MPHYGB24 C++ Assignment #3

Felix Bragman
Student Number: 711399

January 4, 2013

Abstract

This report presents a brief overview of the C++ **Image** class created, based on the **nifticlib** library. The class, member variables and functions are outlined. This is followed by a short description of the implementation of two command line programmes; **VolumeQuery** and **VolumeMedian**. Within each description, the methodology employed in testing the implementation of the code is reported.

The code for this project revolves around four building blocks.

1. **Image** class - Image.cpp and Image.h
2. **Stats** functions - Stats.cpp and Stats.h
3. **VolumeQuery** - VolumeQuery.cpp
4. **VolumeMedian** - VolumeMedian.cpp

The **Image** class allows the manipulation of NIfTI data for quantitative purposes. The **Stats** functions allow the statistical analysis of data, including the calculation of the maximum, minimum and median of a set in addition to its standard deviation. **VolumeQuery** is a command line programme based on the **Image** class, which allows statistical analysis using **Stats** of voxels defined within a local neighbourhood Ω centred at a given physical location. It outputs the local statistical measures for all scalar components of a particular voxel. **VolumeMedian** is a second command line programme based on the class **Image**, which performs median filtering of the image volume and saves the filtered data as a new NIfTI file.

1. Image class

The class has the following **public** member functions and capabilities:

1. A constructor that loads the volumetric data within an NIfTI image - `Image(string filename)`
2. A copy constructor - `Image(const Image& rhs)`
3. Accessor functions for all member variables, except the variable **data**, which stores the image data.

4. Getter function to retrieve a voxel magnitude based on image coordinates - `getVoxel(...)` Note: if voxel indices correspond to outside the boundary of the volume, the voxel is set to 0.0
5. Setter function to set a voxel magnitude based on image coordinates - `setVoxelC(...)`
6. Overloaded operators: $<<$, $+$, $+=$ and $=$
7. Functions to determine the physical location (mm) and continuous index of voxels - `physLoc(...)` and `getContLoc(...)`
8. Trilinear interpolation function based on voxel physical locations - `triInterpolate(...)`
9. Function to save an image volume to a niftii file - `saveVol(...)`

1a. Private and **Public** variables and functions

Firstly, it is important to test that the data is being read and saved correctly and that the image properties are also correct. These properties, which are **private** include the following:

1. **`size`** - an array containing the image/grid dimensions in voxels.
2. **`spacing`** - an array containing the voxel physical size in mm.
3. **`origin`** - the physical location of the voxel centred at [0,0,0].
4. **`noOfComponents`** - number of scalar components at each voxel.
5. **`data`** - volumetric data saved as 1D array.

The use of third-party software is employed to test the output of **main**. This includes mricron, NiftiView and the Matlab environment. Mricron corroborates the values returned by **main** for **`noOfComponents`** (33) and **`size`** [112,112,50]. NiftiView provides evidence of the **`spacing`** (2mm^3 isotropic) and the **`origin`** [113.891,-100.418,2.03138], which are also returned within the test script used.

As the **`data`** is stored as a linear array, this requires one to test whether the auxiliary **private** function `sub2ind(...)`, which maps a 4D index to a linear index and `getVoxel(...)` are correctly implemented. The voxels at varying indices are tested and double checked with NiftyView. An example of the test script is presented overleaf.

```
#include <iostream>
#include <string>
#include "Image.h"
#include "Stats.h"

using namespace std;

//  load volume
Image *nim = new Image(filename);

// voxel magnitude at each component
 double *ans = new double[33];
    for (int i = 0; i<33; ++i)
    {
        ans[i] = nim->getVoxel(30,50,26,i);
    }

    cout << "voxel at [30,50,26,0]  = \n" ;
    cout << ans[0] << endl;
    cout << "voxel at [30,50,26,5]  = \n" ;
    cout << ans[4] << endl;
    cout << "voxel at [30,50,26,17] = \n" ;
    cout << ans[16] << endl;
    cout << "voxel at [30,50,26,33] = \n" ;
    cout << ans[32] << endl;
    delete [] ans;
```

Listing 1: Testing **data** and `getVoxel()`

```
voxel at [30,40,26,0]  =
124924
voxel at [30,50,26,5]  =
53111.5
voxel at [30,50,26,17] =
48623.2
voxel at [30,50,26,33] =
66576.4
```

Listing 2: terminal output

Following this, the implementation of `setVoxelC()` and `saveVol()` is tested. This is performed by modifying the voxel value at an arbitrary location by some arbitrary amount. The function `getVoxel()` is used to check that the voxel magnitude has been altered. A new NIfTI file is thus saved and created using `saveVol()`. NiftyView is used to provide evidence that the discussed functions are working. Figures illustrating the test are presented overleaf. Figure 1. displays the original test data for the components tested. Figure 2. illustrates the new data after setting an arbitrary value at an arbitrary voxel. Note the area of high contrast, depicted by the cross-hair showing the new voxel magnitude and the correct implementation of `saveVol(...)`.

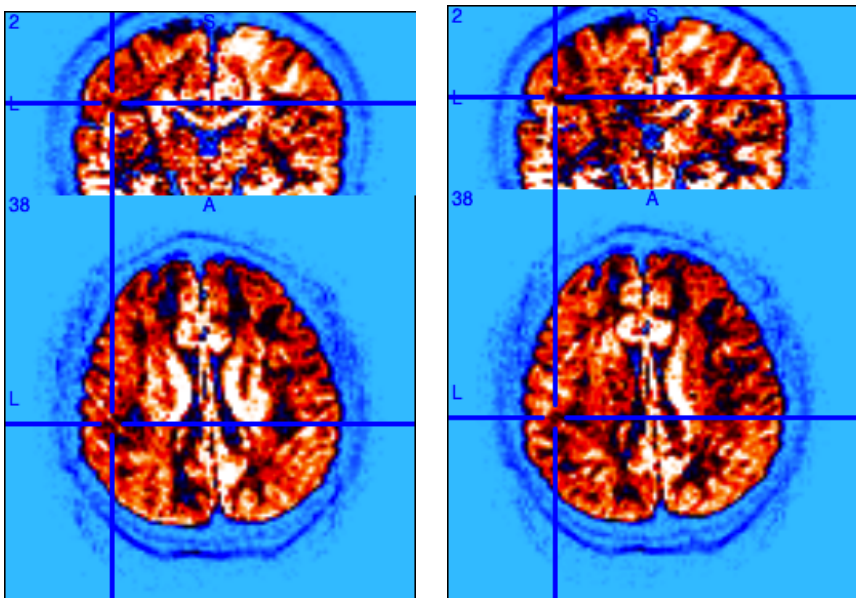
```
 double *T = new double[2];
    cout << "voxel at [30,50,26,5] = " << (*nim).getVoxel(30,50,26,5) << endl;
    cout << "voxel at [30,50,26,17] = " << (*nim).getVoxel(30,50,26,17) << endl;
    cout << "changing voxel at [30,50,26,5] to 690000" << endl;
    nim->setVoxC(30,50,26,5, 690000);
    cout << "changing voxel at [30,50,26,17] to 123456" << endl;
    nim->setVoxC(30,50,26,17, 123456);
    cout << "voxel at [30,50,26,5] = " << (*nim).getVoxel(30,50,26,5) << endl;
    cout << "voxel at [30,50,26,17] = " << (*nim).getVoxel(30,50,26,17) << endl;
    delete [] T;
...

nim->saveVol(filename,output);
```

Listing 3: Testing `setVoxC(...)` and `saveVol(...)`

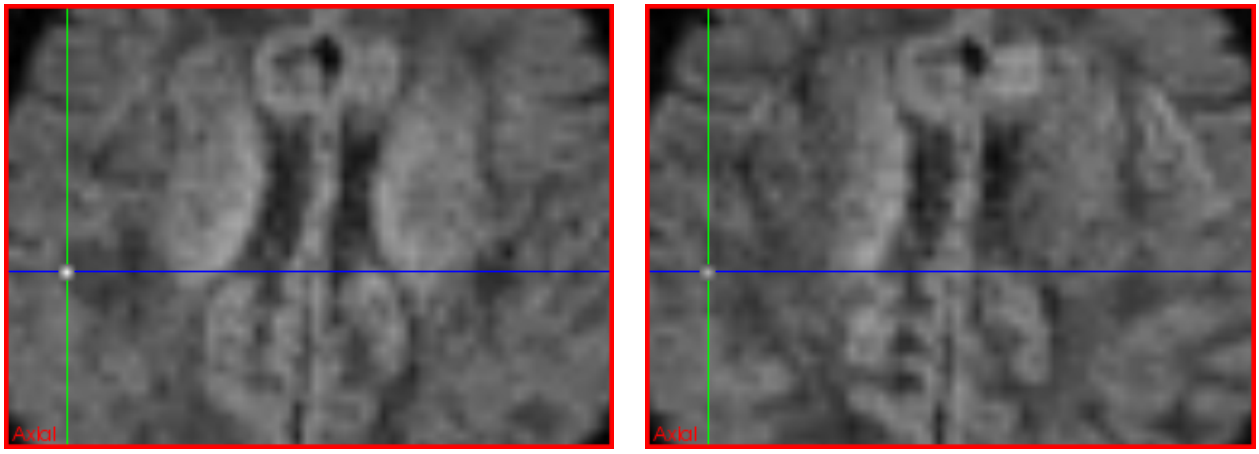
```
voxel at [30,50,26,5] = 53111.5
voxel at [30,50,26,17] = 48623.2
changing voxel at [30,50,26,5] to 690000
changing voxel at [30,50,26,17] to 123456
voxel at [30,50,26,5] = 690000
voxel at [30,50,26,17] = 123456
```

Listing 4: terminal output



a) Component 5 b) Component 17

Figure 1: Test image volume cross-section



a) Component 5 b) Component 17

Figure 2: New axial cross-section with edited voxel magnitude

1b. Trilinear Interpolation

The function `triInterpolate(...)`, which performs the trilinear interpolation of a voxel given a physical location is tested on a simplified case. Please note that the function presented in **Image.cpp** is slightly adapted for this simple test case. A 3 by 3 by 3 image is created, with increasing voxel intensities in the z-direction (Figure 3.). The trilinear interpolation at an arbitrary physical location along the z-axis is calculated. This is compared to the `interp3` function in Matlab for validation. Judging by Listings 6 and 7, one can conclude that `triInterpolate(...)` works correctly.

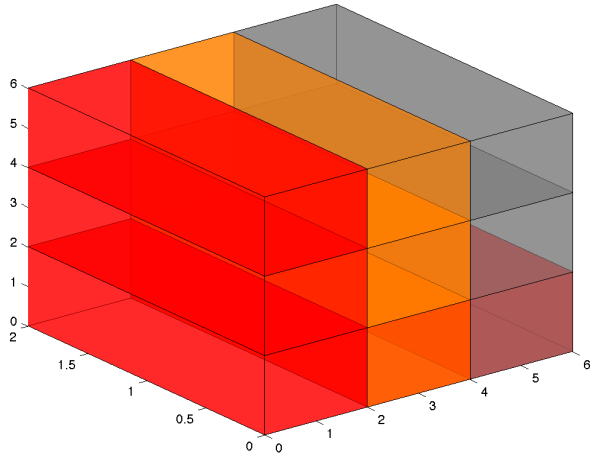


Figure 3: Trilinear interpolation test cube using 3 intensities: 0, 500 and 100

```
//  create 3 by 3 by 3 cube
//  intensity 1: 0 intensity 2: 500 intensity 3: 1000
//  assume voxel spacing = 2mm
//  boundaries 0-6mm in all directions
//  voxel origin at [1,1,1] mm
    double *cube = new double[3*3*3];
    double itsty = 0.0;
    for (int k = 0; k < 3; ++k)
    {
        for (int j = 0; j < 3; ++j)
        {
            for (int i = 0; i < 3; ++i)
            {
                cube[i + j*3 + k*3*3] = itsty;
            }
        }
    itsty += 500;
    }

//  trilinear interpolation (slight adaption to deal with different image grid)

...

//  physical location 1: [1.0 1.0 1.0]
//  physical location 2: [1.25 1.25 1.25]
//  physical location 3: [2.342 2.342 2.342]
//  physical location 4: [3.5 3.5 3.5]
//  physical location 5: [4.75 4.75 4.75]

cout << "voxel magnitude at [1.0 1.0 1.0]: " << voxInterpolate[0] << endl;
cout << "voxel magnitude at [1.25 1.25 1.25]: " << voxInterpolate[1] << endl;
cout << "voxel magnitude at [2.342 2.342 2.342]: " << voxInterpolate[2] << endl;
cout << "voxel magnitude at [3.5 3.5 3.5]: " << voxInterpolate[3] << endl;
cout << "voxel magnitude at [4.75 4.75 4.75]: " << voxInterpolate[4] << endl;
```

Listing 5: Testing `triInterpolate`

```
voxel magnitude at [1.0 1.0 1.0]: 0
voxel magnitude at [1.25 1.25 1.25]: 125
voxel magnitude at [2.342 2.342 2.342]: 171
voxel magnitude at [3.25 3.25 3.25]: 625
voxel magnitude at [4.75 4.75 4.75]: 875
```

Listing 6: terminal output

```
// create 3 by 3 by 3 cube
vol = zeros(3,3,3);
vol(:,:,1) = 0;
vol(:,:,2) = 500;
vol(:,:,3) = 1000;

// voxel centres
X = [0.5,1.5,2.5]; Y = X; Z = X;

// position to evaluate voxel magnitude
// taking into account evaluation position - offset of 0.25

interp3(X,Y,Z,vol,xi(1),xi(2),xi(3),'linear');

x0 = [0.5,0.5,0.5] // equivalent to [1.0 1.0 1.0] mm
ans = 0
x1 = [0.75,0.75,0.75] // equivalent to [1.25 1.25 1.25] mm
ans = 125
x2 = [0.842,0.842,0.842] // equivalent to [2.342 2.342 2.342] mm
ans = 171
x3 = [1.75,1.75,1.75] // equivalent to [3.25 3.25 3.25] mm
ans = 625
x4 = [2.25,2.25,2.25] // equivalent to [4.75 4.75 4.75] mm
ans = 875
```

Listing 7: Matlab interpolation of 3 by 3 by 3 grid

1c. Operator overloading and copy constructor

Testing the implementation for the operator overloading of $+$, $+=$, $=$ and the copy constructor was not possible due to an error in the copy constructor in line 89 of `Image.cpp`.

2. Stats functions

The functions within **Stats** are evaluated by considering the following set of numbers:

$$7, 4, 9, 2, 4, 5, 5, 4$$

The functions `arrayMedian(...)`, `arrayMax(...)`, `arrayMin(...)`, `arrayStd(...)` are tested on this set to verify their implementation. The sample code is presented overleaf in Listings 8 and 9. The terminal output provides justification that the statistical functions are correct implemented.

```
double *te = new double[8];
cout << "Unsorted data: ";
te[0] = 7; te[1] = 4 ; te[2] = 9; te[3] = 2; te[4] = 4; te[5] = 5; te[6] = 5; te
    [7] = 4;
for (int i  = 0; i < 8; ++i)
   {
       cout << te[i] << " ";
   }

int n  = 8;
cout << "\nMedian: " << arrayMedian(te,n) << " ";
cout << "Max: " << arrayMax(te,n) << " ";
cout << "Min: " << arrayMin(te,n) << " ";
cout << "Std: " << arrayStd(te,n) << endl;
```

Listing 8: Testing **Stats** functions

```
Unsorted data: 7 4 9 2 4 5 5 4
Median: 4.5 Max: 9 Min: 2 Std: 2
```

Listing 9: terminal output

3. VolumeMedian

The **VolumeMedian** command line programme allows the user to produce a filtered image based on the median of voxel values within a local neighbourhood Ω defined by an input kernel size in voxels. The pseudo-code shown below in Algorithm 1. demonstrates the algorithm within **VolumeMedian**. It is important to note that voxels at the image boundary may include voxels outside the image grid within Ω. This is dealt with in `getVoxel(...)`, whereby voxels falling outside of the image volume are set to 0.0.

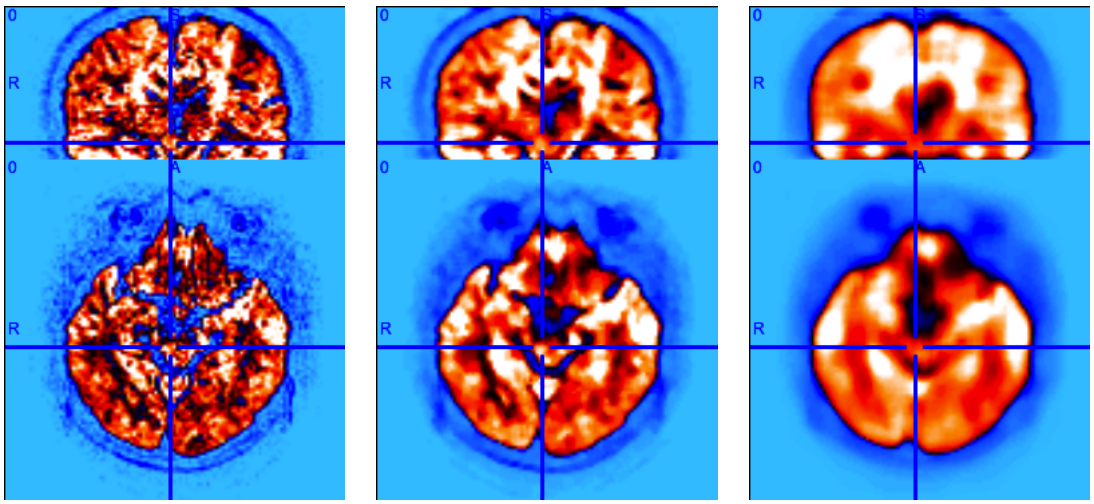
Data: NIfTI **data**
Result: Median filtered **new data**
initialization: load **niftii** object load **dummy niftii** object;
for *all voxels* x_i *within* ***data*** **do**
  initialise **tmp** array;
  for *all voxels within* Ω *centred at* x_i **do**
    fill **tmp**
  end
  median of **tmp** using **Stats**;
  update **data** of **dummy niftii** with median;
  delete **tmp**
end
save filtered image

Algorithm 1: VolumeMedian algorithm

The voxels within Ω are considered by calculating the lower and upper boundaries of the region defined by the voxel indices (i, j, k) and the kernel size. For example, a 1D kernel, size of 5 voxels at an index $i = 72$ would encompass voxels $70 \geq x_i \leq 74$ within the statistical analysis.

Following the testing of the **Stats** functions and the **Setter** and **Getter** functions, the algorithm

of **VolumeMedian** is tested by viewing the filtered result at varying kernel sizes. In addition, a different implementation using a different platform is employed to validate. Three filtered images are presented below; at kernel sizes of 1, 3 and 7 voxels^3. 3D median filtering is performed on the image volume at a kernel size of 3 voxels^3 in Matlab using `ordfilt3D` [1].



a) 1 voxels^3 b) 3 voxels^3 c) 7 voxels^3

Figure 4: Filtered volume of component 19 using **VolumeMedian**

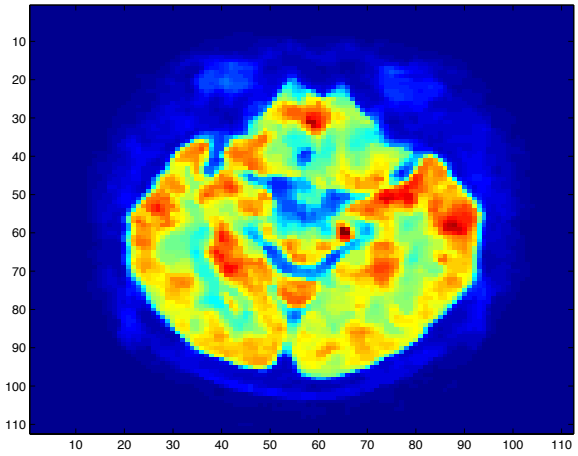


Figure 5: `ordfilt3D` median filter at 3 voxels^3

The aim of a median filter is to reduce "salt and pepper" noise. It is an effective method where the end result lies in reducing noise whilst preserving contrast and edges [2]. Viewing Figure 4., this effect is noticeable. Moreover, one may note that as the kernel size is increased (from 3

[1] Available from: http://www.mathworks.com/matlabcentral/fileexchange/5722. Last Accessed: 03/01/2013

[2] `medfilt2`: http://www.mathworks.co.uk/help/images/ref/medfilt2.html. Last Accessed: 03/01/2013

to 7 voxels^3), the smoothing effect of the filter is more significant; which is expected. Figure 5. displays the output of `ordfilt3D` at a kernel size of 3 voxels^3. One may notice using a qualitative evaluation that the smoothing effect is similar to FIgure 4b and thus presents a certain degree of validation of the implementation of **VolumeMedian**.

4. VolumeQuery

The **VolumeQuery** command line programme allows the users to print the interpolated value of a voxel at a physical location [x,y,z] in addition to statistical values such as the maximum, minimum, median and standard deviation of the voxel values within the neighbourhood Ω, defined by a cube of size provided in voxels^3. The programme outputs this data, formatted within a table for all scalar components of the voxel.

VolumeQuery is dependent on the following, tested functions:

- Accessor functions of private member variables
- `triInterpolate(...)`
- **Stats** functions

Consequently, the correct implementation of **VolumeQuery** is dependent on the implemented algorithm, described below in Algorithm 2.

Data: NIfTI **data**
Result: Statistical analysis of voxels within Ω centred at voxel V at [x,y,z]
initialisation: load **niftii** object;
creation of **User-defined grid** Ω;
calculation of cube boundaries in physical and continuous units;
interpolation of voxel at centred at [x,y,z];
for *all scalar components* **do**
  for *all voxels* x_i *within* Ω *bounded by the physical boundaries* **do**
    get continuous index of voxel V_i from the physical location [x_i,y_i,z_i];
    find nearest voxel in **Image grid** corresponding to voxel V_i;
    calculate voxel boundaries in **Image grid**;
    if *voxel* V_i *is not completely within* Ω **then**
      interpolate voxel magnitude using physical location in Ω
    else
      get voxel magnitude from **Image grid**
    end
  end
  statistical analysis of voxels within Ω using **Stats**
end
print results

Algorithm 2: VolumeQuery algorithm

The reasoning behind the algorithm presented above is as follows. Depending on the input physical location of the centre voxel V_i., the **User-defined grid** Ω will be off-centre with respect to the **Image grid** defined by the indices [i , j , k]. As a consequence, not all voxels

from the **Image grid** will be within Ω. Furthermore, it is problematic to perform the statistical analysis purely on interpolated values. To maximise the accuracy of the algorithm, it is desirable to calculate the *true* magnitude of voxels from the **Image grid** when completely enclosed by Ω. On the other hand, it is not possible to use the *true* magnitude of the voxels at the boundary of Ω. Consequently, one needs to use the trilinear interpolation, implemented within the **Image** class to estimate the intensity within these locations.

The terminal output for a selection of components for the voxel centred at [260.12 , −60.9 , 30.4] is presented below.

```
Regional analysis using a kernel size of 5 voxels^3 at a voxel located at (
    260.12, -60.9, 30.4 )

Component #     Voxel value   Maximum     Minimum     Standard Deviation    Median

1               175792        261817      1496.1      67145                 38150.5
2               23189.5       80041.3     748.049     25944.5               17205.1
3               16457.1       80789.3     748.049     26674.6               9724.64
4               14212.9       71812.7     0           25349.8               13464.9
...
18              14212.9       80789.3     748.049     25862.7               10472.7
19              16457.1       79293.2     748.049     25248.3               13464.9
20              12716.8       79293.2     748.049     26327.7               10472.7
21              20197.3       83033.5     748.049     26479.9               10472.7
...
31              20197.3       77797.1     1496.1      25533.6               10472.7
32              16457.1       84529.6     0           26705.9               10472.7
33              14961         86773.7     0           26464.7               11220.7
```

Listing 10: terminal output of **VolumeQuery**