

NIfTI Shades of Grey: Visualizing Differences in Medical Images

Bachelor Thesis Computer Science

Marco Gunnink - Bart Offereins
University of Groningen

Supervisors:
prof. dr. A.C. Telea
dr. D.P. Williams
Scientific Visualization and Computer Graphics

12th July 2015

Abstract

We propose an application to aid neurologists and clinicians in diagnosing **PD**, **PSP** and **MSA**, by visualizing the differences between a patients PET scan image and the typical brain pattern of each of these diseases, and together in a combined difference image. For calculation of the differences, we use and discuss different algorithms such as (M)SE (P)SNR and SSIM.

Contents

1	Introduction	3
2	Related work	5
2.1	PET scanning	5
2.2	Viewing	6
2.3	Comparing	6
3	Data	8
3.1	Volumes	8
3.2	Scan vs. pattern	8
3.3	Test data	8
4	Architecture	10
4.1	Design choices	10
4.2	Data loading	11
4.3	User Interface	12
4.4	General setup	13
4.5	Comparison	14
4.6	Normalization	14
4.7	Visualization	15
4.8	Rendering	16
4.9	Shading	17
5	Comparison	20
5.1	Primitive comparators	20
5.2	SE and SNR	22
5.3	SSIM	23
5.4	Other index comparators	27
5.5	Windowing	28
6	Discussion and conclusion	29
6.1	Comparators	29
6.2	The program	29
7	Future work	31
	Glossary	32
	References	33
A	UML Diagram	35

1 Introduction

Parkinson's Disease (PD) is a progressive disorder of the central nervous system, that causes symptoms of muscle rigidity, tremors, and changes in speech and gait. Closely related to **Parkinson's Disease** are **Progressive Supranuclear Palsy (PSP)** and **Multiple System Atrophy (MSA)**. They are all disorders of the nervous system, their symptoms are much alike and at the moment there is no cure for these diseases, only treatment and medication that slows down the progression or make life easier for the patients.

The earlier the treatment starts, the better the results for these treatments. Though the symptoms of these 3 disorders are much alike, especially in early stages, the treatments are quite different and a wrong treatment might not help or even worsen the disease. Neurologists keep trying to determine the diseases in an earlier stage by using **Positron Emission Tomography (PET)** scan images and comparing them to images of known patients.

Judging these images and comparing them is done manually by neurologists and clinicians. They perform some simple statistics but most of the work is done by just looking at the images and find the similarities. Our goal is to create a program that indicates/quantifies and visualizes the similarities between 2 images, in order to help neurologists in their task of making the right diagnosis.

The program should meet the following requirements:

1. Load PET scan images.

The program needs to be able to load PET scan images. Preferably we want to support multiple formats, and the program should be extensible in this regard. To ensure basic functionality we want at least one format and set the architecture up in such a way that others can be added later with relative ease.

2. Display the loaded images.

When the images are loaded they will need to be displayed. Though the scans are in 3D, they will have to be displayed in 2D since this is how clinicians are used to viewing PET scans. Additionally, because the program's main focus is comparing images it will have to display multiple images side by side. The program will need to load & display one patient scan image and several pattern images and the visualization of their comparison.

3. Compare two images.

The most important aspect of the program is image comparison. The program should have one or more ways to compare the test image against the patterns. The comparison metrics should help the user in determining which of the patterns matches best with the test image.

4. Visualize similarities and differences.

Finally the comparison between the images should be visualized in some way. Since we're comparing a test image against multiple patterns the program should visualize multiple comparisons so that they can be compared against each other.

So, the program should be able to load multiple PET scan volumes and visualize them on a computer screen. Although there is already some software available that can do this, we haven't come across software that visualizes 2 or more scan images at a time. We decided to build a new program that does not only visualize these images but can compare them as well, using several different comparison metrics.

To achieve this, our first step was to find out what a pet scan is and in what way the existing software handles these pet scans. This is discussed in section 2, along with some general image comparison algorithms that have been used before.

Section 3 covers the the type of data we use and what images we use for testing, followed by the way our program handles this data and an introduction to the user interface in section 4.

Section 5 is an overview of the comparison algorithms we implemented and their results. Finally section 6 describes our evaluation of the program and section 7 contains our suggestions for possible improvements.

2 Related work

Automated image comparison has its application in a wide variety of topics, from comparing subsequent frames in video surveillance to comparing (PET) scans in medical imaging to fingerprint matching in crime scene investigations. Between these applications there is difference in focus and approach of the comparison algorithms. In this chapter we discuss some scientific approaches that might help us create our program. Starting with what a PET scan is and how to view them, followed by studies that show how to interpret them. We finish with some evaluation of generic image comparison algorithms.

2.1 PET scanning

We start off with how PET scanning works. Teune [8] has made a comparison between different neurodegenerative brain diseases and describes how they can be detected using FDG-PET imaging.

Positron Emission Tomography scanning works as follows: before a person is put in a PET scanner, he or she is injected with a radioactive **Fluorodeoxyglucose (FDG)** tracer. This tracer is an analogue of glucose, which is fuel for all human body processes including those in the brain. Once injected, it will be treated as glucose by the body so it will be transported to wherever glucose is needed. The PET scanner will look primarily at the brain. It can then measure gamma rays emitted by the tracer. More gamma rays mean more tracer and therefore glucose uptake, which in turn means more (brain) activity. With triangulation then the exact location of the FDG transmitters is calculated, together with the level of activity, stored in a file, the PET scan.

Figure 1 shows PET scans from PD, PSP and MSA overlayed on top of **Magnetic Resonance Imaging (MRI)** scans. The PET scans in the image are displayed in a scale of red to yellow (rather than the grey-scale we use) to distinguish them from the grey-scale MRI image.

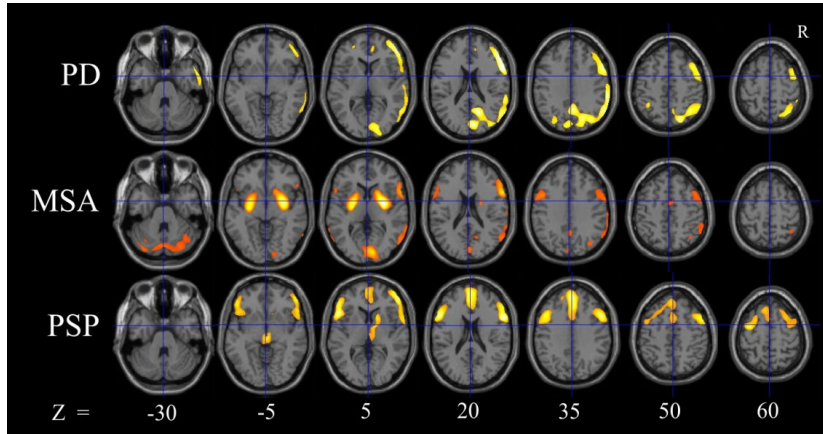


Figure 1: Brain activity from PET scans overlayed on top of MRI scans. (Cropped from [8][fig 1].)

2.2 Viewing

To view this pet scan data, there is free software available, such as MRICron [12] and VOLview [7].

VolView is a comprehensive tool for viewing 3D volume data. It can load a variety of formats among others DICOM and NIFTI. Though the latter is supported under the old ANALYZE format, with which NIFTI is backwards compatible. VolView can show the volumes in 3D, in various ways, or slice by slice in every orientation. It also has advanced shading options for the 3D view and window- leveling for the 2D slice views. Multiple files can be loaded and shown side by side, but the program doesn't do any comparison. Finally, for the formats that contain it, VolView can show patient information alongside the scan images.

MRICron is much simpler than VolView, but it does have the ability to overlay images, using one as a template that another is displayed on top of. This allows the user to specify an MRI image as a template that shows the structure of the brain and then overlay the PET scan. This helps with relating where exactly in the brain the activity is taking place.

2.3 Comparing

To analyse the data, we need knowledge on how to interpret it. Strictly, this falls out of the scope of our project, since we just compare the produced images, but still is good to know.

In 2010, Spetsieris et al.[14] described using **Scaled Subprofile Modeling (SSM)** for comparing fMRI images of 2 groups of PD patients and a control group. The symptoms of PD are somewhat similar to Alzheimers Disease, at least in terms of effects on the brain. Hence, the methods used for detecting Alzheimers also depend on visualizing differences and the studies on Alzheimers can give leads to image comparison.

Fox[6] describes a method to visualize rates of atrophy in Alzheimers disease patients by using automated image extraction. This is used to distinguish atrophy caused by Alzheimers from atrophy caused by aging.

Ashburner & Friston [3] summarize, and introduce some advances to, existing methods of voxel-based-morphometry. This method is used to compare clusters of grey matter in the brain between subjects.

Penney et al [11] compare algorithms that can compare 2-D brain images to 3-D images of a different type. Since PET images are in 3-D, our program does not need this type of conversion, but the article gives an insight in the calculation that is involved with mapping images of different dimensions and sizes.

To compare the images, we take a look at some generic algorithms that assess image quality. Some of them are used in medical imaging and comparisons exist for these algorithms.

Zhou and Bovik discuss whether the use of MSE is appropriate or not [18] and along with others, they propose the **SSIM** algorithm [19].

MSE is a very popular measure of image similarity (or more accurately: dissimilarity) because it's simple, fast and has a very clear meaning. A low MSE means two very similar images and higher values indicate more distortion. However there are also big downsides due to its simplicity. Images that are

visually very similar may have a very high MSE because, for example, every pixel is slightly lighter in one image. To the human eye this doesn't matter much, but in MSE this is translated into an 'error' value, that when summed up for every pixel results in a high MSE. Another problem is that it doesn't indicate how the images are different: do they differ in luminance, is there noise or is one image a blurred version of the original? These can all result in very differently different images with the same MSE score.

To address these issues Zhou and Bovik propose **Structural SIMilarity index (SSIM)**. SSIM is designed to mimick the **Human Visual System (HVS)** by combining three different comparison metrics: luminance, contrast and structure. By selecting different weights for these components you can choose to highlight different aspects of similarity.

Another popular similarity measure is PSNR, based on the SNR for each pixel. As a compromise between SNR and PSNR, Kegelmeyer et al. propose Local Area SNR (LASNR). [10].

3 Data

To visualize differences in images, we first need to know what exactly our data looks like and how it is generated. This section describes what kinds of data the program works with and how. Additionally it describes our test data sets and how we acquired them.

3.1 Volumes

The PET scan produced by the scanner is a three-dimensional image, or **volume**. Just like the individual elements of a 2D image are called pixels, the elements of a volume are called **voxels**.

Before a volume is usable, it is normalized to a mean volume. The result holds grey scale values that can be displayed, where dark areas indicate lower than average activity, and light areas above average activity.

3.2 Scan vs. pattern

Brain diseases generally affect certain regions of the brain, or certain cell groups. [8] Such regions can be found by normalizing to a mean volume that is based only on known patients of a certain disease, and use this as a pattern. Comparing a new patient against such patterns of different diseases, might lead to the right diagnosis.

3.3 Test data

For privacy reasons we were not allowed to use real PET scan data to test our program. Instead, we created our own 3 brain volumes, that represent patterns for the three different parkinsonian diseases and generated 'patient' volumes from these 3 volumes in different proportions and adding noise to them. We refer to the created volumes as Patterns and to the 'patients' as Test volumes. The volumes were generated using a mask so they look structurally similar to a proper PET scan, however the displayed brain activity is purely artificial and has very little basis in reality.

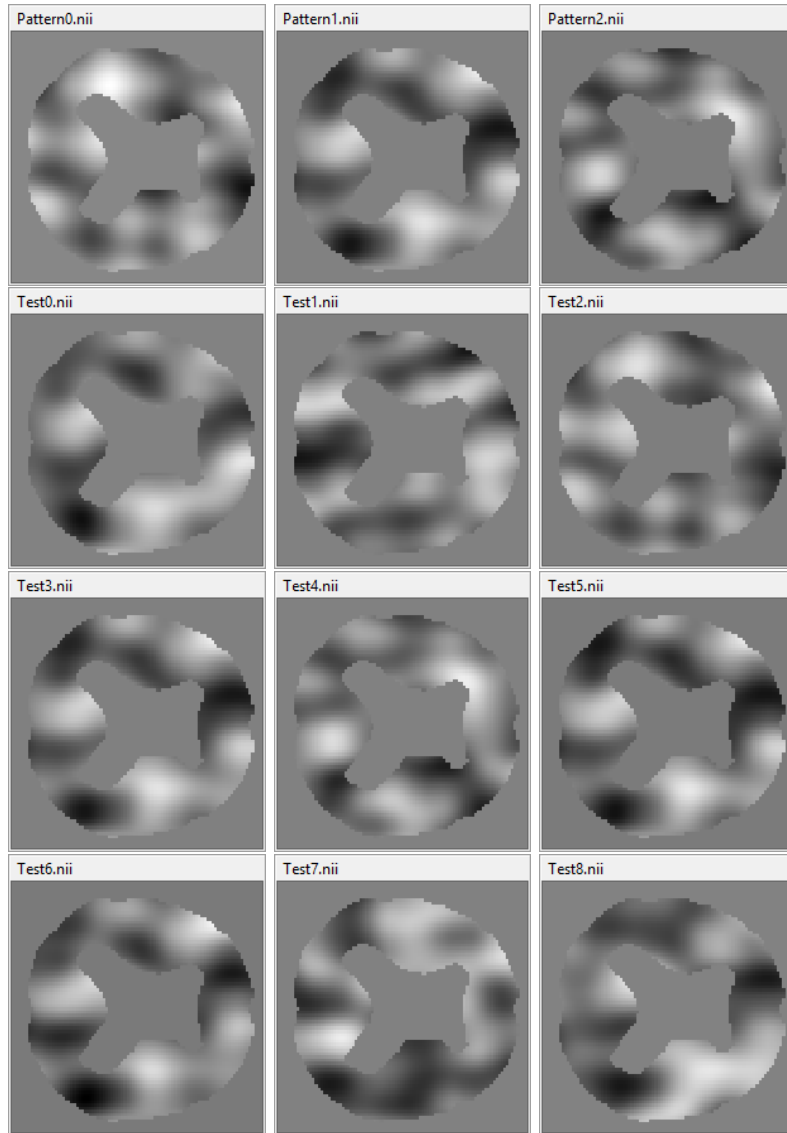


Figure 2: The same slice in the patterns and some of the test volumes

To verify the usability of our test data, we were able to have a quick look at the real patterns and some patient data. It showed that real data generally has sharper outlines and fewer different structures. Consequently, our test data is harder to analyse than the real data and the results of the comparisons are different. However, the impact of sharper images on the comparison quality can be derived, and where applicable, we discuss our expectations and our recommendations for use with the real data.

4 Architecture

This section covers the architecture of the program, starting with the choices we made. It continues with a description of how the program loads the necessary data. Then we go on with how the program looks like and how to use it and finally it gives some background information on how the visualisation and the rendering is done. Appendix A contains an UML diagram with the main components of the program.

4.1 Design choices

Given the assignment with its requirements, there are a few design choices for our program open to us. The first one was the programming language to write it in. We chose C because we're both quite familiar with it, it's simple and small and works well with the libraries we intend to use.

To read the volume files we use the [Neuroimaging Informatics Technology Initiative \(NIFTI\)](#) C library, which is the official implementation of the standard and in the public domain. More on this in section 4.2.1.

To render the volumes efficiently and easily we want to use [OpenGL](#). This way we can leverage the [Graphics Processing Unit \(GPU\)](#) present in most desktop systems and keep the program quick and responsive, at least for the drawing parts. OpenGL also has many image and even volume related operations available so we don't have to re-invent the wheel. However, in order to use OpenGL we need a so called context: a window and related handles that the operating system can put the OpenGL [canvas](#) into.

Several libraries exist that offer this functionality, such as (free)GLUT or GLFW [9]. But these only offer the bare minimum: a window and an OpenGL context. They don't provide other user interface elements, such as buttons, menus or input fields. While it would be possible to implement these in OpenGL this would cost too much time and effort, and fall way outside the scope of this project. Therefore, we need a windowing toolkit: a library that provides these user interface elements. We looked for a windowing toolkit with the following requirements:

- Written in, or accessible from C.
- Support for OpenGL context(s).
- Cross platform, supporting at least Windows and most Linux distributions.
- Freely available and usable (preferably open source).
- Preferably small. The program won't need more than simple elements.

We considered Qt [2] and FLTK [5], but both lack a C interface. Then there is GTK+ [15] which meets all requirements but the last. It is very large, but still an option. Finally the choice fell on IUP [13], created by Tecgraf/PUC-Rio. It is small, cross-platform, written and accessible in C and supports creating and using OpenGL contexts.

4.2 Data loading

In section 3, we described the kind of data we use, namely PET scan volumes. For storing this type of information, there is the **Digital Imaging and Communications in Medicine (DICOM)** standard. Most, if not all, modern PET scanners support this format. If we support the DICOM standard, we would be able to load and visualize any pet scan. However, the DICOM standard is not only proprietary, but DICOM files also contain a lot of information that we don't need and/or are not allowed access to, such as patient names. To overcome both these problems, the **Neuroimaging Informatics Technology Initiative** developed an open file format that contains just the information needed for displaying and analyzing images.

We decided to implement this NIfTI format, not necessarily for its special features, but mainly because its much easier to use via the NIfTI C-library and because we don't need more than just the display information that NIfTI provides. Besides they provide a DICOM-to-NIfTI converter and if it becomes necessary we can always implement a DICOM loader into our program directly.

4.2.1 NIfTI

The **NIfTI** file format consists of a header that describes the type and dimensions of the data. The header and volume information is stored either in 1 file (.nii) or in a separate header and data file (.hdr + .img). The program supports both, as long as a .hdr file and the .img file are in the same directory and have de same basename. When one of the files is loaded, the other one is automatically loaded as well.

Reading these files is done by the NIfTI C library. The **NIfTI** specification allows volume data in various types, ranging from 1-bit black/ white to 24-bit **RGB** colours. The program can handle all these variations. However, it is designed for grayscale images, therefore it converts the loaded data to 32-bit floating point values.

4.2.2 Other formats

Though the program is initially designed to load volumes in the NIfTI file format, though other file formats can easily be implemented. To read other formats a function needs to be implemented that accepts a file name and returns a volume. The program will try every implemented volume loader and use the first non-NULL result. If the given file is not in the expected format for the loader it simply returns NULL. If it is the right format, but otherwise invalid, for example not 3-dimensional or somehow corrupted, it also returns NULL but can additionally log an error message explaining why the volume was not loaded. The program will still continue to try other loaders, but they will fail early since the file won't be in the correct format. Listing 1 shows this process in pseudo-code.

The resulting volume object needs to contain the dimension sizes: the number of slices, rows and columns. It also needs the minimum and maximum voxel values and of course the voxels themselves. The voxels are stored in a 1-dimensional array which contains all voxels in slice-major, then row-major order. Such that a single voxel v at slice s , row r and column c can be accessed via:

Listing 1: Volume loading

```
/* Tries every implemented loader and returns the first succesful result.
 */
loadVolume(file):
    Volume v = NULL
    foreach(Loader l):
        v = l.load(file)
        if(v != NULL):
            return v

/* A loader has the following structure: */
Loader.load(file):
    if(unexpectedFileType(file)):
        return NULL

    if(corrupted(file)):
        log("Volume file is corrupted")
        return NULL

    if(incorrectDimensions(file)):
        log("Incorrect volume dimensions")
        return NULL

    if(allIsWell(file)):
        return makeVolume(file)
```

$$v = V[s \cdot \#rows \cdot \#columns + r \cdot \#columns + c]$$

4.3 User Interface

The user interface of our program, as shown in figure 3 if roughly divided in 4 parts. There is a menu bar that contains the option to load an image, set the comparator and some color settings. The upper part of the main window consists of the images of the test image and the comparison against the 3 reference patterns. The slider sets the slide of the test/pattern that is currently shown. Sliding this will scroll through all volumes simultaneously. The radio buttons set the orienatation of the volumes.

Figure 3 shows the program when started with test image Test0.nii and 3 patterns: Pattern0.nii, Pattern1.nii and Pattern2.nii.

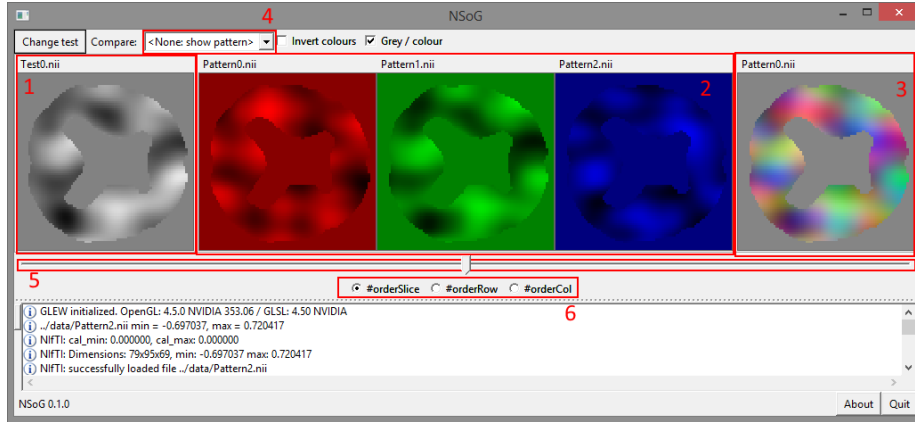


Figure 3: Main view

- 1 Shows the test image.
- 2 Initially shows the pattern images, when a (different) comparator is chosen, the comparison images are shown here.
- 3 The combined pattern-/comparison-images.
- 4 Lists the implemented comparators and is used to select the active one.
- 5 Scrolls through, depending on the selected view order, the slices, rows or columns of all displayed images.
- 6 Switches between the different view-orders.

Below the visualization of the images, there is a log window. Showing information on image loading and rendering, this log window is mainly used for testing and debugging.

Some comparators have additional parameters, such as window size. These can be tuned via a set of controls that are displayed when an applicable comparator is selected. In figure 4 these parameter controls for the SSIM comparator can be seen, to the left of the log messages.

4.4 General setup

As implied in the previous section, the program has one test image. This is the ‘real’ scan from a patient. The test image is matched against a number of pattern images, in our tests and examples: three. These patterns correspond to the expected brain activity of some disease. For every pattern image the program holds a comparison image for every implemented comparator. So the the program will have at most $1 + \#pattern * \#comp$ images loaded.

When choosing a comparator for the first time, the comparison image is computed according to the selected algorithm. This image is then stored in program memory for subsequent selection of the comparator. Since some comparison algorithms require quite a lot of calculation, the initial comparison images might take some time. But once an image is computed, scrolling trough the different

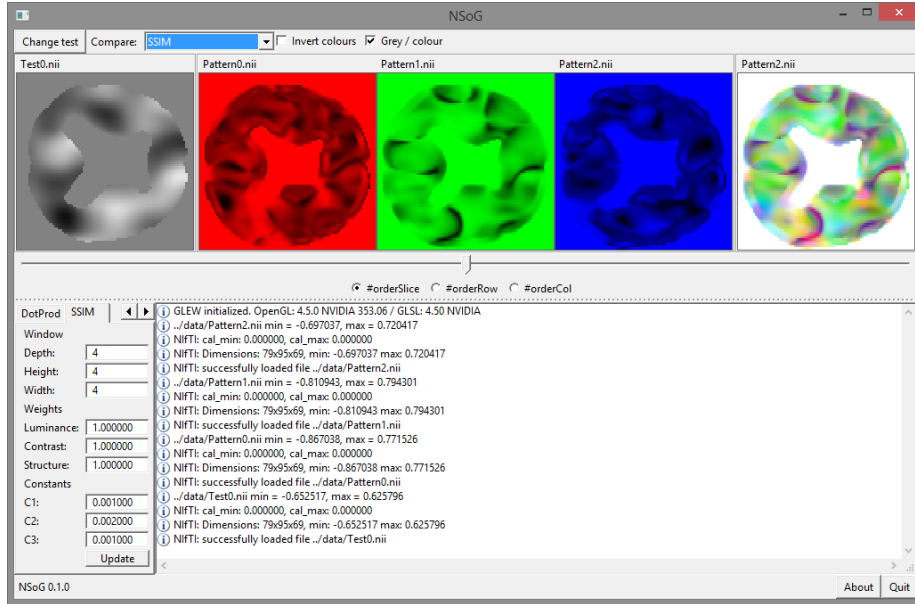


Figure 4: SSIM with parameter controls

slices, or changing orientation is really quick. The program holds only 1 test volume at a time, so when a new test image is loaded, the current one is removed from program memory, along with all its computed comparison images.

4.5 Comparison

To make implementing comparators easy a comparator is simply a function that gets as input 2 volumes and returns a new volume: the comparison image. Besides these volumes the function also gets a pointer to store additional parameter data in. Comparators that use this generally also have a secondary function that creates the appropriate user interface controls to manipulate these parameters. The parameter data is not cleared when a new test image is selected.

4.6 Normalization

For any kind of comparison between 2 images to make sense, the range of possible voxel values needs to be the same for both images. We found that the NIfTI file format does not specify such a range. It does however contain the fields *cal_min* and *cal_max*, indicating respectively the start and end of the range of values that should be displayed [4], but in our test data these values did not appear to correspond to the actual voxel data. The description of these fields states that values outside the specified range should be clamped, indicating that they are used for some sort of window-leveling rather than defining the actual possible values. Thus, images of different sources might have a completely different range. They need to be normalized to the same range before they can be compared.

Lacking an explicitly specified range, we assume that every image contains

(values close to the) minimum and maximum possible values and use these to normalize the images, to assure they have the same range of possible values. This normalization step is done only once, when the volumes are loaded.

While volumes are normalized when they are first loaded, the comparators may emit values that fall outside our normal range. We chose to allow this for several reasons. First, it allows comparators to use other comparators. For example LASNR uses LMSE internally, but this works best when the result of LMSE isn't altered. Secondly, we chose $[-1, 1]$ as our normalization range, while OpenGL uses colour values in the range $[0, 1]$, so there would need to be a correction step anyway. By allowing any range of values in the renderer we can also change our normalization range, should it be necessary, without having to change too much.

Normalizing to a range of $[-1, 1]$ is done by the following formula:

$$vox_{new} = 2 \frac{vox_{old} - min}{max - min} - 1 \quad (1)$$

4.7 Visualization

To visualize the comparison between 2 images or 1 test and a pattern, we show the loaded volumes side by side, with the result of the comparison algorithm in between. This works fine for testing purposes of the comparison algorithm. However, our main objective is to help deciding which pattern is the closest to the subject image, and ultimately whether the subjects brain image can be related to one of the diseases.

We could easily load 1 image and 3 disease patterns and show all source images and a comparison result for the image against each pattern. This would however result in 7 different images on the screen which does not contribute to the overview and usability of the program.

For a new layout we assume that the pattern images don't need to be visible at all times. The test image and especially the comparison images are far more important. Therefore, we show the comparison images in place of the patterns, rather than both side by side. With 3 patterns this brings the total number of displayed images to 4.

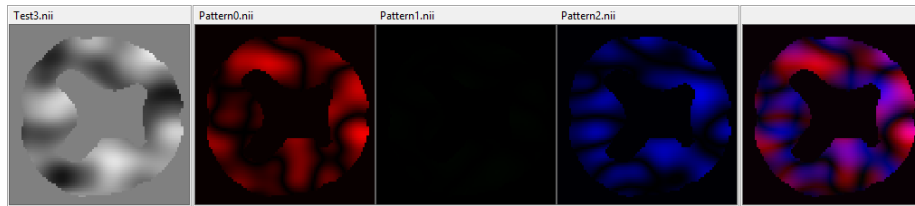


Figure 5: Absolute difference of Test3, in colour

Since all images are in greyscale, and we want to perform 3 different comparisons, we came to the idea to use the different color channels to visualize the differences combined in 1 single image. The pattern images are each assigned their own colour channel then the images are no longer displayed in grey, but shades of red, green or blue, since those are the primary colours that OpenGL uses for rendering. The comparison images (or patterns) with their separate

colours are then merged in another ‘volume’¹. This merged volume helps the user see how the comparison images relate to each other. In figure 5 Pattern0 is mapped to red, Pattern1 to green and Pattern2 to blue. The volume at the far right shows the red and blue peaks combined. The pink spots are the results of overlap between Pattern0 and Pattern2.

4.8 Rendering

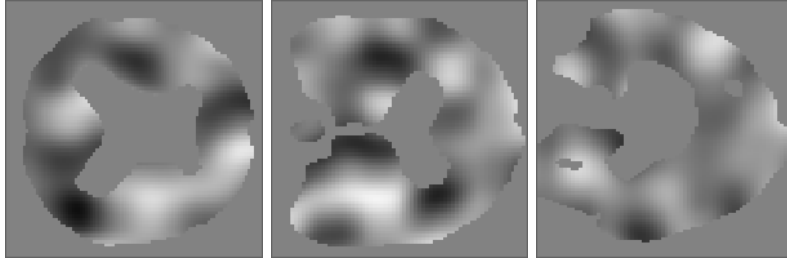


Figure 6: Test0.nii in slice-, row- and column-order

Although **volumes** are in 3D, they are generally displayed in 2D. This is done by dividing the volume in slices. A consequence of this is that one can choose between 3 different perspectives from which to do this slicing. In medical terminology this is equivalent to sliding in the sagittal(ear to ear), coronal (front to back) or transverse (crown to chin) plane. Because the data sources don’t always indicate in what orientation the volume is stored (and when it is, it isn’t always reliable), in the program and this paper we refer to slices, rows and columns. So slicing can be done in slice-order, row-order or column-order (figure 6). The program has no tools to reorientate or scale the input volumes. This means that when comparing different images, they must be stored in the same orientation, and have the same dimensions.

Initially we rendered the volume by drawing a quad for every voxel on the selected slice. While this worked, it was inefficient in both code and execution. And using OpenGL to draw squares is akin to using a crowbar as a bottle opener.

¹In actuality no new volume is generated, combining the colours happens on the GPU.

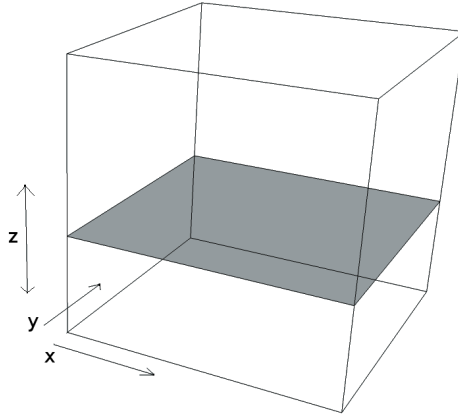


Figure 7: Rendering the volume as a 3D texture

Now rendering is done by uploading the volume to the GPU as a 3D texture. This texture is rendered to a single quad based on the view-order. For slice order the x and y coordinates of the quad are mapped to the columns and rows of the volume, while the selected slice is controlled by a slider in the user interface. For row-order x and y are mapped to the slices and columns and the slider controls the displayed row. Finally column order maps x to slices and y to rows and the column is selected by the slider.

4.9 Shading

With the volumes uploaded to the GPU, the last step in the rendering process is shading. This transforms the floating-point values in the volume to colours on the screen. In OpenGL a **shader** is a separate little program written in a special language called **OpenGL Shading Language (GLSL)**. The shader is split into two parts: the vertex shader and the fragment shader.

In our program the vertex shader does the 3D volume to 2D quad mapping as explained in the previous section. As its input it gets the four coordinates that make up our quad and from those it generates the corresponding coordinates in the 3D texture. Using the information generated by the vertex shader, OpenGL rasterizes the quad: transforming it from the four vertex coordinates into a point for every pixel that will be displayed in the canvas. OpenGL then calls the fragment shader for every pixel to get the colour.

The fragment shader in our program first gets the voxel value, using the coordinates provided by our vertex shader. Because the data in the 3D texture is not normalized by OpenGL standards, it is then normalized to a range of $[0, 1]$. If the 'Invert colours' toggle is turned on the colour value is then inverted. And finally the colour is sent back to OpenGL to be displayed on the screen.

Listing 2: The fragment shader

```

Point3D coord      /* texture coordinate, from vertex shader */

Texture t1, t2, t3 /* references to the textures */
Float min, max     /* min & max voxel values */

Bitmap channels    /* activated colour channels */
Boolean invert     /* whether to invert the colours */

Float r = 0, g = 0, b = 0    /* output */

if(red in channels):
    r = texture(t1, coord)    /* select voxel */
    r = (r - min) / (max - min) /* normalize */

if(green in channels):
    g = texture(t2, coord)    /* select voxel */
    g = (g - min) / (max - min) /* normalize */

if(blue in channels):
    b = texture(t3, coord)    /* select voxel */
    b = (b - min) / (max - min) /* normalize */

if(invert):
    r = 1 - r
    g = 1 - g
    b = 1 - b

return Colour(r, g, b)

```

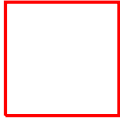
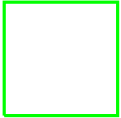
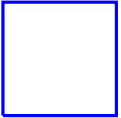
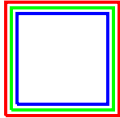
Pattern0	Pattern1	Pattern2	Combined
			
t1 = Pattern0	t1 = Pattern1	t1 = Pattern2	t1 = Pattern 0
t2 = Pattern0	t2 = Pattern1	t2 = Pattern2	t2 = Pattern 1
t3 = Pattern0	t3 = Pattern1	t3 = Pattern2	t3 = Pattern 2
channels = {r}	channels = {g}	channels = {b}	channels = {r,g,b}
r = 0..1	r = 0	r = 0	r = 0..1
g = 0	g = 0..1	g = 0	g = 0..1
b = 0	b = 0	b = 0..1	b = 0..1
t1 → r	t2 → g	t3 → b	{t1,t2,t3} → {r,g,b}

Table 1: Combining images

In order to facilitate the combined, or multi-colour canvas, the fragment shader was modified so that it has access to up to three volumes, instead of just one. This applies to both the ordinary pattern-/comparison-image canvases as well as the combined canvas. While the ordinary canvases have references to three textures, they all point to the same one: the pattern or comparison image that they display. The combined canvas has three different references: one to each of the pattern/comparison images.

Every canvas selects a voxel from the texture based on which colour channels are activated. For the standard canvases this means that they select from their texture and output in only their one colour channel. An ordinary canvas for Pattern0 will have three texture-references to Pattern0.nii, so it will always take voxels from Pattern0. But with only the red channel activated it will only output to the red colour channel, resulting in a black-to-red-scale image. The combined canvas has a texture-reference to Pattern0, one to Pattern1 and one to Pattern2. Since it also has all colour channels activated it will take voxels from Pattern0 and output them to the red channel. Voxels from Pattern1 go to the green channel and voxels from Pattern2 go to the blue channel.

The above applies only when colour mode is activated. When the ‘Grey / colour’ toggle is turned off the ordinary canvases will have all their colour channels activated, just like the combined canvas has. However, since they still only have references to the same texture they still only display their one image. But now they output that image in all colour channels, resulting in grey-scale.

Simplified to pseudo-code, the shader code for each canvas is in listing 2. Table 1 illustrates the configuration for three patterns: Pattern0, Pattern1 and Pattern2 and how they are combined into the combination canvas.

5 Comparison

In this section, we discuss the different comparators we used, and show their results. All the example images are the result of matching Test0 against Pattern0, 1 and 2. Test0 was generated from Pattern1 with 56% similarity. In the equations volume T refers to the test image, and P to the pattern. For comparators with tunable parameters the values were left at the defaults.

5.1 Primitive comparators

A natural approach when creating a difference image, is just taking the per voxel difference. Besides the (simple) difference, there are a few more comparators that operate on a voxel-by-voxel basis.

5.1.1 (Simple) difference

The first comparator we implemented is the ‘simple’ difference: for every voxel subtract b from a (2).

$$diff = T - P \quad (2)$$

In the resulting image dark voxels are the result of a large negative difference, so the pattern image (P) has a bright spot where the test image (T) is darker. Conversely, bright spots on the difference image mean the test image is bright, where the pattern is dark. The grey values that lie in between indicate a lack of difference: similarity.

So, in order to find the pattern that is most like the test image, i.e. the smallest difference, one should look for the difference image with the most grey areas.

The upsides to this approach are that it is simple and quick to compute. However, finding the ‘greyest’ areas to determine similarity is clearly not optimal. While simple difference can indicate big differences between the volumes fairly clearly, in images where the inequality isn’t so pronounced the comparison image won’t be of much help. The color that indicates similarity (50% grey) is not easily identified with the human eye.

5.1.2 Absolute difference

The next comparator is the absolute difference: for every voxel, subtract the higher value from the lower one.

$$absdif = \max(T, P) - \min(T, P) \quad (3)$$

Since the result of this is always positive, black indicates similarity and brighter voxels mean more difference. In this comparator one would look for the darkest image to find the most similar pattern. This is a lot easier to read.

The problem is that, while this comparator indicates where there is difference and how much, it cannot show the ‘direction’ of the difference. Whether the test image is dark and the pattern white (a dark spot in the simple difference comparator), or the other way around, the absolute difference comparator shows a bright spot either way.

5.1.3 Results

In the 'simple' difference, we are looking for the result with the most grey or the least black or white peaks. In figure 8, we can clearly see that that the result on pattern 0 and 2 contain both black and white spots, where pattern 1 contains mostly shades of grey. From this, we can conclude that Test0 is the most similar to Pattern 1.

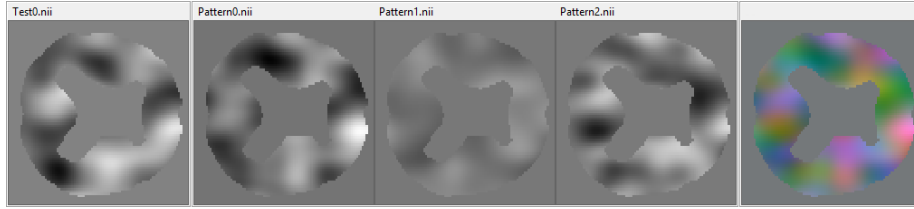


Figure 8: Difference between Test0 and the different patterns

The combined image at the right gives no indication at all that patten 1 (green) is the closest. The lack of red in this image indicates that Pattern 0 is very different from the test, but the intensity difference between the green and blue channel is hard to determine, especially because we are looking for the 50% intensity.

For the absolute difference (figure 9), more black means more similarity. Again this is quite obvious to see. Additionally, in the combined image there is definitely more blue and red than green, indicating the most similarity to Pattern 1 as well.

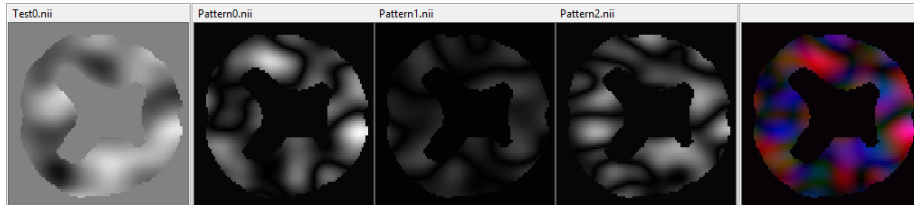


Figure 9: Absolute difference between Test0 and the different patterns

It is slightly counter intuitive, and it can be difficult, to look for the colour that is least present in the image and then conclude that that pattern corresponds the most. In some cases it could be useful to invert the colour scheme and then look for the colour that is most present. In figure 10 in the combined results, green is clearly the brightest.

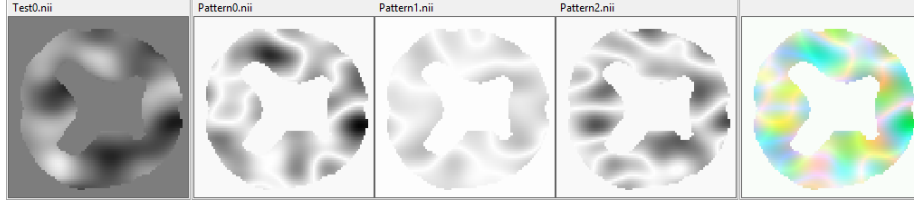


Figure 10: Absolute difference between Test0 and the different pattern with inverted colors.

5.2 SE and SNR

The algorithms above are very easy to implement, but not very powerful for comparison purposes. Two algorithms that are also easy to implement, and regularly used for comparison are **Squared Error (SE)** and **Signal-to-Noise Ratio (SNR)**. These are often used as **Mean Squared Error (MSE)** and **Peak Signal-to-Noise Ratio (PSNR)** respectively to generate a single index value for a whole image, but we want to highlight local differences. To do that, these algorithms compute the raw, un-averaged SE and SNR.

5.2.1 Squared Error

SE is simply the difference, squared:

$$SE = (T - P)^2 \quad (4)$$

Like absolute difference, this comparator generates dark values where the images are similar and goes brighter as they are more different. While this means that it suffers from the same lack of difference directionality, SE has the advantage that it amplifies bigger differences and reduces the smaller ones. The resulting image therefore is less noisy than the one generated by absolute difference. Consequently it is easier to immediately see whether the test image is similar to the pattern and where the biggest differences lie.

5.2.2 Signal-to-Noise Ratio

The **Signal-to-Noise Ratio** of an image is acquired by dividing each element of P by the corresponding element of T (5).² The result of this division is transformed to a logarithmic scale, resulting in a value range that is visualizable.

$$SNR = 10 \cdot \log_{10}(P \circ \left(\frac{1}{T}\right)) \quad (5)$$

5.2.3 Discussion and results

The squared error results in Figure 11 shows clearly that Pattern1, which is practically black, is most similar to Test1. Contrary to the simple difference the result image is no use for determining the rate of similarity, so it is not much of an improvement over the simple difference.

² \circ is the entrywise product of 2 matrices

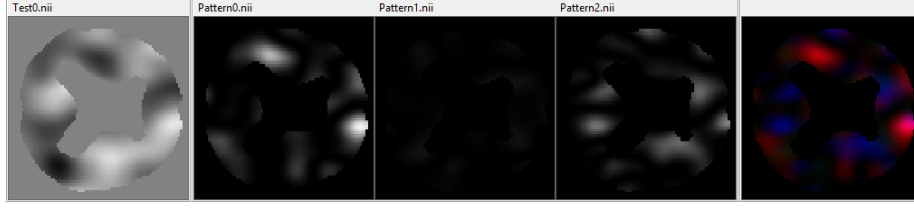


Figure 11: Squared error between Test0 and the different patterns

Though SNR is often used in image comparison, it doesn't work well for our purposes. Our images don't differ so much in noise as they do in structure, so the results of the comparison are usually too intensive to gather any useful information. Figure 12 is indeed not very helpful in determining the best match.

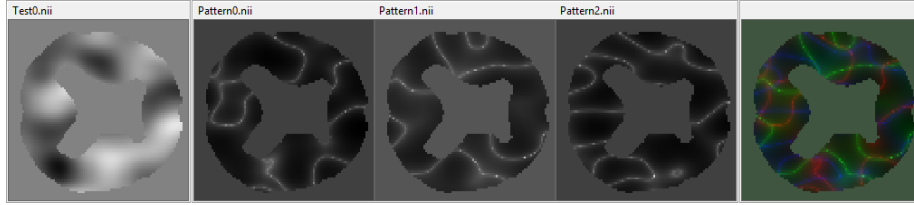


Figure 12: Signal-to-noise ratio for Test0 and the different patterns

5.3 SSIM

The comparisons above are all voxel based, comparing the absolute (color) value, with no regard for structure. Since we are more interested in the difference in structure than the the actual pixel values we implemented the **Structural SIMilarity index** by Bovik et al [19]. This algorithm considers 3 components: luminance, contrast and structure, which are combined to a single value, where each component has its own weight. (6) The full SSIM formula can be found in (12).

$$SSIM = luminance^{\alpha} \cdot contrast^{\beta} \cdot structure^{\gamma} \quad (6)$$

By using these three components, SSIM aims to mimic the **HVS**. Humans are capable of seeing structure and depth, for example. This requires quite a lot of calculation by the brain, so the brain tends to takes shortcuts by heuristics based on the laws of physics. In fact, most of what we 'see' is constructed by our brain.

This means that the human sight can be tricked by manipulating those heuristics. In other words, human sight is extremely subjective. In general this can be useful, but in our case we want an objective tool to guide the human eye and lead the neurologist toward the right diagnosis.

But why use an algorithm that mimics the HVS to avoid the weakness of the human eye? First of all, SSIM is criticized for not simulating the human eye accurately enough [16, 17], but still it tries. The answer lies in the 3 components of SSIM and their different weights.

5.3.1 Luminance

The first component is luminance. The human eye is perfectly capable of recognizing the same picture with different luminance levels. If we would perform (2) on 13, there would be a noise in the difference image because the colors differ in intensity between the 2 images. For our purpose it would be helpful if the comparison filters this kind of noise. However, the NIfTI images are not constructed with a camera, but levels of FDG in a PET scan. Therefore, the result of the luminance component isn't really meaningful, and actually really close to 1 most of the times. We choose $\alpha = 0$ in (6), eliminating the luminance component.



Figure 13: The same picture with different lighting

5.3.2 Contrast

Contrast is one of the things the human eye can be tricked with. If there is a sharp contrast with a light background, we perceive a darker color, and vice versa, while an algorithm just can take the actual values to objectively determine the contrast between an image and its background. Looking at 14, humans might find it difficult to see that all squares are the same color, even while knowing they are.



Figure 14: All grey squares are exactly the same color

We can use this objective measurement of high contrast to get a better overview of the relevant differences between the 2 images. In the original SSIM formula, the contrast component is defined as (7) and designed to actually mimic the behaviour mentioned above by taking the standard deviation and divide it by the squared standard deviation. This results in sharp edges getting

smoother and really soft edges getting sharper. This seems odd, we want to avoid the human bias. Though, despite the edge ‘manipulation’, the σ and μ are still objective measurements. Therefore we suggest a $\beta \neq 0$.

$$contrast = \left(\frac{\sigma_T \sigma_P + c_2}{\sigma_T^2 + \sigma_P^2 + c_2} \right)^\beta \quad (7)$$

where σ is the standard deviation

$$\sigma_x = \left(\frac{1}{N-1} \sum_{i=1}^N (T_i - \mu_T)^2 \right)^{\frac{1}{2}} \quad (8)$$

and μ the mean intensity

$$\mu_x = \frac{1}{N} \sum_{i=1}^N T_i \quad (9)$$

5.3.3 Structure

The third component of SSIM is structure. Humans generally have no problem recognizing structure. In 14, most people recognize the middle squares on the grey background. For computers this apparently simple task can be quite difficult. For this particular image it would be enough to group all pixels with the same colour, if there was no compression loss. But if the squares contained textures of some kind, just grouping colours is not enough.

This is exactly the situation for a typical PET scan analysis. We are not looking for structures of the exact same color, but groups of values that are similar. SSIM uses the Pearson correlation coefficient to find this similarity.

$$structure = \left(\frac{\sigma_{TP} + c_3}{\sigma_x \sigma_y + c_3} \right)^\gamma \quad (10)$$

$$\sigma_{TP} = \frac{1}{N-1} \sum_{i=1}^N (T_i - \mu_T)(P_i - \mu_P) \quad (11)$$

Combining this components results in (12). We choose $\alpha = 0$, omitting the luminance component. In our program it is still implemented, leaving the choice of α to the user.

$$SSIM = \left(\frac{2\mu_T \mu_P + c_1}{\mu_T^2 + \mu_P^2 + c_1} \right)^\alpha \cdot \left(\frac{\sigma_T \sigma_P + c_2}{\sigma_T^2 + \sigma_P^2 + c_2} \right)^\beta \cdot \left(\frac{\sigma_{TP} + c_3}{\sigma_T \sigma_P + c_3} \right)^\gamma \quad (12)$$

SSIM can be used to generate one single index for the whole image, like MSE and PSNR we mentioned earlier. For image quality assessment, Bovik et al suggest [19, p. 6] to apply the SSIM index locally, by computing (12) for every pixel with a certain window. In section 5.5 this is explained further.

5.3.4 Results and discussion

Together with SSIM, we introduced a few variables that need to be set. We already mentioned the α , β and γ component weights from (12), every component has a C constant as well. Their sole purpose is to prevent the equation getting unstable when the denominator is 0. Bovik proposes C_1 and C_2 to be equally (really) close to 0 and C_3 to be half this size. We use these values as proposed to ensure this stability.

Then there is the window size. This should be adjusted to the size of the structures that are to be identified. With our particular test data, a window size of 4 on each dimension, yields the best results. Based on what we saw from the real volumes, this window can and should be bigger but we have not been able to test this.

The default values for α , β and γ are set to 1 each. together with the default window and C values this yields the result as shown in figure 15. The comparison image shows more detail than the test image and the previous comparators, especially for patterns 0 and 2. More interestingly, the combined image shows mostly green, even outside the actual pattern, where we expect no differences at all.

It turns out that this is an effect caused by the C constants. When we increased them by a factor of 10 the backgrounds became much lighter and showed less bias toward a single image, regardless of its similarity. However, the comparison images also turned fuzzier and lost a lot of structure. Lowering the C constants returned much of the structural information, at the cost of messing with the background somewhat.

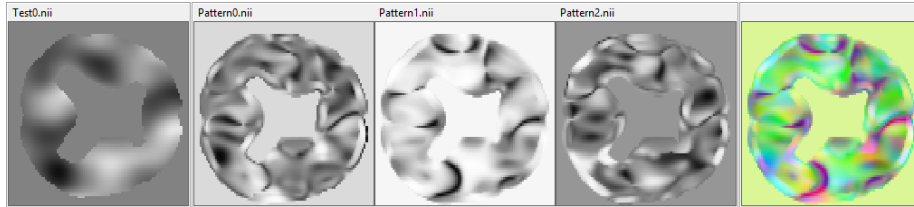


Figure 15: SSIM between Test0 and the different patterns

We proposed SSIM mainly for the structure component, and in de discussion on the different components we proposed to omit luminance, give some weight to contrast but focus on structure, so we set the values α , β and γ to 0, 1 and 2 respectively. The values for β and γ are somewhat arbitrary, but should serve well as a starting point.

The result is shown in figure 16.

In SSIM, white (or color) means similarity. This, combined with the high level of detail results in images that are easy to understand. In figure 16, it is clear that the test matches pattern 1 and where the regions with te most similarity are. The image for pattern 1 is mostly white and the combined image mostly green. On the other hand, patterns 0 and 2 contain so much black that we start losing information. So clearly, the parameters need some additional tuning.

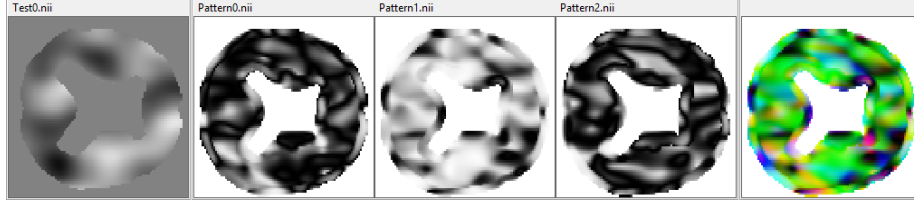


Figure 16: SSIM with $\alpha = 0, \beta = 1, \gamma = 2$

5.4 Other index comparators

Bovik's idea of generating an index for every voxel with a certain window, is also applicable to other other index algorithms like MSE and PSNR [1]. We decided to implement a windowed MSE and PSNR the same way as we did with SSIM, and compare the results. We refer to these as LMSE and LPSNR respectively.

5.4.1 LMSE and LPSNR

The MSE index is a very rough metric for similarity of two images. It can be useful for pre-selection, a very high MSE indicates completely different images, but it is no use for detailed similarity detection.

MSE is simply the mean of the squared errors of every voxel in the image (section 5.2.1). **Local Mean Squared Error (LMSE)** is the same for a subset of voxels, in a window around the calculated voxel.

$$LMSE_{s,r,c} = \frac{1}{dhw} \sum_{(z=-d/2)}^{d/2} \sum_{(y=-h/2)}^{h/2} \sum_{(x=-w/2)}^{w/2} (T_{s+z,r+y,c+x} - P_{s+z,r+y,c+x})^2 \quad (13)$$

LMSE is a compromise between SE and MSE. It is based on multiple voxels, and yet takes into account the local differences within the image. This behaviour can be optimized by tuning the window size. Then it might be a good help in finding the similarities between images.

The relation LPSNR/SNR is similar to (L)MSE/SE, only it is not about the mean of voxel values, but the maximum (PSNR).

$$\begin{aligned} PSNR &= 10 \cdot \log_{10} \left(\frac{Max(T \cup P)^2}{MSE(T, P)} \right) \\ &= 20 \cdot \log_{10} \left(\frac{Max(T \cup P)}{\sqrt{MSE(T, P)}} \right) \\ &= 20 \cdot \log_{10}(Max(T \cup P)) - 10 \cdot \log_{10}(MSE(T, P)) \end{aligned}$$

$$LPSNR_{s,r,c} = 20 \cdot \log_{10}(Max(LMSE(T, P))) - 10 \cdot \log_{10}(LMSE(T, P)_{s,r,c}) \quad (14)$$

5.4.2 Result and discussion

Though windowing works well for SSIM, because it has to: SSIM computes a single index over the whole image. Computing MSE and SNR within a window doesn't provide us with better images than just using the immediate, non-averaged, values. All we get is a blurry version of the original image. They might work for bigger images, to smooth out peak values, but with the fairly small resolution of our data those peaks are too significant to miss.

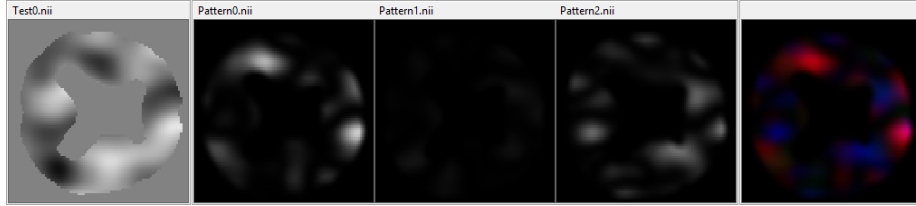


Figure 17: Local MSE between Test0 and the different patterns

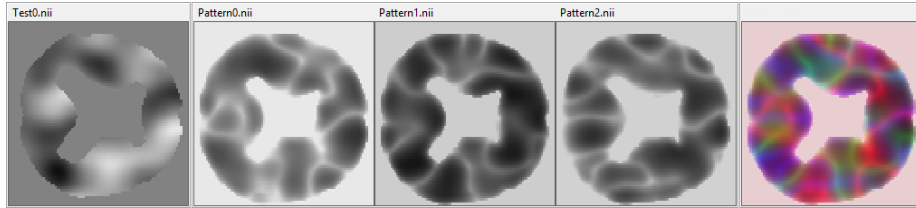


Figure 18: LPSNR between Test0 and the different patterns

5.5 Windowing

In sections 5.3 and 5.4 the algorithms are performed on a subset of the volumes, defined by a window. For example LMSE is computed by taking the MSE of all the voxels in a window around the position of the voxel that is being computed. A window measures d slices, h rows and w columns. The comparison image C is then computed by taking values from T and P within this window around the targeted voxel at s, r, c and averaging them. Or in pseudo code:

Listing 3: Determine the window

```

for(z = -d/2 to d/2):
  for(y = -h/2 to h/2):
    for(x = -w/2 to w/2):
      C[s][r][c] += cmp(T[s + z][r + y][c + x], P[s + z][r + y][c + x])

C[s][r][c] /= d * h * w

```

For LMSE the cmp function would be MSE, for LPSNR cmp is SNR and to make the SSIM image cmp is the SSIM function itself.

6 Discussion and conclusion

In this section we summarize and discuss the various comparators explained in section 5 and give our evaluation of the program as a whole.

6.1 Comparators

The simple difference comparator does not provide very clear images. Though it can indicate the ‘direction’ of difference (whether the test is lighter or darker than the pattern), it is too hard to find similarities because you are looking for the ‘greyest’ areas in a greyscale image.

Absolute difference is better: similarity is indicated by darker colours, black being most similar and white most different. However, this comparator does suffer from its simplicity. It shows where, and to some extent how much, the images are different, but this results in a comparison image that shows a lot of white making it harder to determine where the more important differences lie.

Squared error is, in our opinion, one of the best comparators in our program. Bysquaring the differences it emphasizes big differences, and diminishes the smaller ones. This makes it easy to quickly spot the most similar pattern, the darkest one, and see where the greatest differences lie. Of course, the downside of this is that it may hide small but potentially significant differences. Our recommendation is that squared error be used as a quick overview and then another comparator to figure out the details.

Signal-to-noise, on the other hand, turned out nearly useless for our purpose. The images it produces are hard to nearly impossible to interpret and it only gives somewhat clear results when two images are practically the same. And at that point the other comparators are equally good at indicating the near equality.

SSIM is another contender for best comparator. Where Squared Error is good for making a quick judgement which pattern is the best, SSIM can then be used to get more detailed information. One of the strengths of SSIM is its tunability, but this is at the same time a weakness. Getting a useful result out of SSIM may require quite some parameter tweaking, which combined with the slowness of the algorithm can be very time consuming. We propose the following parameter settings: 0 for luminance, since this component of the algorithm does not apply to our data. Set contrast to 1 and structure to 2 so that the comparison focuses on structure, complemented by contrast.

Unlike SSIM, Squared Error and Signal-to-noise don’t benefit from having a window applied to them. Both algorithms produce a fuzzier version of their non- windowed counterparts, which doesn’t aid much in the comparison.

6.2 The program

We made a program that can load and display PET scans. It can deal with files in the NIFTI format and others can be implemented as needed. The program loads one file as a test image and can load one or more files as pattern images. Initially the patterns are displayed side by side with the test image, when a comparison algorithm is chosen the resulting images are displayed in place of the patterns. Finally there is a combined image that shows a mix of the patterns or comparison images. The program allows the user to scroll through the slices

of all displayed images simultaneously and to choose in which orientation they are rendered.

It can be really helpful to use (a combination of) these side by side comparisons to find the closest match, but we could not extract one single measure that completely quantifies the similarity of three patterns to a test volume.

We implemented several comparison algorithms with varying degrees of usefulness. Squared Error can be used to quickly see which of the patterns is the closest match, but the comparison image isn't very clear in what the differences are exactly. SSIM on the other hand, being a much more complicated algorithm, also generates more complicated comparison images. They are much harder to make sense of at first, but with proper parameter tuning the images can provide a lot of information. Local MSE and Signal-to-Noise ratio, on the other hand turned out nearly useless. Even though they are commonly used when comparing images, the numbers they produce don't translate into comparison images very well.

These comparators are based on generic image quality assesment algorithms. To get more accurate results it would be good to implement algorithms that can deal specifically with PET scans. However before that we need to know what medical experts think of our comparisons.

We have evaluated our comparators ourselves, but we are not medical professionals. We understand the basics of PET scans, but without proper training we lack the expertise to accurately judge them and consequently, our comparison images. Before our program can be used it will have to be tested by neurologists and clinicians who can evaluate the validity and usefulness of our algorithms and the images they produce.

7 Future work

Computing the comparison images takes a noticable amount of time, during which the program is non-responsive to user input. Especially calculating SSIM for 3 patterns can take as much as 15 seconds, even on a recent Intel Core i5³ system. One way to speed this up would be to make use of the multi-core processing capabilities of most modern CPUs by using multiple threads. Another way would be to move calculating the comparison images on the GPU instead. Though we would need to find a way to pass the additional parameters that some of the comparators use to the GPU, without introducing an extravagant amount of variables that would go unused by most algorithms.

Some volume-viewing programs allow the user to specify another volume to be used as a template. It can help the user find specific areas of the brain in the PET scan. Our program does not support this as of yet. It should be fairly straight- forward to implement overlaying two volumes, however making sure they are aligned correctly could pose a bigger challenge. Especially if they are not the same size.

While the NIfTI file format is somewhat common, it is not nearly as ubiquitous as the DICOM format. A NIfTI-to-DICOM converter does exist, but it would be much more user-friendly if our program could load DICOM files directly. Due to the way we set up the volume loading part of the program it is fairly easy to facilitate this, with the help of a DICOM loading library in C.

There are a number of extensions and improvements to SSIM, mostly claiming to have a better approximation of the human eye. In principle, this is not what we are looking for. However, we would like to investigate these further.

Finally, as with any software and especially software that involves a fair amount of user interaction, there are probably still quite a few bugs hiding under the surface. These bugs can pretty much only be found by just using the software a lot, and having it tested by people with varying degrees of familiarity with our program and similar software.

³Intel Core i5-4690K at 3.9Ghz

Glossary

- canvas** A free-form drawing area in the program. 10
- DICOM** Digital Imaging and Communications in Medicine. 11
- FDG** Fluorodeoxyglucose. 5
- gait** The pattern of movement of the limbs. 3
- GLSL** OpenGL Shading Language. 17
- GPU** Graphics Processing Unit. 10, 17
- HVS** Human Visual System. 7, 23
- image** The visual representation of a volume. 13
- LMSE** Local Mean Squared Error. 27
- MRI** Magnetic Resonance Imaging. 5
- MSA** Multiple System Atrophy. 1, 3
- MSE** Mean Squared Error. 22
- NIFTI** Neuroimaging Informatics Technology Initiative. 10, 11
- OpenGL** A software library for accelerated graphics rendering. 10
- PD** Parkinson's Disease. 1, 3
- PET** Positron Emission Tomography. 3–5, 8
- PSNR** Peak Signal-to-Noise Ratio. 22
- PSP** Progressive Supranuclear Palsy. 1, 3
- RGB** Red, green and blue: additive colour model used in most digital devices.
11
- SE** Squared Error. 22
- shader** A program that tells a computer how to draw something in a specific way. 17
- SNR** Signal-to-Noise Ratio. 22
- SSIM** Structural SIMilarity index. 6, 7, 23
- SSM** Scaled Subprofile Modeling. 6
- volume** A 3-dimensional set of data-points. 8, 16
- voxel** A single element of a volume. 8

References

- [1] Peak signal-to-noise ratio as an image quality metric. Technical report, National Instruments, 9 2011.
- [2] Qt — Cross-platform application & UI development framework. <https://www.qt.io/>, june 2015.
- [3] John Ashburner and Karl J. Friston. Voxel-based morphometrythe methods. *NeuroImage*, 11(6):805 – 821, 2000.
- [4] Alex Clark. nifti-1 header field-by-field documentation. http://nifti.nimh.nih.gov/nifti-1/documentation/nifti1fields/index_html, may 2015.
- [5] Bill Spitzak et al. Fast Light Toolkit (FLTK). <http://www.fltk.org/index.php>, june 2015.
- [6] Nick C Fox, Peter A Freeborough, and Martin N Rossor. Visualisation and quantification of rates of atrophy in alzheimer’s disease. *The Lancet*, 348(9020):94 – 97, 1996.
- [7] KitWare. VolView. <http://www.kitware.com/opensource/volview.html>, june 2015.
- [8] A. L. Bartels L. K. Teune and K. L. Leenders. *FDG- PET Imaging in Neurodegenerative Brain Diseases, Functional Brain Mapping and the Endeavor to Understand the Working Brain*, chapter 22. InTech, 2013.
- [9] Camilla Berglund Marcus Geelnard. GLFW - An OpenGL library. <http://www.glfw.org/>, june 2015.
- [10] Laura Mascio Kegelmeyer, Philip W. Fong, Steven M. Glenn, and Judith A. Liebman. Local area signal-to-noise ratio (lasnr) algorithm for image segmentation, 2007.
- [11] G.P. Penney, J. Weese, J.A. Little, P. Desmedt, D.L.G. Hill, and D.J. Hawkes. A comparison of similarity measures for use in 2-d-3-d medical image registration. *Medical Imaging, IEEE Transactions on*, 17(4):586–595, Aug 1998.
- [12] Chris Rorden. MRICron. <http://www.mccauslandcenter.sc.edu/mricro/mricron/>, june 2015.
- [13] Antonio Escañó Scuri. IUP - Portable User Interface. <http://webserver2.tecgraf.puc-rio.br/iup/>, june 2015.
- [14] Phoebe G. Spetsieris and David Eidelberg. Scaled subprofile modeling of resting state imaging data in parkinson’s disease: Methodological issues. *NeuroImage*, 54(4):2899 – 2914, 2011.
- [15] The GTK+ Team. The GTK+ Project. <http://www.gtk.org/>, june 2015.

- [16] Bo Wang, Zhibing Wang, Yupeng Liao, and Xinggang Lin. Hvs-based structural similarity for image quality assessment. In *Signal Processing, 2008. ICSP 2008. 9th International Conference on*, pages 1194–1197, Oct 2008.
- [17] Z. Wang, E.P. Simoncelli, and A.C. Bovik. Multiscale structural similarity for image quality assessment. In *Signals, Systems and Computers, 2004. Conference Record of the Thirty-Seventh Asilomar Conference on*, volume 2, pages 1398–1402 Vol.2, Nov 2003.
- [18] Zhou Wang and A.C. Bovik. Mean squared error: Love it or leave it? a new look at signal fidelity measures. *Signal Processing Magazine, IEEE*, 26(1):98–117, Jan 2009.
- [19] Zhou Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *Image Processing, IEEE Transactions on*, 13(4):600–612, April 2004.

A UML Diagram

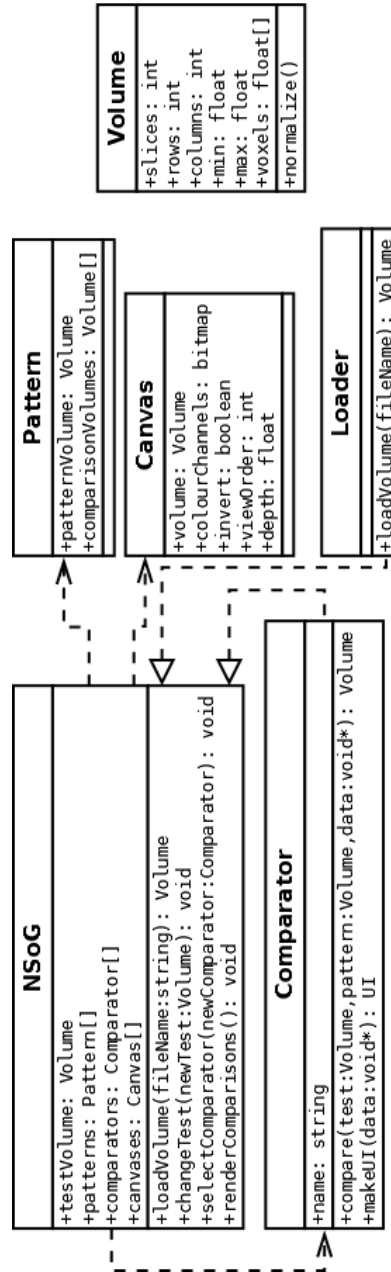


Figure 19: UML diagram of the program's main architecture