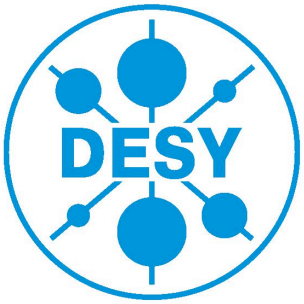

My Summer Studentship at DESY Hamburg IT Department

Oscar Byrne, University of Birmingham, England

September 12, 2013

Abstract

This report describes 2 projects: creating an application for reading NeXus file metadata and making it available over the web via WSDL; and evaluating the decay of magnesium in a corrosive medium from raw tomographic data.

Contents

1 Introduction

I was working in the DESY IT department in Hamburg during the summer of 2013. I worked alongside Darie Picu, a Computer Science and Physics joint honours student from Edinburgh University.
With particular thanks to Dr. Steve Aplin, Dr. Felix Beckmann, Freiderike Nowak and Jürgen Starek for their help and guidance throughout my stay at DESY.

2 Task 1: NeXus to WSDL

Our first task was to develop a computer program for making experiment information more accesible via the web. This consisted of 2 main parts:

- Accessing the relevent metadata from the NeXus file describing the experiment
- Making this data available over the web via WSDL

2.1 Choosing a Language

Before coming to DESY I only really had experience using C++, which is a relatively low-level programming language. This means that it takes a lot of code to do something but can be relatively fast, which is why it is often used in physics. For this quite simple task where speed is not really an issue, me and my partner Darie decided that C++ would be the wrong tool for the job. In the end we decided to use Python - neither of us had much experience with scripting languages, and we both felt this simple task would be a good oppertunity to learn something new.

2.2 Accessing the NeXus file

The NeXus format is an extension of XML which was developed as an attempt to standardise the experimental data produced by neutron, x-ray, and muon science for easier collaborative analysis and visualisation. Like XML, it works as a system of nodes connected in a tree - see figure 1 for the structure of a typical file.
Using the NeXus API, it is possible to recursively loop through this tree and locate the relevent metadata. Our script then stores this in a python standard-library database. Given more time, we would have liked to use a more advanced database such as Apache (the database we used is practically analgous to storing the data in a text file), but neither of us had experience with databases and we decided it would not be a significant-enough improvement to the program to justify the time spent learning something brand new. The final structure of the script written to read the NeXus files is shown in figure 2.

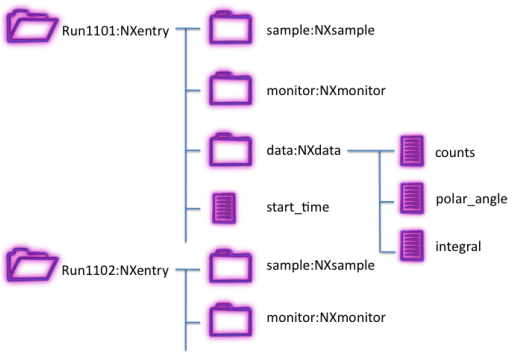

Figure 1: Structure of a Nexus file

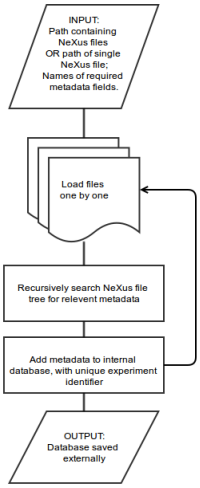

Figure 2: Logic for our program

2.3 Accessing WSDL

The task was to make the data collected by the program in figure 2 easily accesile va te web. We used WSDL ('Web Service Description Language'), which is a way of defining a series of functions attributed to a server that are callable by clients via SOAP (which is a protocol for sending structured information over the web, in this case operating using HTTP although any transport protocol can be used).

```
−<wsdl:definitions targetNamespace="nexus.wsdl.http" name="Application">
 −<wsdl:types>
  −<xs:schema targetNamespace="nexus.wsdl.http" elementFormDefault="qualified">
   −<xs:complexType name="getResponse">
    −<xs:sequence>
      <xs:element name="getResult" type="xs:string" minOccurs="0" nillable="true"/>
     </xs:sequence>
    </xs:complexType>
    <xs:complexType name="get_metadata"/>
   −<xs:complexType name="get_metadataResponse">
    −<xs:sequence>
      <xs:element name="get_metadataResult" type="tns:stringArray" minOccurs="0" nillable="true"/>
     </xs:sequence>
    </xs:complexType>
   −<xs:complexType name="get">
    −<xs:sequence>
      <xs:element name="exp_id" type="xs:string" minOccurs="0" nillable="true"/>
      <xs:element name="path" type="xs:string" minOccurs="0" nillable="true"/>
     </xs:sequence>
    </xs:complexType>
   −<xs:complexType name="stringArray">
    −<xs:sequence>
      <xs:element name="string" type="xs:string" minOccurs="0" maxOccurs="unbounded" nillable="true"/>
     </xs:sequence>
    </xs:complexType>
    <xs:element name="stringArray" type="tns:stringArray"/>
    <xs:element name="get" type="tns:get"/>
    <xs:element name="get_metadataResponse" type="tns:get_metadataResponse"/>
    <xs:element name="get_metadata" type="tns:get_metadata"/>
    <xs:element name="getResponse" type="tns:getResponse"/>
   </xs:schema>
 </wsdl:types>
−<wsdl:message name="get_metadata">
   <wsdl:part name="get_metadata" element="tns:get_metadata"/>
 </wsdl:message>
−<wsdl:message name="get_metadataResponse">
   <wsdl:part name="get_metadataResponse" element="tns:get_metadataResponse"/>
 </wsdl:message>
−<wsdl:message name="get">
   <wsdl:part name="get" element="tns:get"/>
 </wsdl:message>
−<wsdl:message name="getResponse">
   <wsdl:part name="getResponse" element="tns:getResponse"/>
 </wsdl:message>
−<wsdl:service name="NexusMetaService">
```

Figure 3: Portion of the WSDL file defining our service

WSDL is, again, based on XML and is designed to be both machine- and human-readable. However, we found that it is entirely possible to create a WSDL service without ever even having to see the WSDL file that defines it. We used Spyne, a python library which makes it very simple to define many different kinds of web services. To create a WSDL service, it is as simple as defining a python method that has the required functionality and Spyne generates the required WSDL code to make it callable over the net. The final python script controlling the server was only about 50 lines long, with a further 10 for the client script used to connect to the server.

3 Task 2: Evaluating Magnesium Decay

3.1 Introduction to the Problem

Sometimes in surgery it is necessary to insert a device into the body. The problem with this is that the foreign body has to be removed later on down the line with a second surgery which can be both expensive and dangerous. A modern solution to this is to create devices that are designed to dissolve whilst in the body, thereby removing the need for a second surgery. It is important to know exactly how these devices will corrode, however, because if they break prematurely then the device will stop working as intended. This is measured by taking tomograms of the device, which is made of magnesium, after different lengths of time in a corrosive medium designed to mimic blood. Our task, then, was to design a program that can evaluate the evolution of the corrosion of the magnesium by looking at the raw tomographic data.

3.2 Detail of the Problem

In our case, because these kinds of devices are still fairly far from the market, we were studying the decay of a simple magnesium tube. In the tomogram, this decay manifests as 3 discrete regions of what should be uniform density - the uncorroded magnesium; the corrosion layer; and regions where the magnesium has completely dissappeared. This is visible in figure 3.2 where black is the corrosive medium, dark grey is the corrosion layer and light grey is the uncorroded magnesium. Because the tube is not made of pure magnesium, but an alloy, there are also highly-absorbant particles present in the material which manifest in the tomogram as white 'noise'.

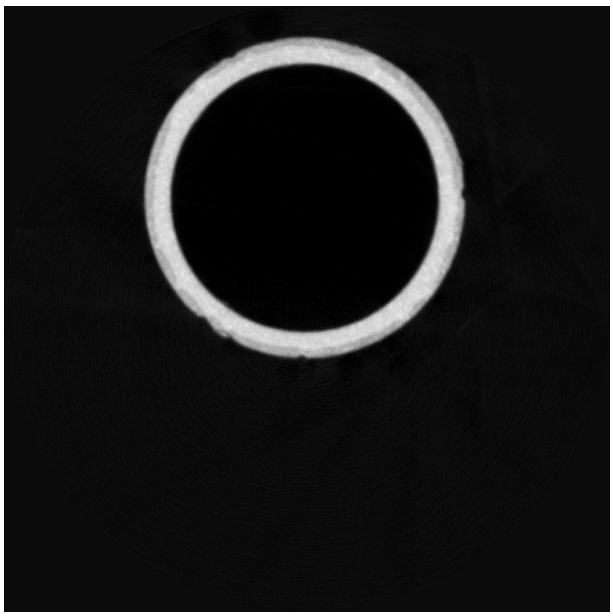

Figure 4: Typical unfiltered tomogram slice

3.3 Step 0: Getting Started / Installing Libraries

For this project, we were told that python would be a good language to use. This is because, initially, our supervisor envisioned quite a complicated work flow which would process the data using python and then use Mathcad for the analysis. Python therefore would have the advantage of being a dynamic language, able to easily invoke other programs.
Our supervisor asked us to use the vigra library for image processing, which is written in C and highly optimised but can be controlled with a python wrapper. He also asked us to install this on scientific linux 6. To test this, I installed SL6 as a virtual machine, which was completely new for me and quite scary. It turned out to be not as difficult as I was expecting, however, although I did come up against some difficulties with how scientific linux assigning drive permissions. I used VirtualBox to simulate the machine and it was a simple matter of downloading the boot image and then installing the operating system via URL.
Vigra was a bit more tricky to install, however. When starting I had not had much experience installing libraries and had only ever compiled from source. I really struggled to install vigra from source because it required many layers of abstraction - for example, the documentation is not hosted online but rather encoded in the source code and must be compiled into HTML using some other librarys. After many frustrated attempts of compiling the source code which lasted almost a week I was able to install the library in a few minutes by using the RPM package manager. This was frustrating, but a good learning experience because I gained experience in troubleshooting installations,

for example how the file system hierarchy functions, and came to realise the usefulness of package managers.
Despite this progress, I was still unable to install the python wrapper for vigra. This is because it was only availabe via pip (a utility for installing python modules), and has not been maintained. The link is now dead and in fact points to a URL that makes no mention of the project or anybody involved with it! In fact, vigra's creator has apparently since 'dropped out', stopped answering emails and is no longer active in the scientific community. This was interesting for me because it demonstrates how tools can quickly become outdated and unusable in computer science, and that they are not as safe from personal issues as one might hope.
Having hit this brick wall we were forced to abandon vigra. We implemented the image processing routines entirely in Python using the SciPy stack (a collection of Python librarys including NumPy) and in fact found that this was not significantly slow. Calculations never took more than overnight to complete, and generally didn't take more than a few minutes. What follows is a description of our process in analysing the tomographic data.

3.4 Step 1: Filtering the Data

The raw data is noisy and must be filtered. We started with an anisotropic diffusion filter, which is an edge-preserving filter widely used in scientific image processing. There are 2 main parameters for such a filter - the number of iterations, and kappa which acts as a sort of threshold gradient for edges to preserve. Setting a large value of kappa therefore makes the anisotropic diffusion act like gaussian diffusion. Setting a too-small value, however, results in noise being treated as edges which should be preserved, making the filter ineffectual. After trial-and-error, we found that the most effective method was to combine an anisotropic filter with a small kappa value with an additional, small-kernal median filter to clean up any residual noise (for each pixel, the median filter looks at a given kernal size of neighboring pixels and chooses the median value). A comparison of a typical tomogram slide before and after filtering is shown in figure 3.4.

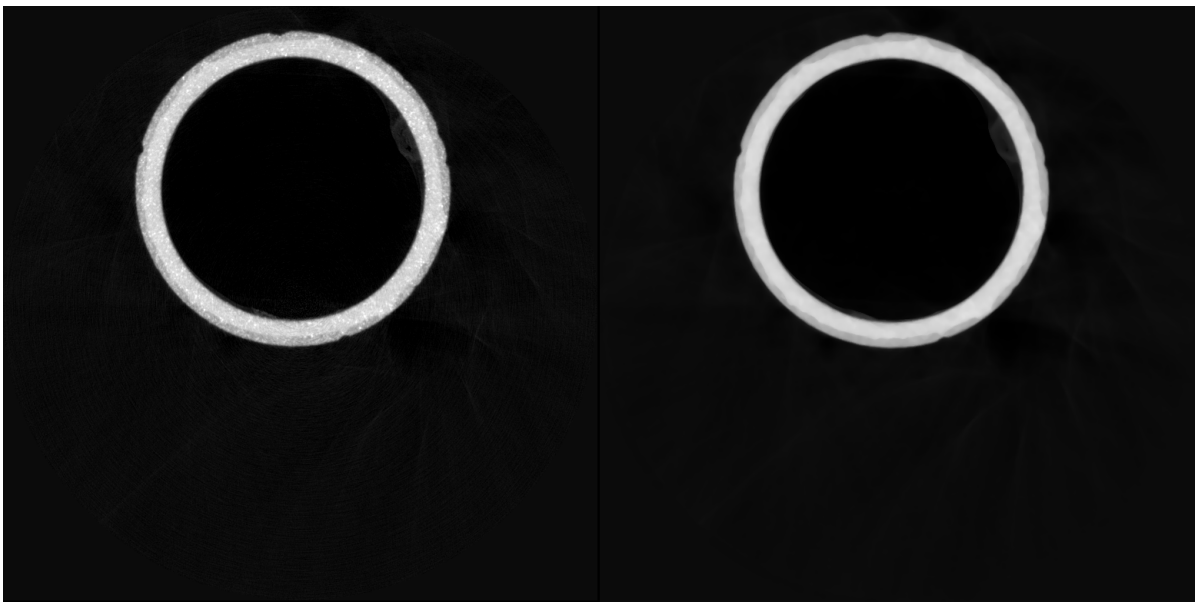

Figure 5: Showing before and after filtering for a typical tomogram slice

3.5 Step 2: Finding the Threshold Values

Having filtered the image, the next stage is to enable the computer to identify the 3 stages of corrosion so we can evaluate how they evolve. This is conceptually fairly simple - first we take an histogram of the tomogram data. This should show 3 gaussian peaks for the 3 regions. The thresholds are then calculated by normalising the gaussians (to take into account the different sizes of the regions - for example the tomograms are mostly corrosive medium and so the first peak is very large) and finding where they intersect.

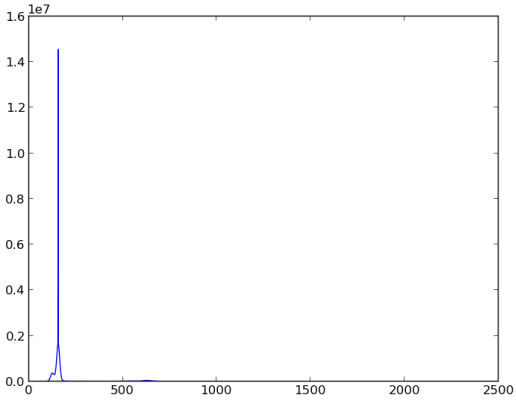

Figure 6: Histogram of the tomogram data split into 2048 bins

In practice, however, we came up against some problems in this process. There is background signal (visible to the left hand side of the gaussians in figure **??**) that means that it is difficult to fit gaussians to the raw data reliably. Our solution to this was to simply cut off the data below the background signal level - the trimmed data is shown in figure 3.5. This allowed us to arrive at a much nicer fit for the gaussians and hence a better approximation for the threshold (see figure 3.5).

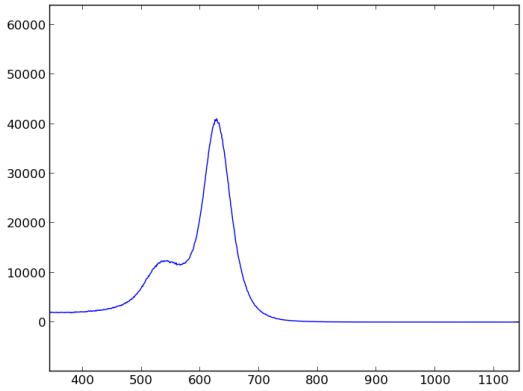

Figure 7: Close up of the peaks due to the magnesium in the hitogram

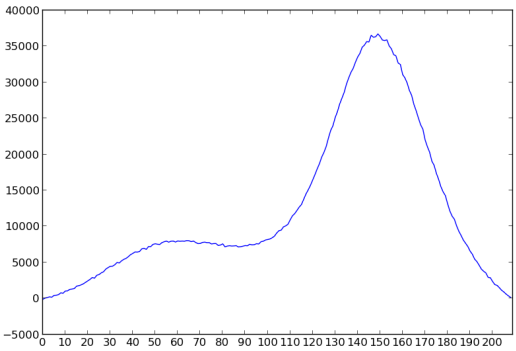

Figure 8: The parsed histogram data

This background signal, and the fact that the peaks for the corrosive medium and magnesium are so far apart, also means that it is difficult to find the threshold between the corrosive medium and magnesium - it is fairly meaningless to find the intersection because it lies well below the background signal. Therefore, failing any more rigorous definition, we classified the edge of the gaussian as being six times its width from the centre.

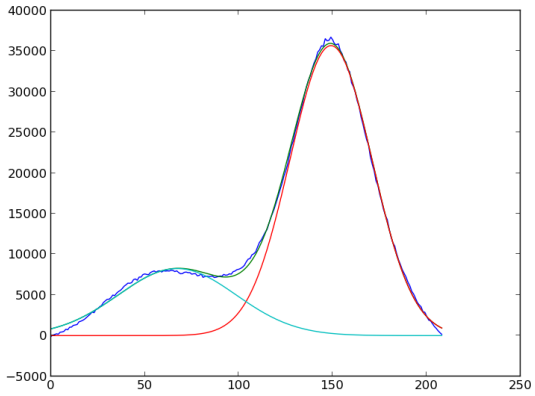

Figure 9: Gaussians fitted to the region of interest

3.6 Step 3: Using the Threshold Values

Using these thresholds, we were able to classify the decay regions in the data. The data was given to us in 2 forms - tiff images and the raw floating-point data. From looking at the tomograms of the tiff images it was clear that these were generated simply by taking a minimum and maximum floating point value to classify a region of interest and then dividing this into 255 equal bins. Any data outside of this region of interest (namely, any highly-absorbent particles and 'negatively' absorbant particles recorded due to the practice of classifying the absorbtion of the corrosive medium as being zero) were simply put into bin 255 or zero respectively. We did clarify this with a member of the group who produced the data, and they said that the maximum and minimum values were probably arbitarily selected. Therefore, the tiff data was fairly analgous to the raw data although with a lower resolution and with arbitary cut off points.
The tiff files did have the advantage of being very easy to recolour so that we could quickly see how well our threshold values classified the regions. This is because we could effectively treat the tiff data as being tomograms with 255 bins. When working with the raw data we first had to build a histogram from the data so that we could run our analysis, so we generally stuck to analysing the tiff data simply because it was faster and simpler. An example of a recoloured tiff slice from the tomogram is shown in figure 3.6. As you can see, the classification was fairly successful.

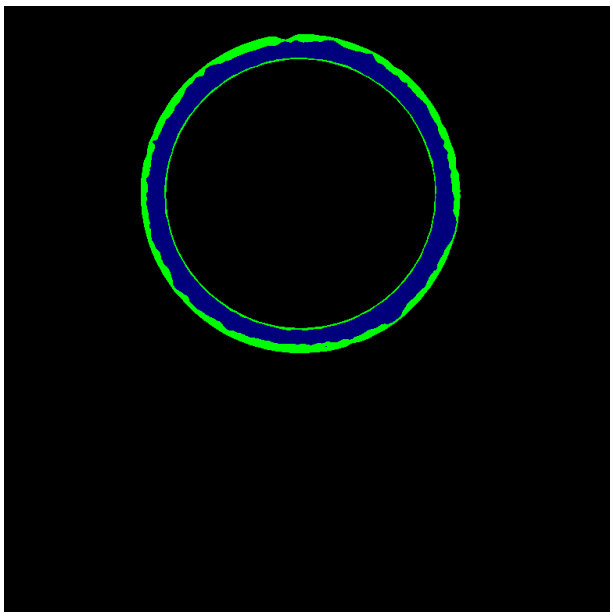

Figure 10: A recoloured slice from the tomogram showing the decay regions

Unfortunately, this method of working from the tiffs rather than the raw data meant that when we tried using the same method on the raw data it failed and we ran out of time before we could identify the bug. I believe that the problem lay in converting from the intersection value in the histogram back to a floating point value for use in classifying the data.

3.7 Next Steps

Given more time, we would have liked to use our classified tomographic data to evaluate how the decay regions evolve with time. We found that the magnesium tube actually moves between scans which makes things more difficult, although we did not establish wether it is merely an x-y translation or if the movement was more complex. Darie wrote a method to allign slices by minimising the dot product between them (more detail about his results can be found in his report), but to find out if the magnesium tube had moved up or down would be more complex. To do this, a number of highly-absorbing particles would need to be identified and it would have to be established how they have changed position between tomograms (they always remain in the same position in the tube).

4 Conclusion

In conclusion, we ran out of time with our main project. We wanted to deliver classified raw data but only managed to recolour the tiffs. We did provide a program for classifying the raw data given more robust values for the thresholds, and did provide a framework

for analysing the data in order to find the thresholds (but did not find a way to reliably automate this).
Despite this, I feel like I learnt a lot during my stay at DESY and am excited to use my new skills and confidence in working with computers. Indeed, during my time in Germany I was able to quickly use these skills on some personal projects - I used my experience in image processing to try using scientific techniques for artistic effect, have restarted an old project of mine (a computer game) in python instead of C++ and have experimented messing around with various web APIs. I think I would not have had the confidence or drive to carry out these projects if it had not been for the DESY studentship, and I will definitely be recommending the experience to my friends and others at my university!