**COSC 4372**

**Virtual CT Scan Reconstruction**

Brayan Sosa - 2021182 - NSM - Undergraduate - [blsosa2@cougarnet.uh.edu](mailto:blsosa2@cougarnet.uh.edu)

Fatima Mirza - 2066039 - NSM - Undergraduate - [fmirza2@cougarnet.uh.edu](mailto:fmirza2@cougarnet.uh.edu)

Hatem Saleh - 8006437 - NSM - Undergraduate - [hhsaleh@cougarnet.uh.edu](mailto:hhsaleh@cougarnet.uh.edu)

Fiza Gubitra - 1985460 - NSM - Undergraduate - [fmgubitr@cougarnet.uh.edu](mailto:fmgubitr@cougarnet.uh.edu)

| ***Introduction*** |
| --- |

The goal of this project is to simulate the reconstruction of images normally generated by a CT scan machine. CT machines scan parts of the human body and display them on a screen as digital images. They operate by collecting little sections of a body and combining these sections to construct an image of the body. The algorithm built into this project will simulate the functionality of the CT scanner horizontally, vertically, and diagonally.

**Objectives:**

1. **Phantom Acquisition:** Create a virtual phantom (Shepp Logan) based on the dimensions specified by user input
2. **Projection Collection (Image Slicing):** Slice the phantom from Objective 1 and slice the image according to the orientation and number of projections that were taken as input (vertical, horizontal, or diagonal and an integer value from *1-phantom dimension* respectively).
3. **Image Reconstruction:** Use all the projection images that were acquired from Objective 2 to reconstruct the original phantom that was obtained from Objective 1.
4. **Image/Projection/Algorithm Analysis:** 
   1. Compare the results of the projection images with a small number of cuts vs a large number of cuts.
   2. Identify what makes our algorithm take the largest amount of time to function and what is wasting most of the time/space.

| ***Method*** |
| --- |

***Input / STDIN:***

1. Phantom dimensions
2. Orientation of the slices
3. A number of projection images

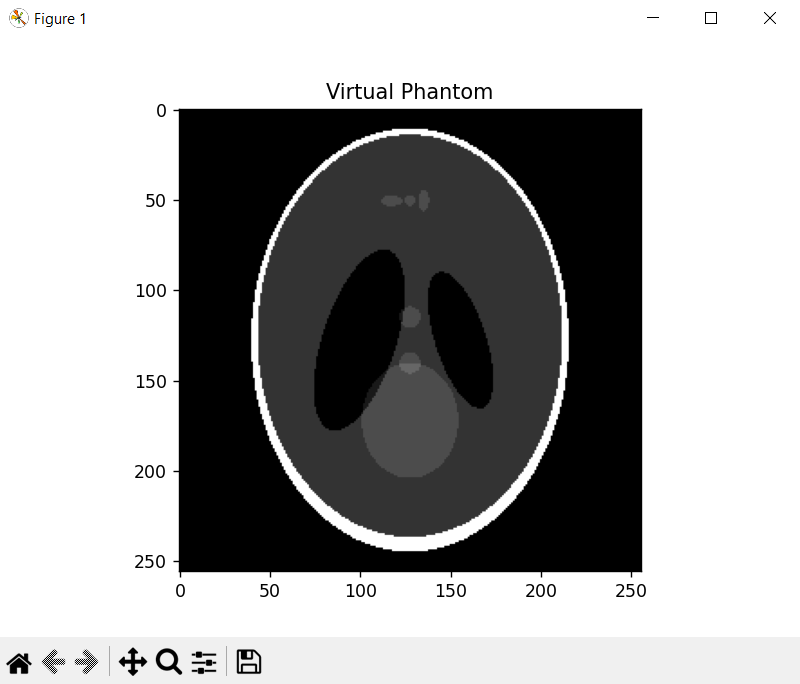
***Final Algorithm***

1. **Phantom Acquisition**
   1. Take in as input the dimensions of the phantom in the form of an integer, we will denote the **size of our phantom as “N”**.

Example input: 256 then the dimensions would be 256x256

* 1. After making sure the dimensions are large enough, use the input from step (a) to create a Shepp Logan phantom using the “shepp\_logan()” function from the phantominator library.
  2. Multiply all the values inside the matrix/phantom by 256 in order to be able to work with the NumPy library, since intensity values in NumPy range from 0-255.
  3. Display phantom using the display() function that was made with helper functions from the matplotlib library.
  4. Create a PNG file titled “phantom.png” for later comparison

Below is an image of what our phantom looks like in the display window:



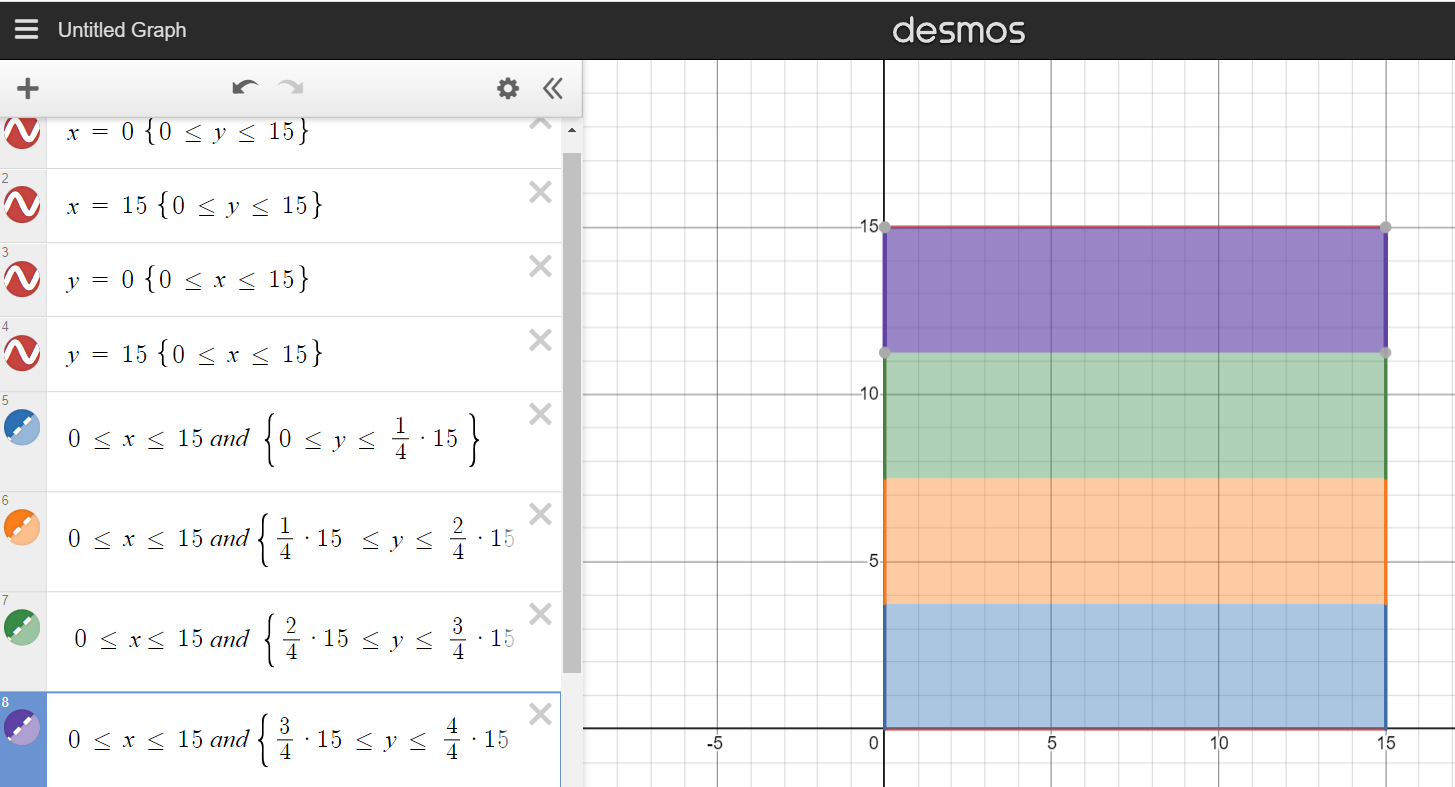
**Figure 1A:** **Virtual Phantom**

1. **Projection Collection (Image Slicing):**
   1. First take in as input the orientation at which the slices should be cut: horizontal, vertical, or diagonal. Receive as input the number of projection images the user would like to see. This is done to simulate the multiple scans a CT scanner does before it combines all the different fragments of the image.
   2. Horizontal Slicing:
      1. After receiving the orientation and the number of projection images from step (a), call the “imgCut()” function that we created. This function takes these parameters: a matrix - in our case **Figure 1A**, the “cutType” - the orientation of the slice (horizontal), the “slicedImageAmount” - the number of projection images the user would like to see, and “count” - an integer that will change ranging from 0 to N-1. Count essentially keeps track of how many slices have already been made.
      2. Inside the “imgCut()” function, create an array that will hold all of the pixels for a singular slice
      3. Check the value of count.

If count is 0, meaning that we have not made any slices to our image yet, we iterate from **0 to floor(width\*(1/slicedImageAmount))** and from **0 to height**. **Width** and **height** refer to the dimensions of the original phantom we are working with. The line **floor(width\*(1/slicedImageAmount))** will help determine the interval for the very first slice.

If count is greater than 0, we will iterate from **floor(width\*(count/slicedImageAmount)) to floor(width\*((count+1)/slicedImageAmount))** and from **0 to height**. This will determine the intervals for all the following slices since “count” is an iterator that ranges from 0 to N-1.

Here is an example image of how our iteration would look:



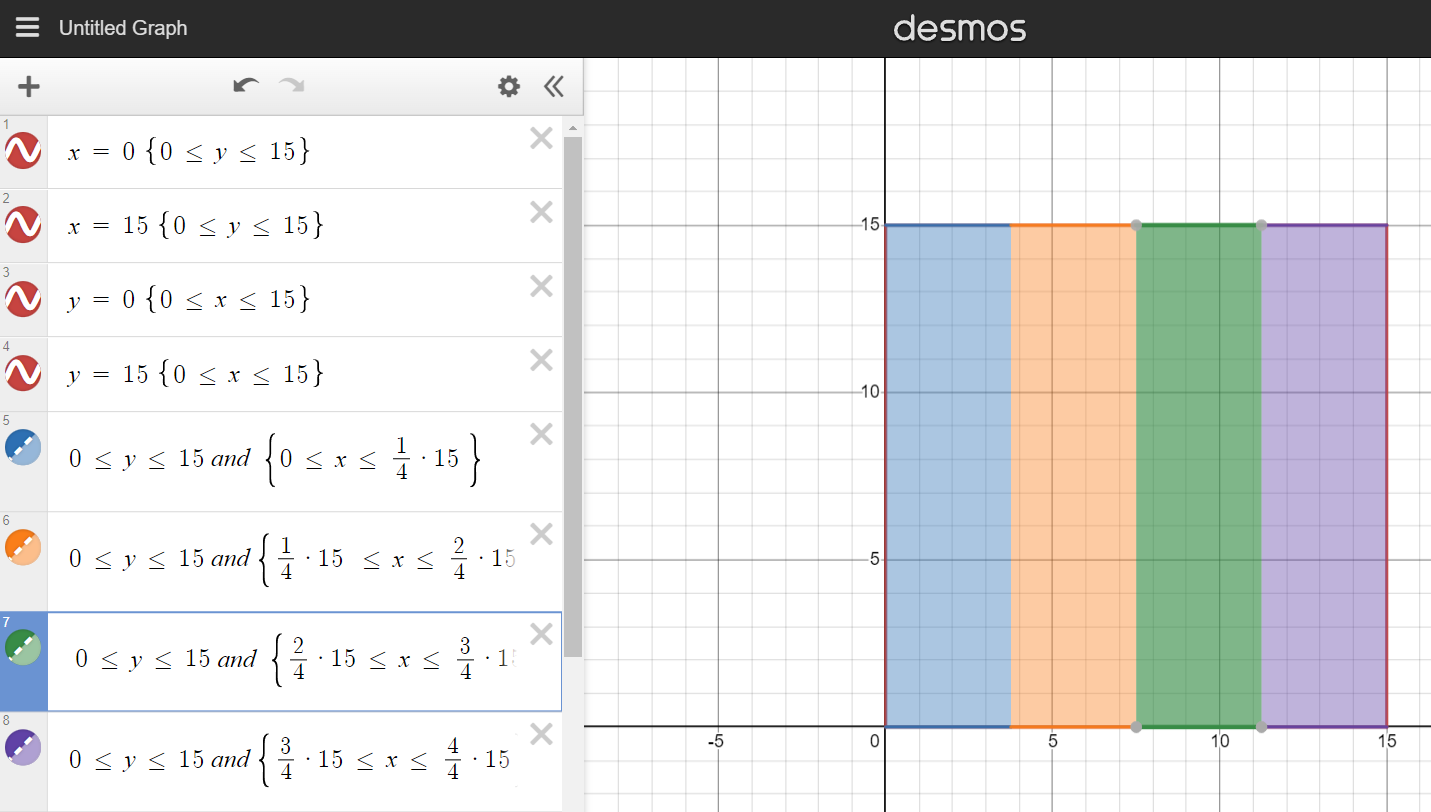
**Figure 2A: Horizontal Slicing**

* + 1. After determining the interval, iterate through the phantom at the given interval and append all those pixels to the array that was created in step (bii).
    2. Finally, compare the total amount of pixels found in the array from step (bii) to the formula **floor(height\*(width/slicedImageAmount))** which will result in the total amount of pixels for a sliced image. If the length of the array is bigger than the result of **floor(height\*(width/slicedImageAmount))**, then reshape the array into a matrix of dimensions **floor((width/slicedImageAmount)+1) x Height.** Otherwise, reshape the array to **floor(width/slicedImageAmount) x Height**
    3. Repeat steps (bi) to (bv) to obtain the remaining slices
  1. Vertical Slicing(follows the horizontal process with a few differences):
     1. After receiving the orientation and the number of projection images from step (a), call the “imgCut()” function that we created. This function takes these parameters: a matrix - in our case **Figure 1A**, the “cutType” - the orientation of the slice (vertical), the “slicedImageAmount” - the number of projection images the user would like to see, and “count” - an integer that will change ranging from 0 to N-1. Count essentially keeps track of how many slices have already been made.
     2. Inside the “imgCut()” function, create an array that will hold all of the pixels for a singular slice
     3. Check the value of count.

If count is 0, meaning that we have not made any slices to our image yet, iterate from **0 to width** and from **0 to (height\*(1/slicedImageAmount))**. **Width** and **height** refer to the dimensions of the original phantom that we are working with. The line **(height\*(1/slicedImageAmount))** will help determine the interval for the very first slice.

If count is greater than 0, iterate from **0 to width** and from **(height\*(count/slicedImageAmount)) to (height\*((count+1)/slicedImageAmount))**. This will determine the intervals for all the following slices since “count” is an iterator that ranges from 0 to N-1.

Here is an example image of how our iteration would look:



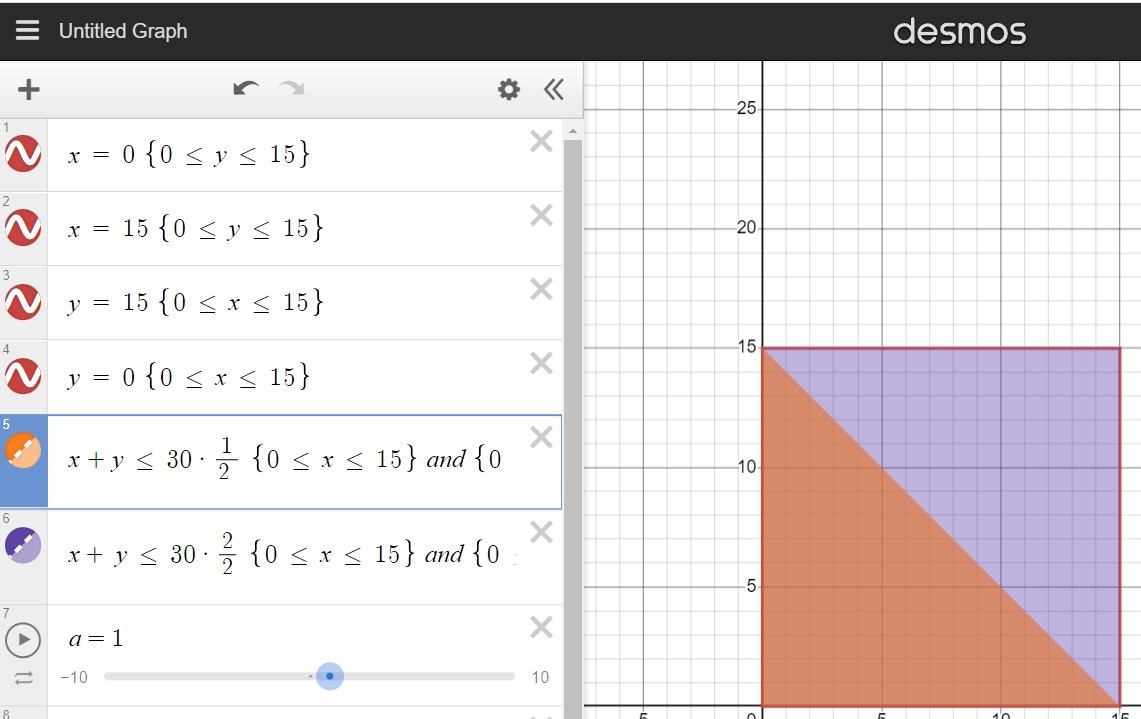
**Figure 2B: Vertical Slicing**

* + 1. After determining the interval, iterate through the phantom at the given interval and append all those pixels to the array created in step (cii).
    2. Finally, compare the total amount of pixels found in the array from step (cii) to the formula **floor(width\*(height/slicedImageAmount))** which will result in the total amount of pixels for a projection image.

If the length of the array is bigger than the result of **floor(width\*(height/slicedImageAmount))**, reshape the array into a matrix of dimensions **Width x (floor((height/slicedImageAmount)+1))**  Otherwise, reshape the array to **Widthx(floor(height/slicedImageAmount))**

* + 1. Repeat steps (ci) to (cv) to obtain the remaining slices
  1. Diagonal Slicing
     1. After receiving the orientation and the number of projection images from step (a), call the “imgCut()” function that we created. This function takes these parameters: a matrix - in our case **Figure 1A**, the “cutType” - the orientation of the slice (diagonal), the “slicedImageAmount” - the number of projection images the user would like to see, and “count” - an integer that will change ranging from 0 to N-1. Count essentially keeps track of how many slices have already been made.
     2. Inside the “imgCut()” function, create a matrix with the same dimensions as the phantom given that we cannot make an image with the exact dimensions of a slice.
     3. Iterate from 0-N for the rows and from 0-N for the columns. During this iteration, pixels will only be added to the matrix made in step (dii) if they fit the criteria from the if statements
     4. Inside the iteration, check whether **the count is 0,** meaning that no slices have been made. If that is the case, then the pixels will have to pass the if statement with **x + y <= (row+height)\*(1/slicedImageAmount)**. This works the same as inequalities would in a graph, so we are able to effectively create a diagonal cut. When the count is 0, we get the top left diagonal cut. If the count is not 0, however, the, if statement has its criteria, change to **(row+height)\*((count+ 1)/slicedImageAmount) <= x + y <=** **(row+height)\*(count/slicedImageAmount)** which would get the next slice.

Here is an example of how it looks:



**Figure 2C: Diagonal Slicing**

* + 1. Repeat steps (di) to (div) to receive the remaining slices

1. **Image Reconstruction:**
   1. After receiving all necessary inputs, begin the image reconstruction process by making sure we have enough slices and that the amount of slices is possible taking into account the size of the phantom.
   2. Check whether we are performing a horizontal, diagonal, or vertical slicing
   3. Horizontal Reconstruction:
      1. Call the “imageCut()” function in order to properly get a slice so we can append it to the “result” matrix (across axis 0 which represents its x-axis). This will display all of the slices after they have been put together. Since this is considered the first slice, increase the count by one.
      2. Iterate inside a for loop ranging from **1 to the amount of projection images.** Repeat step (ci) for all the remaining images and increase count by 1 so we can properly get the intervals for each slice that we will then put back together.
      3. After all intervals have been put together and all PNG slices have been created, display the resulting phantom which should look the same as **Figure 1A.**
   4. Vertical Reconstruction (follows the horizontal process with a single difference):
      1. Call the “imageCut()” function in order to properly get a slice so that we can append it to the “result” matrix (across axis 1 which represents its y-axis). This will display all of the slices after they have been put together. Since this is considered the first slice, increase count by one.
      2. Iterate inside a for loop ranging from **1 to the amount of projection images.** Repeat step (ci) for all the remaining images and increase count by 1 so that we can properly get the intervals for each slice that we will then put back together.
      3. After all intervals have been put together and all PNG slices have been created, display the resulting phantom which should look the exact same as **Figure 1A.**
   5. Diagonal Reconstruction:
      1. Diagonal reconstruction is the most different reconstruction process out of the three and it begins by creating a matrix called “result” with the dimensions of the original phantom.
      2. Call the “imageCut()” function which will return a slice (projection). For us, that is the same size as the original image
      3. Iterate through the projection matrix and if any pixel has an **intensity value that isn’t zero** (0 represents empty space), set the pixel of the result matrix at said location equal to the pixel of our projection matrix, i.e **result[i, j] = slice[i, j]**. After completing this, increase the count by 1.
      4. Repeat step (ciii) for all remaining partitions until all of them have been attained
      5. After all partitions have been “combined”, display the resulting phantom which should look the exact same as **Figure 1A.**
2. **Image/Projection/Algorithm Analysis:** 
   1. \*\*This step is discussed in the “Results and Discussion” section of this report\*\*

| ***Results and Discussion*** |
| --- |

After testing the code with multiple different sizes of the phantom, orientations, and number of images, we encountered some flaws with our algorithm. We determined that performing diagonal cuts are the most computationally demanding of all of the three orientations. Not only does it waste the most amount of space to create slices that take up the same amount of space as the original phantom (most of which will be empty space), but it also takes the most amount of time since we iterate through the entire size of the image for as many image slices the user would like to see. The space issue is not fully a problem in our algorithm for horizontal and vertical cuts. Given that they are rectangular cuts, we are able to resize these slices as thin as possible. This can be better seen in **Figure 2D1-3 and Figure 2E1-3** as opposed to **Figure 2F1-3** which is triple in size of the previous slices. A way to fix the space issue in our algorithm would be to display the images using matplot, but we believe it would be better to permanently have the images as PNGs in case later viewing is desired.

Another issue we ran into was related to the number of image slices desired by the user. A value that was too high would lead to wasted time and space. For example, in a 256 x 256, if a diagonal cut and a total of 5 projection images are desired, we would waste time for two of those slices since they would just be empty space.

Here is an image to better display this issue:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |

**Figure 3A**

There are two completely dark images that are completely unnecessary. Although, if we tried to fix it by avoiding slices that are majority dark spaces, we would cause another problem since we could be skipping over a vital part of our image. Despite all of the flaws that our diagonal cut algorithm has, we still believe that it could perform better than a lot of algorithms that manually manipulate the diagonal movement.

The final issue with our project is that the bigger the Shepp Logan phantom is and the more slices that are desired, the slower it gets. However, this is the case for many other algorithms that have the same functionality as this one. It is inevitable that things will take longer when there is more information to be processed.

Despite all of our shortcomings, our algorithm still runs very well and has a minimal overlap issue, unlike other algorithms. Our algorithm will only overlap for as many slices that we create. Here is an example of our code slicing a 256x256 phantom in all three different orientations for a total of three projection images and the combined result for all three of them:

| **Horizontal** | **Vertical** | **Diagonal** |
| --- | --- | --- |
| **Figure 3B.1** | **Figure 3C.1** | **Figure 3D.1** |
| **Figure 3B.2** | **Figure 3C.2** | **Figure 3D.2** |
| **Figure 3B.3** | **Figure 3C.3** | **Figure 3D.3** |
| **Figure 3E - Combined Result** | | |

| ***Conclusion*** |
| --- |

Despite all of the difficulties and shortcomings, this algorithm is still able to successfully create a phantom with dimensions given by user input and successfully slice and reconstruct images like a CT scan. It still unfortunately has the same issue that the majority of CT scans have: run-time. For our case, we also have the issue of space complexity for creating PNG files for the user to be able to compare all the sliced images at all times and the black projection images shown in **Figure 3A** of this report. However, we were still able to successfully complete our objectives of creating a phantom the size that the user desires, as well as slicing and reconstructing the phantom in its desired orientations and analyzing the images/algorithm.

| ***Literature*** |
| --- |

*CT Colonography*. Radiologyinfo.org. (2020, June 20). Retrieved October 20, 2022,

from https://www.radiologyinfo.org/en/info/ct\_colo

Mayo Foundation for Medical Education and Research. (2021, July 23). *Virtual*

*colonoscopy*. Mayo Clinic. Retrieved October 20, 2022, from https://www

.mayoclinic.org/tests-procedures/virtual-colonoscopy/about/pac-20385156

Referenced the Python\_SheppLogan python file on Teams

| ***Code*** |
| --- |

**\*\* We didn’t include the code here since there are multiple files and many functions - please access the code using this link →** <https://github.com/BrayanSosa04/CT-Scan-Reconstruction-Algo.git>

\*\*The instruction file on how to operate our code is titled “README.md”\*\*

\*\*The main code file is titled “main.py”\*\*

\*\*Example output images (for 6 projection images) for all orientations are in the output folder\*\*

\*\*All of the functions can be found under “Functions” folder\*\*