                sprintf(sinfile_cluster,"%s%i.final",INFILE_CLUSTER,slicenum);

        } else {
                sprintf(sinfile,"%sms_%i.raw",INFILE,slicenum);
                sprintf(sinfile_t2,"%sms_%i.raw",INFILE_T2,slicenum);
                sprintf(sinfile_pd,"%sms_%i.raw",INFILE_PD,slicenum);
                sprintf(sinfile_cluster,"%sms_%i.final",INFILE_CLUSTER,slicenum);
                sprintf(smodel_ms,"%s%i.raw",MODEL_MS,slicenum);
                infile=fopen(smodel_ms, "rb");
                fread(frame, F_SIZE, 1, infile);
                fclose(infile);
                outfile=fopen(OUTFILE_MODEL_MS, "wb");
                fwrite(frame, F_SIZE, sizeof(char), outfile);
                fclose(outfile);
        }

        // write out images the anatomical models
        sprintf(smodel_csf,"%s%i.raw",MODEL_CSF,slicenum);
        infile=fopen(smodel_csf, "rb");
        fread(frame, F_SIZE, 1, infile);
        fclose(infile);
        outfile=fopen(OUTFILE_MODEL_CSF, "wb");
        fwrite(frame, F_SIZE, sizeof(char), outfile);
        fclose(outfile);

        sprintf(smodel_wm,"%s%i.raw",MODEL_WM,slicenum);
        infile=fopen(smodel_wm, "rb");
        fread(frame, F_SIZE, 1, infile);
        fclose(infile);
        outfile=fopen(OUTFILE_MODEL_WM,"wb");
        fwrite(frame, F_SIZE, sizeof(char), outfile);
        fclose(outfile);

        sprintf(smodel_gm,"%s%i.raw",MODEL_GM,slicenum);
        infile=fopen(smodel_gm, "rb");
        fread(frame, F_SIZE, 1, infile);
        fclose(infile);
        outfile=fopen(OUTFILE_MODEL_GM,"wb");
        fwrite(frame, F_SIZE, sizeof(char), outfile);
        fclose(outfile);


        clusterends = (unsigned short int *)malloc(clusternum[0]*sizeof(short int));
        if(clusterends==NULL) {
                printf("color, couldn't allocate memory...\n");
                exit(1);
        }

        // Main loop... Waits for messages from the EVM
        // Performs a transfer depending on the value of the message received
        while(!program_exit) {
                fprintf(stderr, "...");
                //fprintf(stderr, "receving ... \n");
                wait_request(&ts);
                fprintf(stderr, "...");
                //fprintf(stderr, "Transfer request: CMD %x, SIZE %i, ADDRESS %x\n",
                                        ts.command, ts.size, ts.buffer);

                // Now based on the command that was sent, do something
                switch(ts.command) {
                case(0x01): // Send frame
                        infile=fopen(sinfile, "rb");
                        fread(frame, F_SIZE, 1, infile);
                        fclose(infile);
                        outfile=fopen(OUTFILE_ORIGINAL_T1, "wb");
                        fwrite(frame, F_SIZE, sizeof(char), outfile);
                        fclose(outfile);
                        if(ts.size != HPIF_SIZE) fprintf(stderr, "Wrong size!!!\n");
                        send_data(&ts, frame);
                        break;
                case(0x02): // Get frame
                        if(ts.size != HPIF_SIZE*sizeof(short int))
                                fprintf(stderr, "Wrong size\n");
                        get_data(&ts, result);
                        data_to_file(++i);
                        break;
                case(0x03): // Print string
                        get_data(&ts, temp);
                        printf("...");
                        //printf("%s\n",temp );
                        break;
                case(0x04): // Exit program
                        program_exit = 1;
                        fprintf(stderr, "\n\nSegmentation done!\n\n");
                        break;
                case(0x05):  //Get Kmeans data (EVM -> kmeans)
                        if(ts.size != 3*HPIF_SIZE*sizeof(short int))
                                fprintf(stderr, "Wrong size\n");
                        get_data(&ts, kmeans_result);
                        kmeans_to_file();
                        break;
                case(0x06): // Send T2 frame
                        infile=fopen(sinfile_t2, "rb");
                        fread(frame, F_SIZE, 1, infile);
                fclose(infile);
                outfile=fopen(OUTFILE_ORIGINAL_T2, "wb");
                fwrite(frame, F_SIZE, sizeof(char), outfile);
                fclose(outfile);
                if(ts.size != HPIF_SIZE) fprintf(stderr, "Wrong size!!!\n");
                send_data(&ts, frame);
                break;
        case(0x07): // Get T2 frame
                if(ts.size != HPIF_SIZE*sizeof(short int))
                        fprintf(stderr, "Wrong size\n");
                get_data(&ts, result);
                data_to_file_t2(++i);  // Output frame to a new file
                break;
        case(0x08): // Send PD frame
                infile=fopen(sinfile_pd, "rb");
                fread(frame, F_SIZE, 1, infile);
                fclose(infile);
                outfile=fopen(OUTFILE_ORIGINAL_PD, "wb");
                fwrite(frame, F_SIZE, sizeof(char), outfile);
                fclose(outfile);
                if(ts.size != HPIF_SIZE) fprintf(stderr, "Wrong size!!!\n");
                send_data(&ts, frame);
                break;
        case(0x09): // Get PD frame
                if(ts.size != HPIF_SIZE*sizeof(short int))
                        fprintf(stderr, "Wrong size\n");
                get_data(&ts, result);
                data_to_file_pd(++i);  // Output frame to a new file
                break;
        case(0x0A): // DEBUG: get a line of the frame
                get_data(&ts, lineresult);
                data_to_file_line();
                break;
        case(0x0B): // Send # of clusters
                if(ts.size != sizeof(int)) fprintf(stderr, "Wrong size!!!\n");
                send_data(&ts, clusternum);
                break;
        case(0x0C): // Send cluster lines (kmeans -> EVM)

                // Wait for kmeans to finish, wait for user's signal:
                //printf("\n\nWaiting for kmeans output... [Press Enter when ready]\n");
                //getchar();getchar();

                infile=fopen(sinfile_cluster, "r");
```

```
                                // Discard four header lines from kmeans output
                                fgets(temp,1024,infile);
                                fgets(temp,1024,infile);
                                fgets(temp,1024,infile);
                                fgets(temp,1024,infile);
                                // Read kmeans output to clusterframe then send clusterframe to EVM
                                j = k = 0;
                                while(fgets(temp,1024,infile) != NULL) {
                                        if(strlen(temp) == 1) {// found end of a cluster (temp = "\n")
                                                clusterends[k++] = clusterframe[j-1];
                                        } else { // append temp to clusterframe
                                                clusterframe[j++] = (unsigned short int)atoi(temp);
                                        }
                                }
                                fclose(infile);
                                if(ts.size != HPIF_SIZE*sizeof(short int)) fprintf(stderr, "Wrong size!!!\n");
                                send_data(&ts, clusterframe);
                                break;
                        case(0x0D): // Send cluster end values
                                if(ts.size != clusternum[0]*sizeof(int)) fprintf(stderr, "Wrong size!!!\n");
                                send_data(&ts, clusterends);
                                break;
                        case(0x0E): // Get final colored image (EVM -> PC)
                                if(ts.size != 3*HPIF_SIZE*sizeof(short int))
                                        fprintf(stderr, "Wrong size\n");
                                get_data(&ts, cluster_result);
                                data_to_file_cluster();
                                break;
                        default:
                                fprintf(stderr, "Unknown command\n");
                                break;
                        }
                }

                //match and analyze the materials
                if(braintype == MSBRAIN)
                        getMaterialsMS();
                else
                        getMaterialsNormal();

                //wait for user
                getchar();getchar();
        } // end valid command (0-c)
        // Clean up and exit
        if (!evm6x_hpi_close(hHpi)) exit(9);
        if (!evm6x_close(hBd)) exit(16);
        return(0);
}

/* Waits for windows event signalling incoming message.  Then reads
        address from mailbox 1, size from mailbox 2, and command from
        mailbox 3 */
int wait_request(struct transfer_s *ts)
{
        // wait for event signaling a message from the DSP
        WaitForSingleObject( h_event, INFINITE );

        if(!evm6x_retrieve_message(hBd, (unsigned long *)&ts->buffer))
                fprintf(stderr, "Error retrieving 1...\n");
        if(!evm6x_mailbox_read(hBd, 2, (unsigned long *)&ts->size))
                fprintf(stderr, "Error retrieving 2...\n");
        if(!evm6x_mailbox_read(hBd, 3, (unsigned long *)&ts->command))
                fprintf(stderr, "Error retrieving 3...\n");

        return(0);
}

/* Size and address are given in struct ts.  local_buf is the address of
        the PC buffer to send */
int send_data(struct transfer_s *ts, void *local_buf)
{
        unsigned long int ulLength = ts->size;

        /* Write the data to EVM memory*/
        if (!evm6x_hpi_write(hHpi, (PULONG)local_buf, (PULONG)&ulLength, (ULONG)ts->buffer)) {
                fprintf(stderr, "HPI write error\n");
                exit(1);
        }
        if (ulLength != ts->size) {
                fprintf(stderr, "Send: HPI only wrote %i bytes\n", ulLength);
                fprintf(stderr, "%i, %i\n", ulLength, ts->size);
                exit(1);
        }

        /* Use a message to signal the EVM that the transfer is done */
        if (!evm6x_send_message(hBd, (PULONG)&ts->command)) {
                fprintf(stderr, "Send message error!\n");
                exit(1);
        }
        return(0);
}

/* Same format as send_data */
int get_data(struct transfer_s *ts, void *local_buf)
{
        unsigned long int ulLength = ts->size;

        /* Read data from EVM memory */
    if (!evm6x_hpi_read(hHpi, (PULONG)local_buf, (PULONG)&ulLength, (ULONG)ts->buffer)) {
                fprintf(stderr, "HPI write error\n");
                exit(1);
        }
        if (ulLength != ts->size) {
                fprintf(stderr, "Get: HPI only wrote %i bytes\n", ulLength);
                fprintf(stderr, "%i, %i\n", ulLength, ts->size);
                exit(1);
        }

        /* Signal EVM that transfer is done */
        if (!evm6x_send_message(hBd, (PULONG)&ts->command)) {
                fprintf(stderr, "Send message error!\n");
                exit(1);
        }
        return(0);
}

/* Write a single 32-bit word to EVM memory */
void hpi_write_word(ULONG addr, ULONG data)
{
        ULONG ulLength = 4;

        if (!evm6x_hpi_write(hHpi, &data, &ulLength, addr)) {
                fprintf(stderr, "Error writing word via HPI\n");
                exit(1);
        }
        if (ulLength != 4) {
                fprintf(stderr, "Error writing word via HPI\n");
                exit(1);
        }
}
```

11.1 filterpc.c

```
#ifdef LOAD_FILE
/* Load the .out file and run the program.  Code Composer must not be running */
int load_file(HANDLE hBd, LPVOID hHpi)
{
  if ( !evm6x_reset_board(hBd) ) exit(2);
  if ( !evm6x_reset_dsp(hBd,HPI_BOOT) ) exit(3);
  if ( !evm6x_init_emif(hBd, hHpi) ) exit(5);

  /* Load program */
  if (!evm6x_coff_load(hBd,hHpi,EVM_FILE,0,0,0)) exit(8);

  /* Set HPI priority and reset mailboxes */
  hpi_write_word(0x01840070 /*Addr*/, 0x00000010 /*Data*/ );
  hpi_write_word(0x0170003C, 0x0e000000);

  if (!evm6x_unreset_dsp(hBd)) exit(10);
  if (!evm6x_set_timeout(hBd,1000)) exit(11);
        return(0);
}
#endif

// getColor(...)
// extracts specific colors from the image returned after clustering
// --------------
// *input: input frame, RGB interlaced, 3x181x217
// *output: output frame, binary mask, 1x181x217
// index: index of color to find (i.e. index=7 means "find yellow")
void getColor(short int *input, short int *output, int index) {
                       // background          pinks          greens                 yellow  blues                  red
        int colorR[12] = {0  , 255,255,196,   64 ,0  ,128,   255,    0  ,0  ,0  ,    255};
        int colorG[12] = {0  , 192,128,128,   255,255,255,   255,    0  ,128,0  ,    0  };
        int colorB[12] = {128, 255,255,255,   64 ,0  ,0  ,   0  ,    192,255,255,    0  };
        int j=0;

        for(j=0; j<F_SIZE; j++) {
                if((input[3*j]==colorR[index] && input[3*j+1]==colorG[index] && input[3*j+2]==colorB[index]) || output[j]==255)
                        output[j]=255;
                else
                        output[j]=0;
        }

}

// calcAccuracy(...)
// compares a binary frame to an anatomical model and returns the accuracy and # of false positives
// --------------
// *rval: a "score" struct containing fields for return values
// *experiment: a B&W frame (generated using getColor) to be compared with a model
// *modelFileName: the file that contains the model (i.e. white matter, gray matter, CSF, MS)
void calcAccuracy(score *rval, short int *experiment, const char *modelFileName) {
        int i;
        int count=0, falsePositive = 0, modelSum=0, clustercount=0;
        FILE *infile;

        // read in the model into frame
        infile=fopen(modelFileName, "rb");
        fread(frame, F_SIZE, 1, infile);
        fclose(infile);

        //compare the model with the experimental output
        for(i=0; i<F_SIZE; i++) {
                if(frame[i]>64) {  // 64 is the min. intensity before 'overlap' occurs in the models
                        modelSum++;
                        if(experiment[i]==0)
                                count++;
                }
                if(experiment[i]!=0) {
                        clustercount++;
                        if(frame[i]<=64)
                                falsePositive++;
                }
        }

        // the number of pixels we designated as "[material]"
        rval->clustercount = clustercount;

        // accuracy: the percentage of pixels we found that are in the model.
        rval->accuracy = 100-(count/(float)modelSum)*100;

        // falsepositive: the percentage of pixels we found that are NOT in the model
        rval->falsepositive = falsePositive/(float)clustercount;
}


// finder(...)
// Loops through clusters 3 through 12, matching the pixels in that cluster to a
// specific material (WM, GM, CSF, or MS).  It computes the accuracy and # of
// false positives of the cluster against each of the materials, then decides
// what model best describes this cluster based on accuracy. It will disregard
// the cluster if there are too many false positives when compared to the model.
// --------------
// braintype: specifies whether to find MS (multiple sclerosis)
void finder(int braintype) {
        int tclustercount;
        float taccuracy;
        float tfalsepositive;
        char tmaterial;
        float minfpval = 1.00, maxacval = 0.00;
        int minfp = 3, maxac = 3;

        double fpthreshold = .40;      // set false positive threshold to 40%
        int i, j, k;

        for(i = 3; i < 12; i++) {

                // clear out the frame
                for(j=0; j<F_SIZE; j++)
                        clusterframe[j]=0;

                // get the cluster (based on color)
                getColor(cluster_result, clusterframe, i);
                // get the stats for this cluster vs. white matter
                calcAccuracy(&sorter[0], clusterframe, smodel_wm);
                sorter[0].material = 'w';

                for(j=0; j<F_SIZE; j++)
                        clusterframe[j]=0;

                getColor(cluster_result, clusterframe, i);
                // get the stats for this cluster vs. gray matter
                calcAccuracy(&sorter[1], clusterframe, smodel_gm);
                sorter[1].material = 'g';

                for(j=0; j<F_SIZE; j++)
                        clusterframe[j]=0;

                getColor(cluster_result, clusterframe, i);
                // get the stats for this cluster vs. CSF
                calcAccuracy(&sorter[2], clusterframe, smodel_csf);
                sorter[2].material = 'c';

                if(braintype==MSBRAIN) {
```

```
                for(j=0; j<F_SIZE; j++)
                        clusterframe[j]=0;

                getColor(cluster_result, clusterframe, i);
                // get the stats for this cluster vs. MS (multiple sclerosis)
                calcAccuracy(&sorter[3], clusterframe, smodel_ms);
                sorter[3].material = 'm';

                // keep track of which cluster BEST describes MS:
                minfpval = (minfpval < sorter[3].falsepositive)?minfpval:sorter[3].falsepositive;
                maxacval = (maxacval > sorter[3].accuracy)?maxacval:sorter[3].accuracy;
                minfp = (minfpval < sorter[3].falsepositive)?minfp:i;
                maxac = (maxacval > sorter[3].accuracy)?maxac:i;
        } else {
                sorter[3].accuracy = 0.00;
                sorter[3].falsepositive = 1.00;
        }

        // sort the list according to accuracy, from min to max
        for(j = 0; j < 3; j++) {
                for(k = 0;k < 3-j; k++) {
                        if(sorter[k].accuracy > sorter[k+1].accuracy) {
                                tclustercount = sorter[k].clustercount;
                                taccuracy = sorter[k].accuracy;
                                tfalsepositive = sorter[k].falsepositive;
                                tmaterial = sorter[k].material;

                                sorter[k].accuracy = sorter[k+1].accuracy;
                                sorter[k].clustercount = sorter[k+1].clustercount;
                                sorter[k].falsepositive = sorter[k+1].falsepositive;
                                sorter[k].material = sorter[k+1].material;

                                sorter[k+1].accuracy = taccuracy;
                                sorter[k+1].clustercount = tclustercount;
                                sorter[k+1].falsepositive = tfalsepositive;
                                sorter[k+1].material = tmaterial;
                        }
                }
        }

        // go through the sorted list from max to min
        // match this cluster with the material with highest accuracy,
        // as long as it's false positives are higher than 40% of the total
        for(j = 3; j > -1; j--) {
```

```
                        if(sorter[j].falsepositive < fpthreshold) {
                                switch(sorter[j].material) {
                                        case 'w': wm[wmlen++] = i;break;
                                        case 'g': gm[gmlen++] = i;break;
                                        case 'c': csf[csflen++] = i;break;
                                        case 'm': ms[mslen++] = i; break;
                                }
                                break;
                        }
                }
        } // end loop over all clusters

        // once we assign all clusters, we want to go back and see if there were any
        // MS clusters that had high false positive counts:
        if(mslen == 0 && maxacval > 26.0 && minfpval < 0.98) {
                for(i = 0; i < 5; i++) {
                        if(wm[i] == minfp) { wm[i] = wm[--wmlen]; ms[mslen++] = minfp; break;}
                        if(gm[i] == minfp) { gm[i] = gm[--gmlen]; ms[mslen++] = minfp; break;}
                        if(csf[i] == minfp) { csf[i] = csf[--csflen]; ms[mslen++] = minfp; break;}
                }
        }
}

// getMaterialsMS()
// calls finder to match clusters with materials, (WM, GM, CSF, and MS)
// then writes out images of each extracted material, and prints stats
void getMaterialsMS() {
        int i;
        score temp;

        finder(MSBRAIN);

        for(i=0; i<F_SIZE; i++)
                clusterframe[i]=0;

        printf("=======================Clustering Accuracy=======================\n");

        for(i = 0; i < wmlen; i++) { // compile all white matter clusters into a single frame
                getColor(cluster_result, clusterframe, wm[i]);
        }
        data_to_file_materials(OUTFILE_WM);                           // write out the frame
        calcAccuracy(&temp, clusterframe, smodel_wm); // compute statistics vs. GM model
        printf("%s accuracy: %3.2f%%\n", smodel_wm, temp.accuracy);
        printf("%s false positives: %3.2f\n\n",smodel_wm,temp.falsepositive);
        for(i=0; i<F_SIZE; i++)
                clusterframe[i]=0;
        for(i = 0; i < gmlen; i++) { // compile all gray matter clusters into a single frame
                getColor(cluster_result, clusterframe, gm[i]);
        }
        data_to_file_materials(OUTFILE_GM);                            // write out the frame
        calcAccuracy(&temp, clusterframe, smodel_gm); // compute statistics vs. GM model
        printf("%s accuracy: %3.2f%%\n", smodel_gm, temp.accuracy);
        printf("%s false positives: %3.2f\n\n",smodel_gm,temp.falsepositive);

        for(i=0; i<F_SIZE; i++)
                clusterframe[i]=0;

        for(i = 0; i < csflen; i++) { // compile all CSF clusters into a single frame
                getColor(cluster_result, clusterframe, csf[i]);
        }
        data_to_file_materials(OUTFILE_CSF);                  // write out the frame
        calcAccuracy(&temp, clusterframe, smodel_csf);        // compute statistics vs. GM model
        printf("%s accuracy: %3.2f%%\n", smodel_csf, temp.accuracy);
        printf("%s false positives: %3.2f\n\n",smodel_csf,temp.falsepositive);

        for(i=0; i<F_SIZE; i++)
                clusterframe[i]=0;
        for(i = 0; i < mslen; i++) { // compile all MS clusters into a single frame
                getColor(cluster_result, clusterframe, ms[i]);
        }
        data_to_file_materials(OUTFILE_MS);                   // write out the frame
        if(mslen > 0) {
                calcAccuracy(&temp, clusterframe, smodel_ms); // compute statistics vs. GM model
        }
        else {
                // sometimes we will not find MS (mslen = 0)
                temp.accuracy = 0.0;
                temp.falsepositive = 0.0;
        }
        printf("%s accuracy: %3.2f%%\n", smodel_ms, temp.accuracy);
        printf("%s false positives: %3.2f\n\n",smodel_ms,temp.falsepositive);

        wmlen = gmlen = csflen = mslen = 0;
}

// getMaterialsNormal()
// calls finder to match clusters with materials, (WM,GM,and CSF)
// then writes out images of each extracted material, and prints stats
void getMaterialsNormal() {
        int i;
        score temp;

        finder(NORMALBRAIN);

        printf("========================Clustering Accuracy========================\n");

        for(i = 0; i < wmlen; i++) { // compile all white matter clusters into a single frame
                getColor(cluster_result, clusterframe, wm[i]);
        }
        data_to_file_materials(OUTFILE_WM);                           // write out the frame
        calcAccuracy(&temp, clusterframe, smodel_wm); // compute statistics vs. WM model
        printf("%s accuracy: %3.2f%%\n", smodel_wm, temp.accuracy);
        printf("%s false positives: %3.2f\n\n",smodel_wm,temp.falsepositive);

        for(i=0; i<F_SIZE; i++)
                clusterframe[i]=0;

        for(i = 0; i < gmlen; i++) { // compile all gray matter clusters into a single frame
                getColor(cluster_result, clusterframe, gm[i]);
        }
        data_to_file_materials(OUTFILE_GM);                           // write out the frame
        calcAccuracy(&temp, clusterframe, smodel_gm); // compute statistics vs. GM model
        printf("%s accuracy: %3.2f%%\n", smodel_gm, temp.accuracy);
        printf("%s false positives: %3.2f\n\n",smodel_gm,temp.falsepositive);

        for(i=0; i<F_SIZE; i++)
                clusterframe[i]=0;

        for(i = 0; i < csflen; i++) { // compile all CSF clusters into a single frame
                getColor(cluster_result, clusterframe, csf[i]);
        }
        data_to_file_materials(OUTFILE_CSF);                  // write out the frame
        calcAccuracy(&temp, clusterframe, smodel_csf);        // compute statistics vs. CSF model
        printf("%s accuracy: %3.2f%%\n", smodel_csf, temp.accuracy);
        printf("%s false positives: %3.2f\n\n",smodel_csf,temp.falsepositive);

        wmlen = gmlen = csflen = mslen = 0;
}
```

11.2 filterevm.c

```
/***************************************************/
/* 18-551 Group 13 Project, Spring 2003             */
/* filterevm.c: Communication and file I/O routines  */
/*              for PC-side of "Color Your Head"    */
/* Authors: Jerry Tsai, Yuchun Wang, Jeffrey Yang    */
/* Based on code by <pwb@andrew.cmu.edu>            */
/***************************************************/

#include <stdio.h>
#include <stdlib.h>

#include <common.h>
#include <board.h>          /* EVM library */
#include <pci.h>            /* PCI communication library */
#include <dma.h>

/* Defines for the size of the image */
#define X_SIZE 181
#define Y_SIZE 217
#define BLK_SIZE 22                           // BLK_SIZE = Y_SIZE/n + 2, where n = # of block iterations
#define F_SIZE X_SIZE*Y_SIZE   // F_SIZE = 39277
#define HPIF_SIZE (F_SIZE+3)   // HPI transfers require 4-byte alignment

/* static global variables */
unsigned char wkspace[BLK_SIZE*X_SIZE];
short int front[X_SIZE];
short int back[X_SIZE];
short int top[Y_SIZE];
short int bottom[Y_SIZE];
short int outspace[BLK_SIZE*X_SIZE];
unsigned char kernel[25] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
int frontT;    //threshold from front
int backT;     //threshold from back
int t; // just in case :)


/* malloc'd globals */
short int *result;
short int *tempresult;
unsigned char *frame;
unsigned char *mask;
unsigned short int *sstate;
unsigned short int *color;
unsigned short int *colorends;

/* Function prototypes */
int request_transfer(void *buf, int size, int command);
int wait_transfer();
int dma_copy_block(void *src, void *dest, int numBytes, int chan);
int findT(short int *arr, short int maxT);           //find threshold to isolate the brain
void extractLine(short int *image, int line); //find threshold to isolate the brain
void extractLineV(short int *image, int line);       //vertical version of extractLine
void extractBrain(short int *image, int maxT);                       //extract the brain from the background and border
void zeroPad(short int *image, int line, int firstT, int secondT);   //zero pad to extract the brain
void zeroPadV(short int *image, int line, int firstT, int secondT);  //vertical version of zeroPad
void copyImage(unsigned char *original, short int *new);                     //copy

void quicksort(unsigned char a[], int lowIndex, int highIndex);
void median_filter();
void dilate(short int *inframe, short int *outframe);
int test_dilation(int i, int j, short int *inframe);
void erode(short int *inframe, short int *outframe);
int test_erosion(int i, int j, short int *inframe);

void colorframe();                                   //create colored image
void makeState(int type);                            //combine 3 images

/* Copy output directly from input, for verifying proper I/O. */
void do_comp1(void)
{
        int r, c;// j ,k, t;
        for(r=0; r<Y_SIZE-4; r++){ //x direction - ROW
                for(c=0; c<X_SIZE-4; c++){ // y direction - COLUMN
                        result[(r+2)*X_SIZE + (c+2)] = (short int)frame[(r+2)*X_SIZE + (c+2)];
                }
        }
  }

/* Version 4: Use DMA to copy data */
/* // Cant get this to work!?!?!
void do_comp3x5(void)
{
        int i;

        for(i=0; i < Y_SIZE/(BLK_SIZE-4); i++) {
                dma_copy_block(frame+i*(BLK_SIZE-4)*X_SIZE,wkspace,X_SIZE*BLK_SIZE,0);      // read from frame
                while(DMA0_XFER_COUNTER != 0) ;

                median_filter();

                dma_copy_block(outspace+2*X_SIZE*sizeof(short int), result+((BLK_SIZE-4)*i+2*X_SIZE)*sizeof(short
int),(BLK_SIZE)*X_SIZE*sizeof(short int),1);          // write to result
                while(DMA1_XFER_COUNTER != 0) ;
        }
}
*/

/* Top level for median filter, based on do_comp3 from Lab 3 */
void do_comp3x5(void)
{
        int i, r, c;

        //preload wkspace with BLK_SIZE lines from frame
        for(r=0; r < BLK_SIZE; r++) {
                for(c=0; c<X_SIZE; c++) {
                        wkspace[r*X_SIZE + c] = frame[r*X_SIZE + c];
                }
        }

        for(i=0; i < Y_SIZE/(BLK_SIZE-4); i++) {

                median_filter();

                // copy outspace to result, reload wkspace with next BLK_SIZE lines from frame
                for(r=0; r<BLK_SIZE; r++){ //x direction - ROW
                        for(c=0; c<X_SIZE; c++){ // y direction - COLUMN
                                wkspace[r*X_SIZE + c] = frame[(r+(i+1)*(BLK_SIZE-4))*X_SIZE + c];
                                result[(i*(BLK_SIZE-4)+r+2)*X_SIZE + (c+2)] = outspace[(r+2)*X_SIZE + (c+2)];
                        }
                }
        }
}
/* Used for median filter */
/************************************************************
 ** quicksort C version adapted Jeffrey Yang, 20 Apr 2003 **
 ************************************************************
 * QuickSort.java by Henk Jan Nootenboom, 9 Sep 2002
 * Copyright 2002-2003 SUMit. All Rights Reserved.
 *
 * Algorithm designed by prof C. A. R. Hoare, 1962
 * See http://www.sum-it.nl/en200236.html
 * for algorithm improvement by Henk Jan Nootenboom, 2002.
 *
 * Recursive Quicksort, sorts (part of) a Vector by
 *  1.  Choose a pivot, an element used for comparison
 *  2.  dividing into two parts:
 *      - less than-equal pivot
 *      - and greater than-equal to pivot.
 *      A element that is equal to the pivot may end up in any part.
 *      See www.sum-it.nl/en200236.html for the theory behind this.
 *  3. Sort the parts recursively until there is only one element left.
 *
 * www.sum-it.nl/QuickSort.java this source code
 * www.sum-it.nl/quicksort.php3 demo of this quicksort in a java applet
 *
 * Permission to use, copy, modify, and distribute this java source code
 * and its documentation for NON-COMMERCIAL or COMMERCIAL purposes and
 * without fee is hereby granted.
 * See http://www.sum-it.nl/security/index.html for copyright laws.
 *************************************************************/

void quicksort(unsigned char a[], int lowIndex, int highIndex) {
        int lowToHighIndex;
    int highToLowIndex;
    int pivotIndex;
    short int pivotValue;
    short int lowToHighValue;
    short int highToLowValue;
    short int parking;
    int newLowIndex;
    int newHighIndex;

        if(lowIndex >= highIndex) return;

        lowToHighIndex = lowIndex;
    highToLowIndex = highIndex;

    pivotIndex = (lowToHighIndex + highToLowIndex) / 2;
    pivotValue = a[pivotIndex];

    // Partition:
    newLowIndex = highIndex + 1;
    newHighIndex = lowIndex - 1;
    // loop until low meets high
    while ((newHighIndex + 1) < newLowIndex) // loop until partition complete
    { // loop from low to high to find a candidate for swapping
      lowToHighValue = a[lowToHighIndex];
      while (lowToHighIndex < newLowIndex && lowToHighValue < pivotValue )
      { newHighIndex = lowToHighIndex; // add element to lower part
        lowToHighIndex++;
        lowToHighValue = a[lowToHighIndex];
      }

      // loop from high to low find other candidate for swapping
      highToLowValue = a[highToLowIndex];
      while (newHighIndex <= highToLowIndex && highToLowValue > pivotValue)
      { newLowIndex = highToLowIndex; // add element to higher part
        highToLowIndex --;
        highToLowValue = a[highToLowIndex];
      }

      // swap if needed
      if (lowToHighIndex == highToLowIndex) // one last element, may go in either part
      { newHighIndex = lowToHighIndex; // move element arbitrary to lower part
      }
      else if (lowToHighIndex < highToLowIndex) // not last element yet
      { //compareResult = lowToHighValue.compareTo(highToLowValue);
        if (lowToHighValue >= highToLowValue) // low >= high, swap, even if equal
        { parking = lowToHighValue;
                 a[lowToHighIndex] = highToLowValue;
          a[highToLowIndex] = parking;

          newLowIndex = highToLowIndex;
          newHighIndex = lowToHighIndex;

          lowToHighIndex ++;
          highToLowIndex --;
        }
      }
    }

    // Continue recursion for parts that have more than one element
    if (lowIndex < newHighIndex)
    { quicksort(a, lowIndex, newHighIndex); // sort lower subpart
    }
    if (newLowIndex < highIndex)
    { quicksort(a, newLowIndex, highIndex); // sort higher subpart
    }
  }
/* end quicksort */

//implementation of median filter, for noise removal
void median_filter() {
        int r,c,j,k;
                // 5x5 median filter
                for(r=0; r<BLK_SIZE-4; r++){   // x direction - ROW
                        for(c=0; c<X_SIZE-4; c++){     // y direction - COLUMN
                                for(j=0; j<5; j++){            // row
                                        for(k=0; k<5; k++){    // column
                                                kernel[j*5 + k] = wkspace[(r+j)*X_SIZE + c+k];
                                        } //k=0
                                } //j=0
                                quicksort(kernel,0,24);
                                outspace[(r+2)*X_SIZE + (c+2)] = (short int)kernel[12];
                        } //c=0
                } //r=0
}

/*********************************************************************
 ** Binary erode, EVM version adapted by Jeffrey Yang, 21 Apr 2003    **
 *********************************************************************
 * Implements 15x15 binary erosion on a given frame array.
 *
 * Based on:
 *   Binary Morphological Processing Functions for PIP.
 *
 * Created by:  Tamas Juray (h430845@stud.u-szeged.hu)
 *                     Gabor Torok (h455100@kurrah.cab.jgytf.u-szeged.hu)
 *
 * http://www.inf.u-szeged.hu/~ssip/1996/morpho/source.html
 ********************************************************************/

int test_erosion(int i, int j, short int *inframe)
{
  int i1, j1, k = 0;
  int kernel[225] = {0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,
                     0,0,0,0,0,0,1,1,1,0,0,0,0,0,0,
                     0,0,0,0,0,1,1,1,1,1,0,0,0,0,0,
                     0,0,0,0,1,1,1,1,1,1,1,0,0,0,0,
                     0,0,0,1,1,1,1,1,1,1,1,1,0,0,0,
                     0,0,1,1,1,1,1,1,1,1,1,1,1,0,0,
                     0,1,1,1,1,1,1,1,1,1,1,1,1,1,0,
                     1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                     0,1,1,1,1,1,1,1,1,1,1,1,1,1,0,
                     0,0,1,1,1,1,1,1,1,1,1,1,1,0,0,
                     0,0,0,1,1,1,1,1,1,1,1,1,0,0,0,
                     0,0,0,0,1,1,1,1,1,1,1,0,0,0,0,
                     0,0,0,0,0,1,1,1,1,1,0,0,0,0,0,
                     0,0,0,0,0,0,1,1,1,0,0,0,0,0,0,
                     0,0,0,0,0,0,0,1,0,0,0,0,0,0,0};

  for (i1 = i - 7; i1 <= i + 7; i1++)
    for (j1 = j - 7; j1 <= j + 7; j1++)
          if(kernel[k++] && inframe[i1*X_SIZE + j1] == 0) return 0;
  return 1;
}



int test_dilation(int i, int j, short int *inframe)
{
  int i1, j1, k = 0;
/* // This kernel was too "soft" on the diagonals
  int kernel[225] = {0,0,0,0,0,1,1,1,1,1,0,0,0,0,0,
                     0,0,0,0,1,1,1,1,1,1,1,0,0,0,0,
                     0,0,0,1,1,1,1,1,1,1,1,1,0,0,0,
                     0,0,1,1,1,1,1,1,1,1,1,1,1,0,0,
                     0,1,1,1,1,1,1,1,1,1,1,1,1,1,0,
                     1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                     1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                     1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                     1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                     1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                     0,1,1,1,1,1,1,1,1,1,1,1,1,1,0,
                     0,0,1,1,1,1,1,1,1,1,1,1,1,0,0,
                     0,0,0,1,1,1,1,1,1,1,1,1,0,0,0,
                     0,0,0,0,1,1,1,1,1,1,1,0,0,0,0,
                     0,0,0,0,0,1,1,1,1,1,0,0,0,0,0};
*/
  int kernel[225] = {0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,
                    0,0,0,0,0,0,1,1,1,0,0,0,0,0,0,
                    0,0,0,0,0,1,1,1,1,1,0,0,0,0,0,
                    0,0,0,0,1,1,1,1,1,1,1,0,0,0,0,
                    0,0,0,1,1,1,1,1,1,1,1,1,0,0,0,
                    0,0,1,1,1,1,1,1,1,1,1,1,1,0,0,
                    0,1,1,1,1,1,1,1,1,1,1,1,1,1,0,
                    1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                    0,1,1,1,1,1,1,1,1,1,1,1,1,1,0,
                    0,0,1,1,1,1,1,1,1,1,1,1,1,0,0,
                    0,0,0,1,1,1,1,1,1,1,1,1,0,0,0,
                    0,0,0,0,1,1,1,1,1,1,1,0,0,0,0,
                    0,0,0,0,0,1,1,1,1,1,0,0,0,0,0,
                    0,0,0,0,0,0,1,1,1,0,0,0,0,0,0,
                    0,0,0,0,0,0,0,1,0,0,0,0,0,0,0};



  for (i1 = i - 7; i1 <= i + 7; i1++)
    for (j1 = j - 7; j1 <= j + 7; j1++)
      if (kernel[k++] && (inframe[i1*X_SIZE + j1] != 0)) return 1;
  return 0;
}

void erode(short int *inframe, short int *outframe)
{
  int i, j;

  for (i = 7; i < Y_SIZE - 7; i++) {
    for (j = 7; j < X_SIZE - 7; j++) {
      if (test_erosion(i,j,inframe)) outframe[i*X_SIZE + j] = inframe[i*X_SIZE + j];
      else outframe[i*X_SIZE + j] = 0;
        }
  }
}

void dilate(short int *inframe, short int *outframe)
{
  int i, j;

  for (i = 7; i < Y_SIZE - 7; i++) {
    for (j = 7; j < X_SIZE - 7; j++) {
      if (test_dilation(i,j,inframe)) outframe[i*X_SIZE + j] = frame[i*X_SIZE + j];
      else outframe[i*X_SIZE + j] = 0;
        }
  }
}

/* FindT: Find threshold in a line of input image (X_SIZE long)
 * Algorithm: find a peak above the cutoff value maxT and then
 * find the absolute min in a window of 20 pixels to the right of the peak.
 * The position of this minimum is our brain boundary.
 */
int findT(short int *arr, short int maxT) {
        int i=0;
        int max_index;
        int min_index;
        short int max; //initial max1,max2

        max=arr[0];    //initial max

        // find local max, stop at middle of image,
        // since we will do two thresholds per line,
        // one from the left, and one from the right
        for(i=0; i<X_SIZE/2; i++) {
        if(arr[i] > max && arr[i]>maxT && arr[i+1]<=arr[i]) { //if current value > max, value > threshold, arr[i+i] < arr[i]
                        max = arr[i];  //found max1
                        max_index=i;
                        min_index = i+1;       //set preliminary min
                        break;
                }
        }

        // simply find the minimum, stop after 20 pixels
        // 20 pixels through trial and error:
        for(i=max_index; i<max_index+20; i++) {
                if(arr[i]<arr[min_index]) {
                        min_index = i; //found threshold!
                }
        }
        return min_index;
}

/* Extract a specific line from one slice of data */
void extractLine(short int *image, int line) {
        int i;
        for(i=0; i<X_SIZE; i++) {      //store one line of image
                front[i] = image[X_SIZE*line+i];
                back[X_SIZE-i-1] = image[X_SIZE*line+i];
        }
}

/* Extract a specific vertical line from one slice of data */
void extractLineV(short int *image, int line) {
        int i;
        for(i=0; i<Y_SIZE; i++) {      //store one line of image
                top[i] = image[X_SIZE*i+line];
                bottom[Y_SIZE-i-1] = image[X_SIZE*i+line];
        }
}

/* Extract the brain from the background depending on frontT and backT */
void extractBrain(short int *image, int maxT) {
        int i;

        //create a copy of the original image to apply threshold mask on
        copyImage(frame, tempresult);

        //apply threshold mask line by line (horizontal)
        for(i=0; i<Y_SIZE; i++){
                extractLine(image, i);                          // create front, back
                frontT = findT(front, maxT);                    // front threshold
                backT = X_SIZE-findT(back, maxT)-1;             // back threshold


                if(frontT != 0 && backT != X_SIZE-1)
                        zeroPad(tempresult, i, frontT, backT);          //found both thresholds
                else
                        zeroPad(tempresult, i, X_SIZE, X_SIZE);         //didnt find either/both threshold(s), zero all
        }

        // vertical extraction, for better segmentation
        for(i=0; i<X_SIZE; i++){
                extractLineV(image, i);                         // create top, bottom arrays (bottom = reverse of top)
                frontT = findT(top, maxT);                      // top threshold
                backT = Y_SIZE-findT(bottom, maxT)-1;           // bottom threshold
                if(frontT != 0 && backT != Y_SIZE-1)
                        zeroPadV(tempresult, i, frontT, backT);         //found both thresholds
                else
                        zeroPadV(tempresult, i, Y_SIZE, Y_SIZE);        //didnt find either/both threshold(s), zero all
        }
}

//zero pixels in range, horizontal
void zeroPad(short int *image, int line, int firstT, int secondT) {
        int j,k;
        int offset = line*X_SIZE;
        //take out the first peak
        for(j=offset; j<firstT+offset; j++)
                image[j] = 0;

        //take out the 2nd peak
        for(k=offset+secondT; k<X_SIZE+offset; k++)
                image[k] = 0;
}

//similar to zeroPad, except in vertical mode
void zeroPadV(short int *image, int line, int firstT, int secondT) {
        int j,k;
        //int offset = line; // which row to pad
        for(j=0; j<firstT; j++)                         //take out the first peak
                image[line+X_SIZE*j] = 0;

        for(k=secondT; k<Y_SIZE; k++)  //take out the 2nd peak
                image[line+X_SIZE*k] = 0;
}

//makes a copy of original, used for preserving original input image
void copyImage(unsigned char *original, short int *new){
        int i;
        for(i=0; i<F_SIZE; i++){
                new[i] = (unsigned short int)original[i];
        }
}



//interlace 3 masked images(T1,T2,PD) into one, preparation for kmeans
void makeState(int type) {
        int i;
        for(i = 0; i < F_SIZE; i++) {
                int j= 3*i;
                sstate[j+type] = (unsigned short int)tempresult[i];
        }
}

/* Colorframe: makes colored frame for final output
 * Use color[] array to assign colors to sstate (final output)
 */
void colorframe() {
        // this RGB array must exactly match the PC side RGB array!!
        unsigned char colorR[12] = {0  ,255,255,196,64 ,0  ,128,255,0  ,0  ,0  ,255};
        unsigned char colorG[12] = {0  ,192,128,128,255,255,255,255,0  ,128,0  ,0  };
        unsigned char colorB[12] = {128,255,255,255,64 ,0  ,0  ,0  ,192,255,255,0  };

        int i, c=0;
        for(i = 0; i < F_SIZE; i++) {
                sstate[3*color[i]] = colorR[c]; // red
                sstate[3*color[i]+1] = colorG[c]; // green
                sstate[3*color[i]+2] = colorB[c]; // blue
                if(color[i] == colorends[c])  c++;    // if we found an end value, change the color
        }
}

// From a grayscale source, make a binary mask (src[i] > 0 means "on", src[i] = 0 means "off")
void createmask(short int *src) {
        int i, j, foreground;
        for(i = 0; i < Y_SIZE; i++) {
                foreground = 0;// flags when we cross the background/foreground boundary
                for(j = 0; j < (X_SIZE-1)/2; j++) {  // proceed from the left to the middle
                        if(foreground > 0) {
                                mask[i*X_SIZE+j] = 1;
                        } else if(src[i*X_SIZE+j] > 0) {
                                mask[i*X_SIZE+j] = 1;
                                foreground = 1;
                        } else {
                                mask[i*X_SIZE+j] = 0;
                        }
                }
                foreground = 0;  // reset the flag
                for(j = X_SIZE-1; j >= (X_SIZE-1)/2; j--) {  // proceed from the right to the middle
                        if(foreground > 0) {
                                mask[i*X_SIZE+j] = 1;
                        } else if(src[i*X_SIZE+j] > 0) {
                                mask[i*X_SIZE+j] = 1;
                                foreground = 1;
                        } else {
                                mask[i*X_SIZE+j] = 0;
                        }
                }
        }
}

// Simply apply a binary mask to the dst frame
void applymask(short int *dst) {
  int i;
  for(i = 0; i < F_SIZE; i++) {
        dst[i] = dst[i]*mask[i];
  }
}


/* Set the four edges of result to 0 */
void zero_edges()
{
        int i;

        for(i=0; i<X_SIZE; i++) {
                result[i] = 0;
                result[i+X_SIZE] = 0;
                result[(Y_SIZE-1)*X_SIZE + i] = 0;
                result[(Y_SIZE-2)*X_SIZE + i] = 0;
        }
        for(i=0; i<Y_SIZE; i++) {
                result[i*X_SIZE] = 0;
                result[i*X_SIZE+1] = 0;
                result[i*X_SIZE + (X_SIZE-1)] = 0;
                result[i*X_SIZE + (X_SIZE-2)] = 0;
        }
}

/*====================================== MAIN ============================================*/
int main(void)
{
  char tmp[256];
  int i;
  unsigned int clusternum[1];

  evm_init();               /* Initialize the board */
  pci_driver_init();        /* Call before using any PCI code */

  DMA_AUXCR = 0x00000010;      /* Set priority of HPI over CPU to avoid crashing */

  /* Allocate frame and result in external memory */
  frame = (unsigned char *)malloc(F_SIZE);
  if(frame==NULL) {
    printf("Frame: Couldn't allocate memory...\n");
    exit(1);
  }
  result = (short int *)malloc(F_SIZE*sizeof(short int));
  if(result==NULL) {
    printf("Result: Couldn't allocate memory...\n");
    exit(1);
  }

  tempresult = (short int *)malloc(F_SIZE*sizeof(short int));
  if(tempresult==NULL) {
    printf("Tempresult: Couldn't allocate memory...\n");
    exit(1);
  }

  sstate = (unsigned short int *)malloc(3*F_SIZE*sizeof(short int));
  if(sstate==NULL) {
    printf("Sstate: Couldn't allocate memory...\n");
    exit(1);
  }

  mask = (unsigned char *)malloc(F_SIZE*sizeof(char));
  if(mask==NULL) {
    printf("Mask: Couldn't allocate memory...\n");
    exit(1);
  }

  printf("Allocated memory...\n");


/*==============================  PD =======================*/
  request_transfer(frame, HPIF_SIZE, 0x08);
  printf("Request PD transfer\n");
  wait_transfer();   /* Wait until it's done */

    //apply median filter
  do_comp3x5();
  zero_edges();

  // extracts to tempresult
  extractBrain(result, 58);

  // refine the brain mask
  for(i = 0; i < F_SIZE; i++) { result[i] = 0; }
  erode(tempresult,result);
  for(i = 0; i < F_SIZE; i++) { tempresult[i] = 0; }
  dilate(result,tempresult);

  //save this mask to apply to T1 & T2
  createmask(tempresult);

  // add to state-space plot (for kmeans later)
  makeState(2);

  request_transfer(tempresult, HPIF_SIZE*sizeof(short int), 0x09);
  wait_transfer();
  /*==============================  T2 ========================*/
  request_transfer(frame, HPIF_SIZE, 0x06);
  printf("Request T2 transfer\n");
  wait_transfer();   /* Wait until it's done */

  copyImage(frame, tempresult);
  applymask(tempresult);

  // add to state-space plot (for kmeans later)
  makeState(1);

  request_transfer(tempresult, HPIF_SIZE*sizeof(short int), 0x07);
  wait_transfer();
  /*==============================  T1 ========================*/
  request_transfer(frame, HPIF_SIZE, 0x01);      /* Ask PC to send image data into frame */
  printf("Request T1 transfer\n");
  wait_transfer();                              /* Wait until it's done */

  copyImage(frame, tempresult);
  request_transfer(tempresult, HPIF_SIZE*sizeof(short int), 0x02);
  wait_transfer();

  applymask(tempresult);

  // add to state-space plot (for kmeans later)
  makeState(0);

  // extracted brain
  request_transfer(tempresult, HPIF_SIZE*sizeof(short int), 0x02);
  wait_transfer();
/*======================== KMEANS OUTPUT ===================*/
  request_transfer(sstate, 3*HPIF_SIZE*sizeof(short int), 0x05);
  wait_transfer();
  printf("Kmeans input sent to PC...\n");
```

```
/*========================== GET CLUSTERS ====================*/
  free(frame); free(result); free(tempresult);

  color = (unsigned short int *)malloc(F_SIZE*sizeof(unsigned short int));
  if(color==NULL) {
    printf("color, couldn't allocate memory...\n");
    exit(1);
  }

  colorends = (unsigned short int *)malloc(clusternum[0]*sizeof(unsigned short int));
  if(colorends==NULL) {
    printf("colorends, couldn't allocate memory...\n");
    exit(1);
  }

  // get the # of clusters:
  request_transfer(clusternum, sizeof(int), 0x0B);
  printf("request # of clusters\n");
  wait_transfer();

  // get the kmeans output from PC:
  request_transfer(color, HPIF_SIZE*sizeof(short int), 0xC);
  printf("request clusters\n");
  wait_transfer();

  // get the flag values that specify the end of a cluster:
  request_transfer(colorends, clusternum[0]*sizeof(int), 0x0D);
  printf("request cluster ends\n");
  wait_transfer();

  // create a colored frame:
  colorframe();

  request_transfer(sstate, 3*HPIF_SIZE*sizeof(short int), 0x0E);
  wait_transfer();
  printf("Color result sent to PC...\n");

/*========================== EXIT ===================*/
  printf("Exiting program...");
  sprintf(tmp, "Exiting program...");
  pc_printf(tmp);

  /* A command of 0x04 means to exit the program */
  request_transfer(NULL, 0, 0x04);
```

```
}

/* Use mailbox 1 for address, 2 for size, and 3 for command */
int request_transfer(void *buf, int size, int command)
{
    amcc_mailbox_write(2, size);
    amcc_mailbox_write(3, command);
        pci_message_sync_send((unsigned int)buf, FALSE);
        return(0);
}

/* The PC will send a message when the transfer is complete.  Wait
        for that to happen */
int wait_transfer()
{
        unsigned int value;

        pci_message_sync_retrieve(&value);
        return(value);
}

/* Use the request_transfer framework to send data to be output to
        the screen.  The PC uses command 0x03 to indicate data to be
        output */
  int pc_printf(char *message) {
  int len;

  len = strlen(message);
  while(len%4) len++;  /* Must be multiple of 4 */

  request_transfer(message, len, 0x03);
  wait_transfer();
  return(0);
}

/* dma_copy_block: Copies numBytes from src to dest using DMA channel
        chan.  chan can be 0 or 1.  this function is ASYNCHRONOUS!  You must
        poll DMA0_TRANSFER_COUNT (or DMA1_TRANSFER_COUNT) to see when the
        transfer is complete */
/* Only valid for numBytes < 4 * 0xFFFF */
int dma_copy_block(void *src, void *dest, int numBytes, int chan)
{
  unsigned int dma_pri_ctrl=0;
  unsigned int dma_tcnt=0;
  /* Give DMA priority over CPU, and increment src and dest
      after each element */
  dma_pri_ctrl = 0x01000050;

  /* One frame, and we're using 4 byte elements */
  dma_tcnt = 0x00010000 | (numBytes/4);

  /* Write to DMA channel configuration registers */
  dma_init(chan,dma_pri_ctrl,0,(unsigned int) src,unsigned int) dest,dma_tcnt);

  DMA_START(chan);
  return(OK);
}
```