Anthony Bisulco
Biomedical Imaging
Professor Fang
March 7th, 2018

Midterm Project

Introduction

The purpose of this midterm project was to simulate an x-ray projection imaging system. X-rays are one type of electromagnetic radiation used in medical imaging. These rays interact with different regions of the body and based off the attenuation in that area will reduce the signal strength of the original incident x-ray. A common technical term for this attenuation in a given local part of the body is the linear attenuation coefficient. The linear attenuation coefficient describes the amount of energy that is lost per unit of length. Therefore, a simple interaction model for x-rays with multiple different materials would just be a compounding of these different attenuation coefficients. These attenuation coefficients are then compounded and received by an x-ray detector which could be a Thin Film Detector or film.

The way that this model was used in this investigation was by given a voxelized imaging object, a program had to be written to develop the image given an object and its attenuation coefficients. The overall flow of this imaging scheme would entail first an x-ray source position, object position and detector position are placed. After that ray tracing can be performed in order to trace a given source position to the detector. During the ray tracing the scheme of an x-ray interaction and linear attenuation coefficient is performed. The given formula to find the exiting energy given the incident energy is $M_{out} = M_{in}e^{-\mu(i,jj,k)*L(i,j,k)}$ After compounding this formula multiple times and tracing the given ray from source to detector the resulting energy on the detector can be found.

Methodology

For this investigation the code was written in Python utilizing the libraries Numpy, Matplotlib, Numba and H5PY. The first step in performing this analysis was writing the given mat file containing a map of linear attenuation coefficients to HDF5 a cross platform data format that can be read into any language. This was achieved by using the h5write and h5create commands in Matlab and then using H5PY in order to read the data in as a Numpy Ndarray(Multidimensional array). After this the geometry of the image was setup by defining the various constants associated with the setup such as Mx, My, D, h, H, source end point and changing the normalization of the linear attenuation coefficient array. The values for the normalization constants of $\mu_{bone} = 0.573, \mu_{fat} = 0.193$were found in the textbook for the given configuration of a 50KeV x-ray source.

Imaging Geometry Constants	
Mx=128	# of x pixels
My=128	# of y pixels
D	pixel length
Nx=208	imaging volume length in x direction
Ny=256	imaging volume length in y direction
Nz=225	imaging volume length in z direction
H=2	distance from detector to bottom of imaging volume
H=h + Nz + 600	distance(Z) between detector and x-ray source

After setting up the necessary geometry for the setup, two for loops were setup in order to loop through all the x and y pixel combinations. For figuring out the dimension of each pixel

$dx = \frac{H*Nx}{(H-Nz-h)Mx}, dy = \frac{H*Ny}{(H-Nz-h)My}$ and $D = \max(dx, dy)$ [1]. At the start of every combination the detector pixel position in calculated as [orginOffset[0]+i*D,orginOffset[1]+D*j, 0] where i and j are loop counters for the x,y pixels, originOffset is the distance from (0,0) to the start of the detector and D is the pixel length. After that a direction vector can be calculated between the detector and source pixel by taking the difference between the source and detector position. Additionally, this position was normalized by dividing the above directly by the norm of the above direction.

After creating the necessary start position and direction vector, a while loop is performed until the current position is above the imaging z volume. First in the while loop the next position to move is calculated using the one_move_in_cube function which was rewritten in python. The function is pretty much the same as the Matlab version although with Numpy syntax. One caveat to this python version is divide by 0s are not handled properly, therefore an extremely small direction vector is initialized to any 0-direction component. This will not modify the algorithm since this small direction will always require the longest move in the function therefore it will never be updated.

Once the new position is calculated, this position is verified to be in the box or not. If it is not in the box, the while loop continues. If it is in the box, then an interaction model is used where the exited energy is calculated. One problem with the formula for this is$M_{in}e^{-\mu(i,jj,k)*L(i,j,k)}$an exponent can be a costly lookup. Therefore to speed up this process if we look at this equation over 2 iterations it is: $M_1 = e^{-\mu_1 x_1}, M_2 = e^{-\mu_1 x_1} e^{-\mu_2 x_2} = e^{\sum -\mu_i x_i}$. So this equation turns into a sum of the linear attenuation coefficients times the distance. Additionally, one thing to note in this example is that our distance x is in millimeters and the linear attenuation coefficient is in centimeters. Therefore, to account for this the equation turns into $e^{-10\sum \mu_i x_i}$. So, for each pixel the exponent is not calculated for the individual values but only after the entire sum has been performed. After the final value is calculated this is placed in the detector 2D array from which the image can be produced.

The image is produced by plotting the log intensity of the detector array. The reason for this is the values for pixels that don't interact with tissue and bone are very high and pixels that interact with tissue and bone are very low. Therefore, there is a disparity in values which can't be displayed properly on the plot. To resolve this problem the logarithm of the detector is taken in order to allow for one to visualize the contrast between the different imaging elements. This plot is produced by Matplotlib displaying a 2d plot of the image.

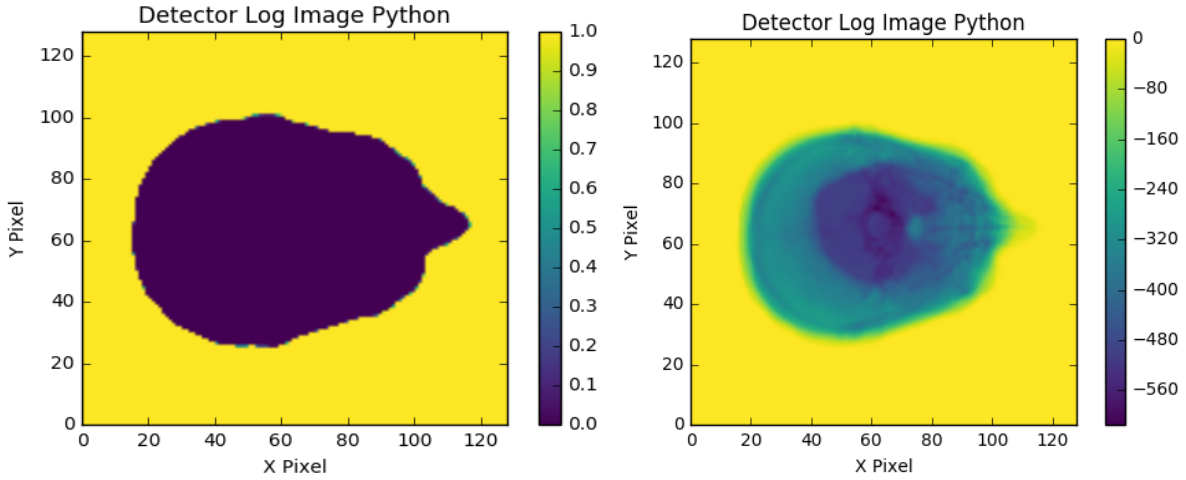
Profiling and Speed Up

After being able to verify that the output image was correct and produced a head projection image, the goal of this investigation was to speed up the code. To get a rough idea of how slow this Python version of code was first the line_profiler utility was used to benchmark the code. Upon first review of the code, the code ran in a total of 660 seconds, comparing this to another version of the code written in Matlab, which ran in 90s, this code was pretty slow. One of the first things to ensure fast operations were occurring was compiling the Numpy version used with Intel's Math Kernel Library. Instructions for this can be found here. After performing this optimization, the speed of the code was around 450 seconds.

To perform further optimization a highly efficient numerical computing library was used for Python known as Numba. Numba is a wrapper for LLVM to compile python code to machine code in a fast and efficient way. Numba will recognize different instruction that can be run in parallel and utilize Single Instruction Mutiple Data(SIMD) instructions. The first goal was to compile the one move in cube function to speed up this portion of the code which is hit the largest amount of times. To perform this compilation, a @jit decorator is added to the function which indicates when the python code is run that the LLVM complier will compile and optimize the code. There are limits to this complier for certain Numpy operations therefore, the simplest form of operations was utilized. Additionally, on this decorator some optional arguments are nopython mode, which tells the complier not to use slow python objects, but fast floating-point mathematics. Other decorators include, NOGIL which stands for no global interpreter lock which allows for python to parallelize operations and cache which will compile a function and keep the results for use during the next run. Hence, to properly time this operation the code must be run twice, once with the running time + compilation and another with just the code running time. After employing this strategy, the code ran with a running time of about 220s.

To further speed up the code the main loop of the code had to be further accelerated. Therefore a @jit decorator was placed around the main pixel loop in order to further speed up operations. Additionally, all math was forced to be performed on float 32 bits which is faster than doubles. After using this strategy, the running time went down to 9s. Finally, by removing unnecessary libraries and changing a few lines of the code It now is optimized to run in 1.65s. As of right now this code is running on a single core, as a version of this code was written for multiple cores but took over 28s to run due to parralelism overhead.

Results



The results of this investigation can be seen above in the skull figures generated. As stated previously there is a large disparity between the pixels that have interacted with the tissue and bone, thus the disparity between the logarithm and none logarithm image. These large number of ones in the image come from not interacting with any tissue. As for the internal structure of the image, the overall profile of a head image can be seen. One of the first interesting features that can be seen is the jaw in the image. Additionally, a large piece of bone can be seen in the

center of the image which seems to be where the jaw attaches. Finally, one other thing that can slightly be seen in the image is the water barrier within the brain. The barrier can be seen towards the extreme edges of the skull where there is a large intensity gradient difference.

Conclusion

The purpose of this investigation was to develop an image of a given object using an x-ray projection imaging setup in software. This study successfully completed this through the development of a custom Python program to perform the imaging of a given linear attenuation coefficient object matrix. Additionally, to speed up this code the Numba library was used to compile Python code and accelerate this code to run in 1.65 seconds(10.7ms standard deviation). The images produced from this accelerated code were found to be an image of a human skull where the jaw, water barrier of the brain and large piece of bone can be seen in the image. All in all, this investigation successfully completed the task.

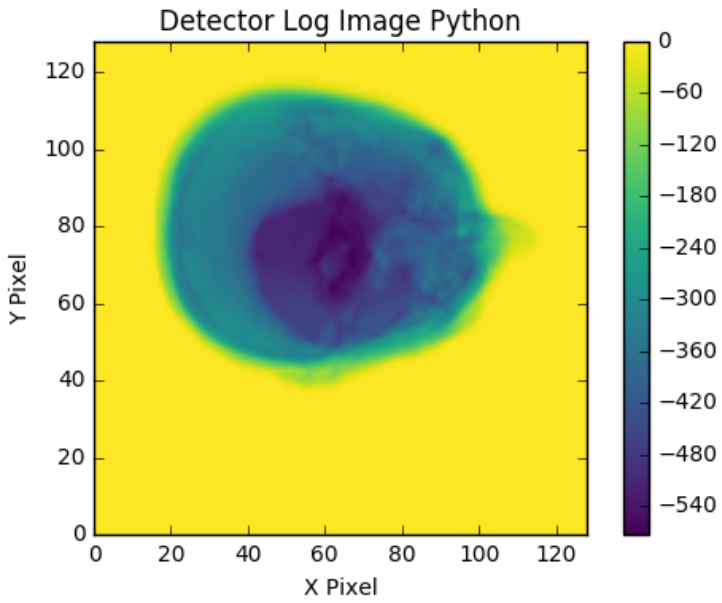
Final Run Time: 1.65s +-10.7ms across 7 runs, Ubuntu 16.04, MKL

```
%%timeit
%run bio.py
```

```
/usr/lib/python3/dist-packages/matplotlib/__init__.py:1352: UserWarn
because the backend has already been chosen;
matplotlib.use() must be called *before* pylab, matplotlib.pyplot,
or matplotlib.backends is imported for the first time.

  warnings.warn(_use_error_msg)
1.65 s ± 10.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Bonus(Moved source by 200mm)



Install Instructions
Tested code so far on Ubuntu 16.04, Mac OS 10.13.3 and Centos 7

-Clone my repository on Github https://github.com/anthonytec2/xrayscanner
Optionally just download repo:
https://github.com/anthonytec2/xrayscanner/archive/master.zip
-Install Python 3.6 here
-Install Pip here
-Download MKL on your machine here
-Compile Numpy with MKL here
Run these commands
-pip3 install numpy
-pip3 install numba
-pip3 install h5py
-pip3 install matplotlib
-Run python bio.py for running code

[1]$\alpha = length\ of\ middle\ to\ the\ end\ of\ one\ of\ the\ ray\ extremes$

$$\tan(\theta) = \frac{\frac{Nx}{2}}{H - Nz - h}$$

$$\tan(\theta) = \frac{\alpha}{H} => Htan(\theta) = \alpha$$

$$D_x = \frac{2Htan(\theta)}{Mx} = \frac{H*Nx}{(H-Nz-h)Mx}$$

Same analysis can be performed for dy