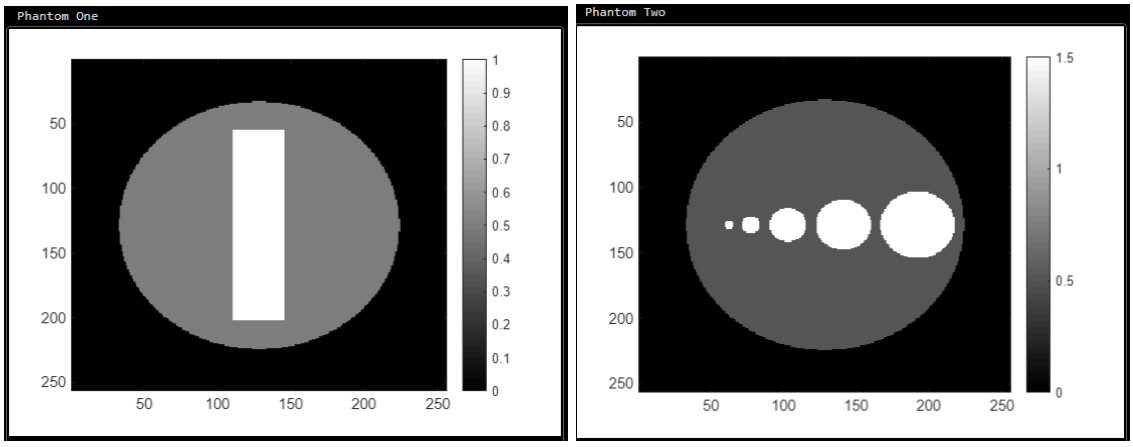
Virtual MRI Scanner Report

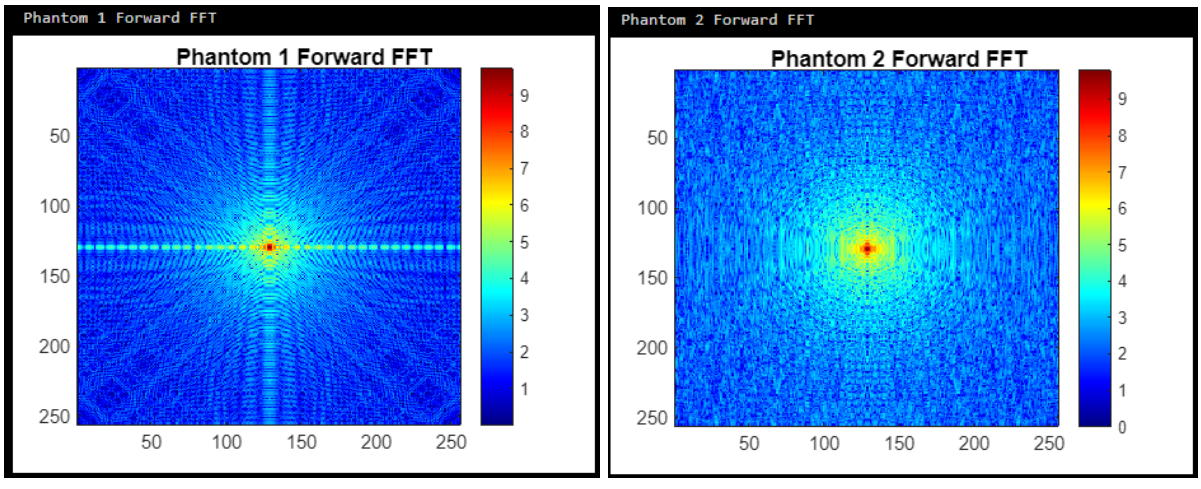
Introduction:

The goal of this project was to create software to simulate the data acquisition of a virtual MRI scanner via a GUI. Two different acquisition trajectories were implemented: cartesian and radial. The techniques used to implement these acquisition trajectories will be mentioned below. Of course, for a scanner to operate, there should be something to scan. To that end, we first allow the user to enter a size, *N*. With this, we create two different validation phantoms of size *N* in which our scanner uses as input (shown below).

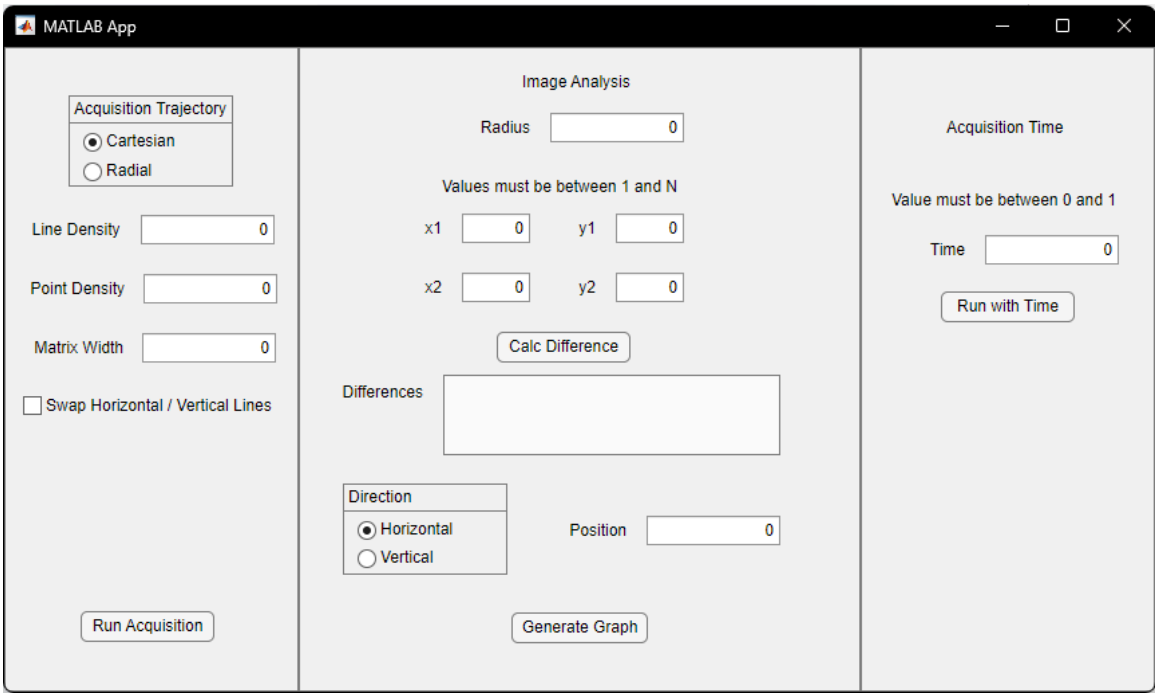


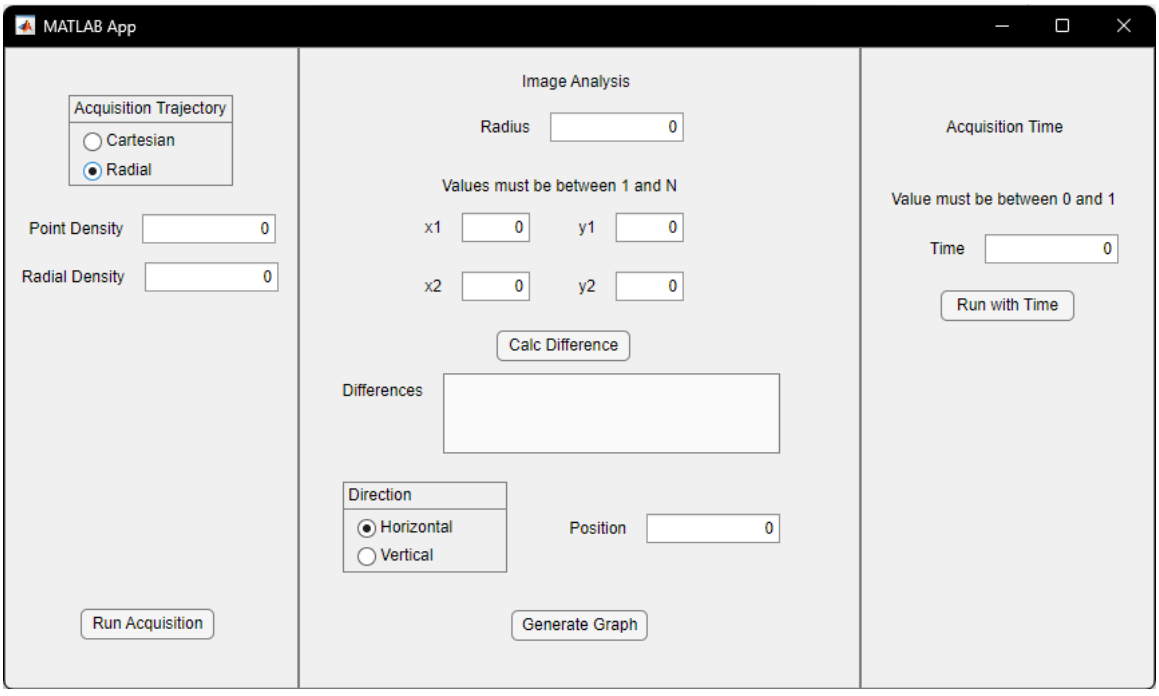
Process:

MRI data collection occurs in the frequency-domain (or k-space). For this reason, the first step in data acquisition for our virtual scanner involves converting the phantom image data (real) into k-space via FFT. In our case, we adopted the use of MATLAB's built in fft2() function. Also, since the real and k-space are Fourier pairs, we used the ifft2() function to invert the image back into the real. Shown below are the above phantom's k-spaces:



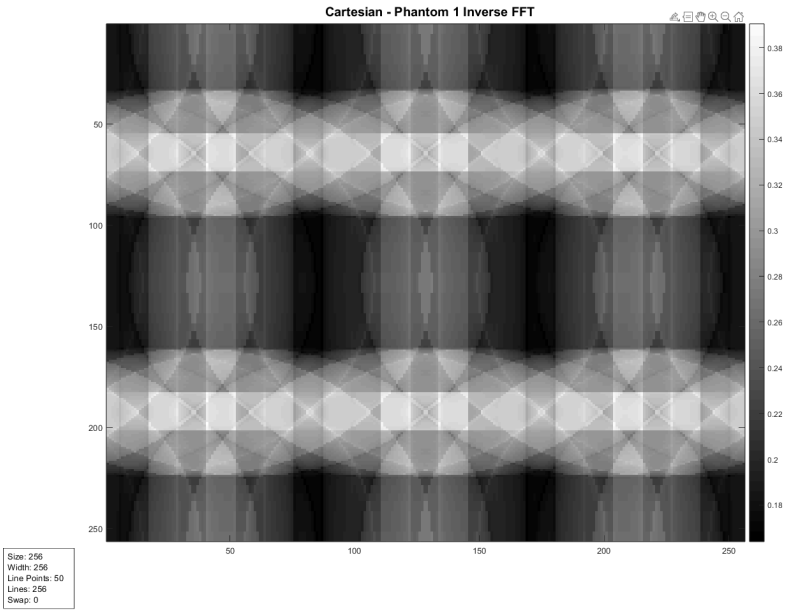
Following the generation of these k-spaces, we allow the user to select the acquisition trajectory of either *Cartesian* or *Radial*. Once selected, a set of appropriate options show to allow the user to enter additional parameters:

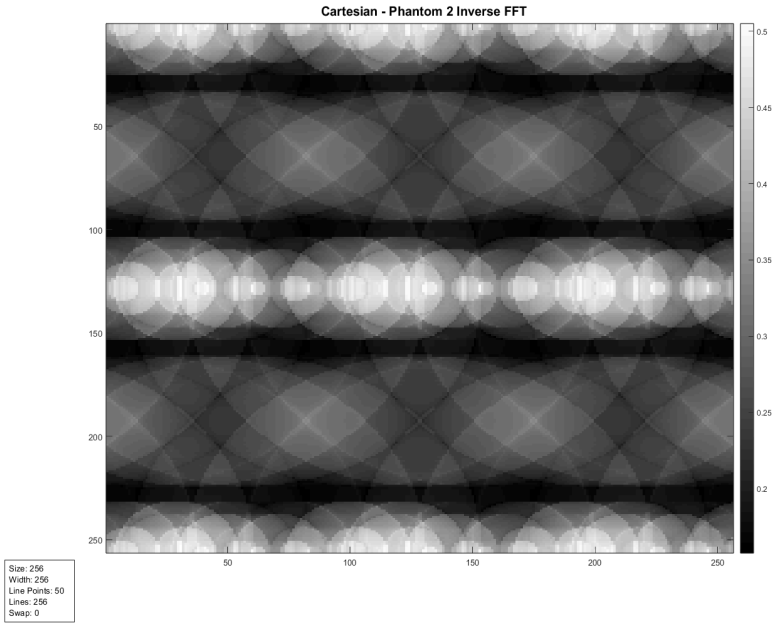




Cartesian Acquisition Trajectory:

For the cartesian route, we first create an empty matrix *reconstruction* of the size of the input matrix (the k-space of the current phantom). We scale the point and line densities according to the *acquisitionTime* to reflect how time would affect the quality of an image. With these values, we find how much space should be between each line (rows divided by line density) and how much space should be between each point (columns divided by point density). If the checkbox for swapping horizontal with vertical lines, the end effect would be a 90 degree rotation of the image. We then iterate through all points for the lines and set the value of the point in *reconstruction* to the value in the original matrix. Below is an example output:





Radial Acquisition Trajectory:

The idea behind the radial acquisition trajectory is similar, though implemented differently. Where in cartesian we went line by line left to right from top to bottom, the radial method involves dividing the image in *lines* radial lines having *linePoints* points. That is each line is at an angle of 2π / *lines.* We iterate through each line and calculate the angle in which the current line would be from the start. Then for all the points we calculate the coordinates of the point on the radial line and ensure the coordinate is within grid boundaries. If so, we set the point in *reconstruction* to the value of the point of the original matrix. Below is an example output:

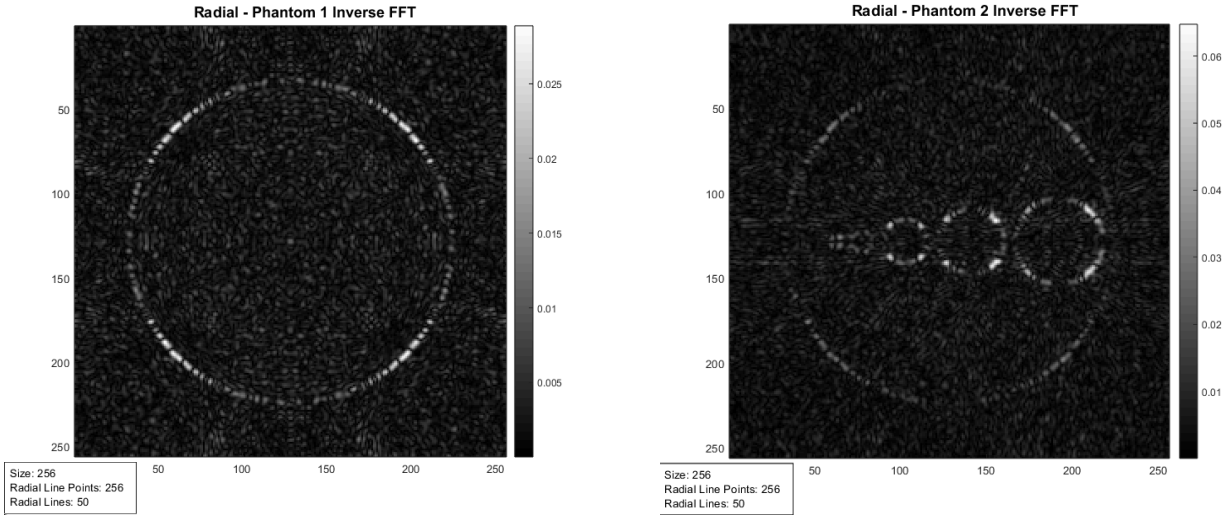


Image Analysis:

With the now generated final images, we can conduct some analysis comparing our original two phantoms with our final output versions. Firstly, we examine how the signal intensity and contrast compares. We do so by identifying two circular regions, centered at coordinates (x_1, y_1) and (x_2, y_2) both with radius r (coordinates and radius entered via the GUI).

For each region, we calculate the region's average signal intensity and take the difference of the two values to give us a numerical difference value. Shown below is example data (the top section showing the difference values for the original phantom, the cartesian output, then the radial output. Below is the same but for the second phantom):

```
testDifferenceOne          = signalIntensityDifference(matrixOne, radius, [x1, y1], [x2, y2])
testDifferenceOneCartesian = signalIntensityDifference(postFFTOneCartesian, radius, [x1, y1], [x2, y2])
testDifferenceOneRadial    = signalIntensityDifference(postFFTOneRadial, radius, [x1, y1], [x2, y2])

testDifferenceTwo          = signalIntensityDifference(matrixTwo, radius, [x1, y1], [x2, y2])
testDifferenceTwoCartesian = signalIntensityDifference(postFFTTwoCartesian, radius, [x1, y1], [x2, y2])
testDifferenceTwoRadial    = signalIntensityDifference(postFFTTwoRadial, radius, [x1, y1], [x2, y2])
```

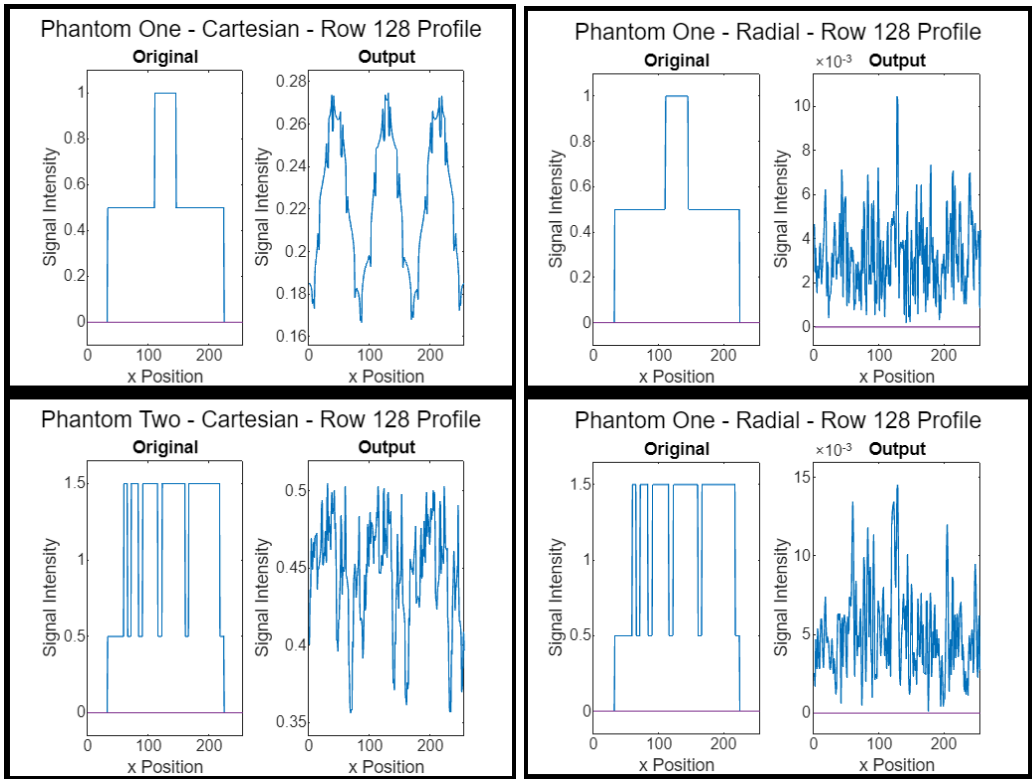
```
testDifferenceOne = 0.6747
testDifferenceOneCartesian = 0.0277
testDifferenceOneRadial = 3.9009e-04

testDifferenceTwo = 0.6517
testDifferenceTwoCartesian = 0.0106
testDifferenceTwoRadial = 0.0020
```

Signal Intensity Profiles:

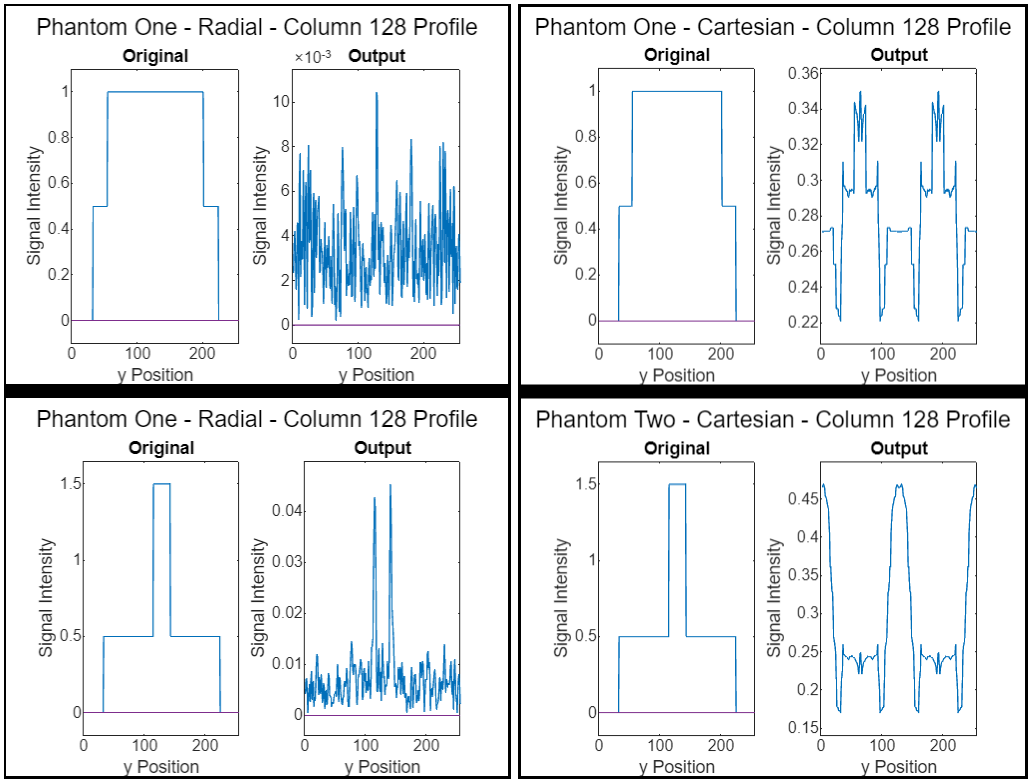
In order to generate a signal intensity profile we must be given a target matrix, a *direction* (horizontal or vertical) and a *position* (indicating which row or column to "travel along"). The target matrices would be the original and output images. If horizontal was chosen, for each matrix, we would begin at the *position* row and continue along that row gathering all signal intensities (stored in an array for later use in a plot).

Of course, since our final output images were "complex" arrays, we had to first convert the complex number into the magnitude of the signal intensity using abs(). Also, we keep track of the min and max signal intensity values to later affect our plot such that we can see the profile better. With this, we end with an array of signal intensities and we plot the profiles of the two matrices side by side for ease of comparison. Below are example profile comparisons:

Direction: horizontal



Direction: vertical



Methods:

To further describe the above process in greater depth (at the method level) we have this "methods" section. In total, the project utilizes 10 custom created functions. Here, we will dive into each so that reproducibility can be extended to audiences of this document.

*matrix = **generatePhantoms**(size, ellipses, addRectangle):*

```
% Generates a phantom image
function matrix = generatePhantoms(size, ellipses, addRectangle)
    matrix = phantom(ellipses, size);

    % Adds a rectangle to the matrix by defining a rectangular region in
    % which the signal intensity for pixels in this region is set appropriately
    if addRectangle == true
        [x, y] = deal(size);
        midX = x / 2;
        deltaX = x / 15;
        leftXBoundary = midX - deltaX;
        rightXBoundary= midX + deltaX;

        midY = y / 2;
        deltaY = y / 3.5;
        topYBoundary = midY - deltaY;
        bottomYBoundary= midY + deltaY;

        for i=1:x
            for j=1:y
                % Ensure current coordinate is within the rectangle's boundaries
                if (j >= leftXBoundary) && (j <= rightXBoundary) && (i >= topYBoundary) && (i <= bottomYBoundary)
                    matrix(i, j) = 1;
                end
            end
        end
    end
    %figure, imshow(matrix)
    figure;
    imagesc(matrix);
    colormap("gray");
    colorbar;
end
```

We begin with ***generatePhantoms()***. This function generates a phantom matrix given the desired *size(int)* and the *ellipses(2D array)* found within the phantom. Additionally, we have *addRectangle(boolean)* to add in a rectangular region within the center of the matrix. We have this due to our requirements stating phantom one to have such a region, and creating a rectangle out of ellipses would have been too cumbersome.

In the function we make a function call to the MATLAB native ***phantom**(ellipses, size)* method. This gives us our phantom matrix, and for function calls in which *addRectangle* is set to false, we simply show the figure. Else, we begin the process of programmatically adding a rectangular region. First, we define [x,y] to be the dimensions of the matrix and get the mid coordinates for the matrix. Then we define a deltaX and deltaY to be how long and tall our rectangle to be. With the deltas, we find the 4 boundaries of the rectangle (midX $\pm$ deltaX and midY $\pm$ deltaY). We then iterate through our size by size matrix and for all points within this

boundary, we set the signal intensity to 1. Following this, we exit the nested loop, display the matrix, and return the modified phantom.

With this function defined, we can create our phantoms to later be virtually scanned.

*kspace = **fftTransform**(matrix, direction, text, parameters):*

This function generates and shows the k-space of a given phantom (*matrix)*, given the *direction (*"forward" or "inverse" to use either fft2() or ifft2()). The other parameters: *text,* and *parameters* are used for setting the title of the k-space image, and showing the user used parameters within a legend box, respectively.

We first of course check the direction of the FFT as it is vital to properly apply *kspace*= fft2() for "forward" and *kspace* = ifft2() for "inverse". When going in the forward direction, we set *imageArray = log(1 + abs(fftshift(kspace)));* as only setting it to *abs(kspace)* shows little, if any, information in the produced kspace image. You'll notice the size of the parameters array is dynamic as the inputs for cartesian or radial acquisition trajectories are different. As such, the corresponding legend boxes will also be different, shown below. We finally output the image.

This method will now allow us to transform our above phantoms into kspace to then apply our acquisition trajectories to, and to invert back into the real once done so.

```
% Generate and show the k-space of a phantom
function kspace = fftTransform(matrix, direction, text,  parameters)
    legendText = "";
    if direction == "forward"
        color = "jet";
        fprintf(text);
        kspace = fft2(matrix);
        imageArray = log(1 + abs(fftshift(kspace)));
    else
        color = "gray";
        fprintf(text);
        kspace = ifft2(matrix);
        imageArray = abs(kspace);
    end

    % Cartesian
    if (length(parameters) == 5)
        compartment1 = sprintf("Size: %d", parameters(1));
        compartment2 = sprintf("Width: %d", parameters(2));
        compartment3 = sprintf("Line Points: %d", parameters(3));
        compartment4 = sprintf("Lines: %d", parameters(4));
        compartment5 = sprintf("Swap: %d", parameters(5));
        legendText = compartment1 + newline + compartment2 + newline + compartment3 + newline + compartment4 + newline + compartment5;

    % Radial
    elseif (length(parameters) == 3)
        compartment1 = sprintf("Size: %d", parameters(1));
        compartment2 = sprintf("Radial Line Points: %d", parameters(2));
        compartment3 = sprintf("Radial Lines: %d", parameters(3));
        legendText = compartment1 + newline + compartment2 + newline + compartment3;
    end

    figure;
    sgtitle(text, 'FontWeight', 'bold');
    imagesc(imageArray);
    if (legendText ~= "")
        annotation('textbox', [.005 .1 0 0],'String', legendText, 'FitBoxToText','on');
    end
    colormap(color);
    colorbar;
end
```

reconstruction = ***cartesianSample****(matrix, width, pointDensity, lineDensity, swap, acquisitionTime):*

The ***cartesianSample***() function takes as parameters: the kspace matrix of a given phantom, followed by all of the user defined inputs. Its goal is to use the cartesian acquisition trajectory to sample the kspace and create a new reconstruction matrix based on its scans. We scale the point and line densities according to acquisition time to mirror the effects time has on the quality of an image. Swap switches the image around to treat x as y and y as x when scanning through *matrix*. In either case, we find the rowSpacing and pointSpacing by dividing rows and cols by the appropriate parameter. In the swap=false case, row spacing is the number of rows in the image divided by how many lines we scan with. Similarly, point spacing is the number of columns in the image divided by how many points we will have on a line. We then iterate for all of our width via a step of rowSpacing and iterate through all of our width via a step of pointSpacing. We find our current point and set the point in our initially empty reconstruction matrix the value of the point on our kspace matrix. In the end, we return our reconstruction matrix. This matrix then is used in a call to our previous ***fftTransform()*** function in the inverse direction to produce our final phantom output.

```
% Cartesian Acquisition Trajectory which returns final image matrix given user parameters
function reconstruction = cartesianSample(matrix, width, pointDensity, lineDensity, swap, acquisitionTime)
    [rows,cols] = size(matrix);
    reconstruction = zeros(rows, cols);

    % Scale the point and line density based on acquisition time
    pointDensity = round(pointDensity * acquisitionTime);
    lineDensity = round(lineDensity * acquisitionTime);

    if swap == true
        rowSpacing   = ceil(cols / pointDensity);
        pointSpacing = ceil(rows / lineDensity);
    else
        rowSpacing   = ceil(rows / lineDensity);
        pointSpacing = ceil(cols / pointDensity);
    end

    for i=1:rowSpacing:width
        for j=1:pointSpacing:width

            if swap == true
                point = matrix(j,i);
            else
                point = matrix(i,j);
            end
            reconstruction(i, j) = point;
        end
    end
end
```

reconstruction = ***radialSample***(kspace, numRadialLines, numPointsPerLine, acquisitionTime):

The ***radialSample()*** also takes in the kspace matrix of a given phantom and all the user defined inputs. This time, since we aren't going line by line as in the cartesian method, but by radial lines about the origin, we first define a local origin centered at (centerX, centerY). We begin again by scaling our line and point densities by our acquisition time and define an angleIncrement to be (2π / *numRadialLines).* With these defined, we loop through all *numRadialLines* and set our current angle appropriately. Once on a line, we loop through all points on the line and calculate the x and y values for that point. After we ensure we are still within boundaries, we add that data to our reconstruction matrix. Finally, we return *reconstruction* to also be used in the ***fftTransform()*** method in the inverse direction.

```
% Radial Acquisition Trajectory which returns final image matrix given
function reconstruction = radialSample(kspace, numRadialLines, numPointsPerLine, acquisitionTime)
    [rows, cols] = size(kspace);
    centerX = rows / 2;
    centerY = cols / 2;
    reconstruction = zeros(rows, cols);

    % Scale the radial lines and points per line based on acquisition time
    numRadialLines = round(numRadialLines * acquisitionTime);
    numPointsPerLine = round(numPointsPerLine * acquisitionTime);

    % Calculate the angle between each radial line
    angleIncrement = 2 * pi / numRadialLines;

    for n = 0:numRadialLines-1
        angle = n * angleIncrement;
        for p = 1:numPointsPerLine
            % Calculate the coordinates of the point on the radial line
            x = round(centerX + (p - numPointsPerLine/2) * cos(angle));
            y = round(centerY + (p - numPointsPerLine/2) * sin(angle));

            % Ensure the coordinates are within the grid boundaries
            if x > 0 && x <= rows && y > 0 && y <= cols
                reconstruction(x, y) = kspace(x, y);
            end
        end
    end
end
```

*result = **isOutOfBounds**(point, size):*

This function is one of two helper functions that assist in calculating the average signal intensity of a given region. Its purpose is to ensure a point does not escape the boundaries of a matrix bounded by [1, size].

```
% Signal Intensity and Contrast
% Return if point is out of matrix boundary
function result = isOutOfBounds(point, size)
    x = point(1);
    y = point(2);

    result = (x < 1) || (y < 1) || (x > size) || (y > size);
end
```

*result = **isOutOfRegion**(center, point, radius):*

This second helper function is also used to detect when a point is out of a boundary. In this case, the boundary is a circle centered at point *center* with radius *radius*. If the distance between the point and the center is greater than the radius, then the point is out of bounds.

```
% Return if point is out of region boundary
function result = isOutOfRegion(center, point, radius)
    x = center(1);
    y = center(2);

    x1 = point(1);
    y1 = point(2);

    distance = sqrt((x - x1)^2 + (y - y1)^2);
    result = distance > radius;
end
```

average = ***calculateAverageSignalIntensityOfRegion****(matrix, radius, position):*

This method calculates the average signal intensity for a region in *matrix* centered *position* with radius *radius*. We first define a box region centered at *position* so that we may later loop through a nested loop. We define left, right, too, and bottom boundaries to ensure we later do not extend past in our loop. We keep a running total for our sum (which adds up the signal intensities for all points in the region). With all that setup, we enter our nested loop and for all points within our boundary and within our region, we add to our sum and increment our count. Following the loops' completion, we calculate the average and return it.

```
% To calculate SI difference of two regions, we first get each region's average SI
function average = calculateAverageSignalIntensityOfRegion(matrix, radius, position)
    dimensions = size(matrix);
    length = dimensions(1);
    x = position(1);
    y = position(2);

    % Box containing region
    leftBoundary   = x - radius;
    rightBoundary  = x + radius;
    topBoundary    = y - radius;
    bottomBoundary = y + radius;

    % Running sum to calculate average
    sum = 0;
    count = 0;

    % Get SI for all points within the boundary to later calculate the region's average SI
    for i=leftBoundary:rightBoundary
        for j=topBoundary:bottomBoundary
            point = [i, j];
            % Ensure point is within matrix and region boundaries
            if (isOutOfRegion(position, point, radius) || isOutOfBounds(point, length))
                continue;
            end

            % Increment sum and count to average later (uses abs() to get
            % magnitude of signal intensity, especially for when the matrix
            % has complex numbers
            sum = sum + abs(matrix(i, j));
            count = count + 1;
        end
    end

    average = sum / count;
end
```

difference = ***signalIntensityDifference****(matrix, radius, positionOne, positionTwo):*

This function takes in a singular *matrix* and the center of two circle regions: *positionOne* and *positionTwo* both with radius *radius.* It uses our previous function to get the average signal intensity for each region and returns the difference between the two.

```
% Returns the difference of SI of two regions of given radius for a given matrix
function difference = signalIntensityDifference(matrix, radius, positionOne, positionTwo)
    averageOfRegionOne = calculateAverageSignalIntensityOfRegion(matrix, radius, positionOne);
    averageOfRegionTwo = calculateAverageSignalIntensityOfRegion(matrix, radius, positionTwo);

    % Calculate difference
    if averageOfRegionOne > averageOfRegionTwo
        difference = averageOfRegionOne - averageOfRegionTwo;
    else
        difference = averageOfRegionTwo - averageOfRegionOne;
    end
end
```

[signalIntensities, minSI, maxSI, xLabel, titleText] = ***getSignalIntensities****(matrix, direction, position):*

In order to generate a signal intensity profile for a *matrix* in a given *direction* starting at *position* we create an empty array of signalIntensities and initialize minSI and maxSI so that our plot can be more easily readable / all data captured within it. The idea for filling the array is that we begin at row *position* (if selecting horizontal *direction*) and iterate through all of its columns. At each point, we get the SI and append it to our array [the process is identical for the vertical *direction* except we start at column *position* and iterate in the vertical *direction*]. We set min/maxSI as we iterate. Then, to avoid the case in which minSI and maxSI are equal, we decrease minSI by 1. We return a lot of variables in this function as they are used in the next function to be plotted.

```
% Returns an signal intensity array and other similar data to be plotted for a given matrix
function [signalIntensities, minSI, maxSI, xLabel, titleText] = getSignalIntensities(matrix, direction, position)
    [x, y] = size(matrix);
    signalIntensities = zeros(x);
    minSI = inf;
    maxSI = -inf;

    if (direction == "horizontal")
        xLabel = "x Position";
        titleText = "Row " + position + " Profile";

        for i=1:x
            signalIntensity = abs(matrix(position, i));
            signalIntensities(i) = signalIntensity;

            % Set min/maxSI accordingly
            if (signalIntensity < minSI)
                minSI = signalIntensity;
            end
            if (signalIntensity > maxSI)
                maxSI = signalIntensity;
            end
        end
    elseif (direction == "vertical")
        xLabel = "y Position";
        titleText = "Column " + position + " Profile";

        for i=1:y
            signalIntensity = abs(matrix(i, position));
            signalIntensities(i) = signalIntensity;

            % Set min/maxSI accordingly
            if (signalIntensity < minSI)
                minSI = signalIntensity;
            end
            if (signalIntensity > maxSI)
                maxSI = signalIntensity;
            end
        end
    end

    range = maxSI - minSI;
    minSI = minSI - (range * .1);
    maxSI = maxSI + (range * .1);
    if minSI == maxSI
        minSI = maxSI - 1;
    end
end
```

signalIntensityVsPositionGraph*(matrixOne, matrixTwo, direction, position, type):*

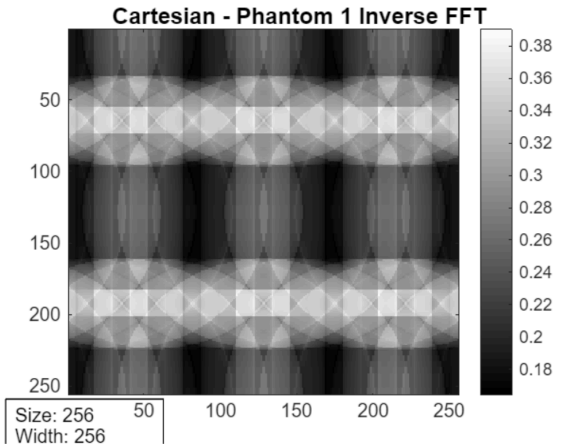
Here, we have two matrices: the first being the original unmodified phantom and the second our output phantom (post acquisition trajectory and inverse FFT). We call ***getSignalIntensities()*** for both and using the data from these calls, we simply plot our results side by side.

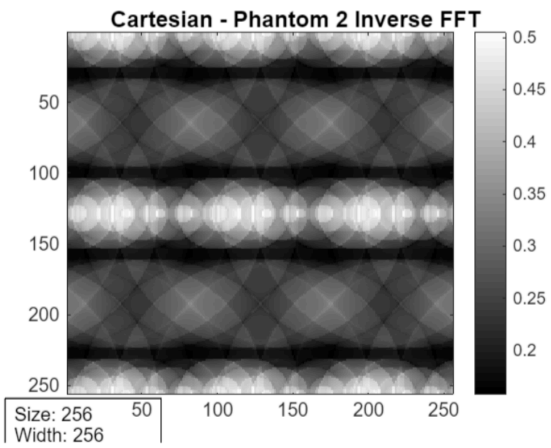
```
% Generates a pair of plots plotting SI vs Position
function signalIntensityVsPositionGraph(matrixOne, matrixTwo, direction, position, type)
    [x1, y1] = size(matrixOne);
    [x2, y2] = size(matrixTwo);

    [signalIntensitiesOne, minSIOne, maxSIOne, xLabelOne, titleTextOne] = getSignalIntensities(matrixOne, direction, position);
    [signalIntensitiesTwo, minSITwo, maxSITwo, xLabelTwo, titleTextTwo] = getSignalIntensities(matrixTwo, direction, position);

    % Plot figure
    figure;
    sgtitle(type + " - " + titleTextOne);
    subplot(1, 2, 1); plot(signalIntensitiesOne); xlabel(xLabelOne); ylabel("Signal Intensity"); ylim([minSIOne maxSIOne]); xlim([0 x1]); title("Original");
    subplot(1, 2, 2); plot(signalIntensitiesTwo); xlabel(xLabelTwo); ylabel("Signal Intensity"); ylim([minSITwo maxSITwo]); xlim([0 x2]); title("Output");
end
```

Test/Validation Phantom: Changing Parameters

Cartesian Acquisition:

Original Phantom using code:



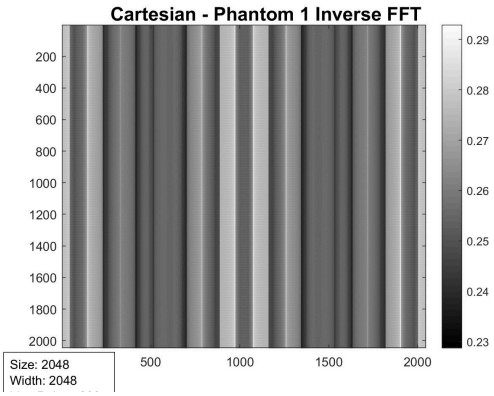


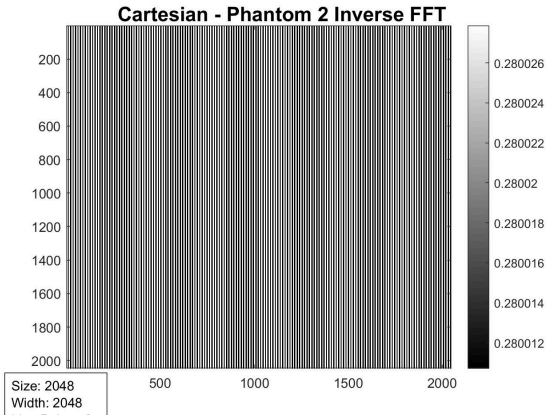
Using the GUI, we can adjust the parameters to observe and analyze their impact on the generated images, as shown below.

N = 256

Line Density = 2

Point Density = 2

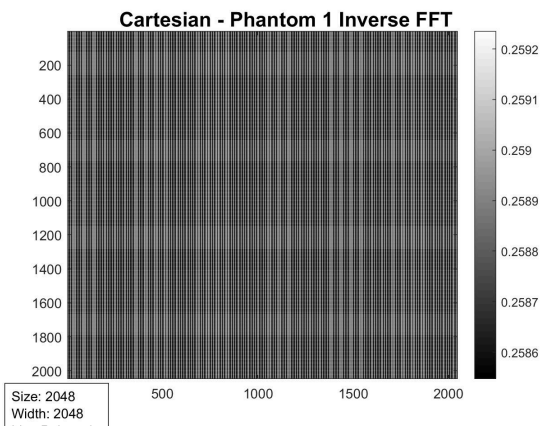


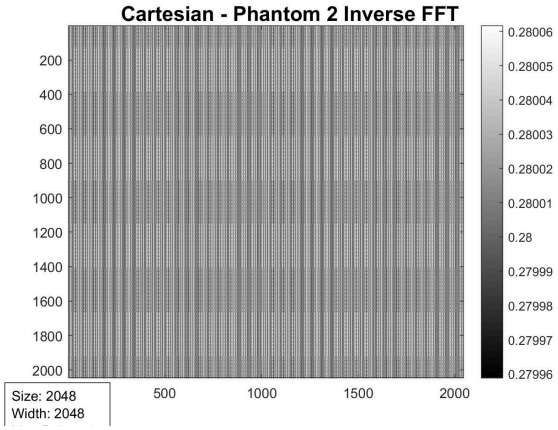


N = 256

Line Density = 4

Point Density = 4

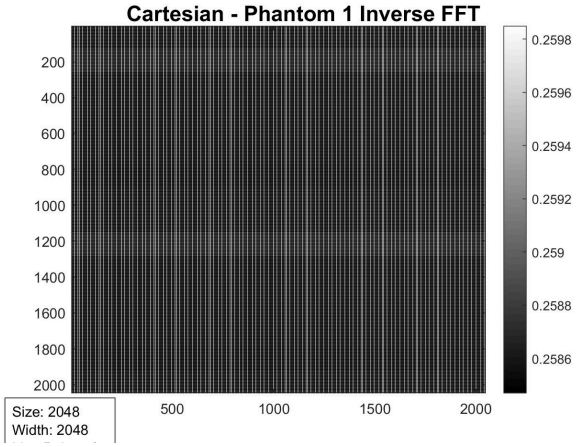


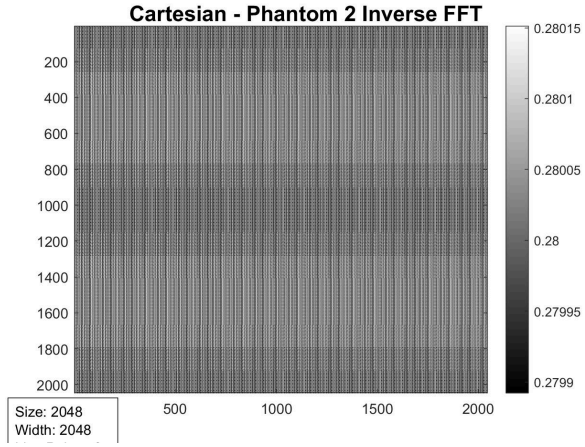


N = 256

Line Density = 8

Point Density = 8

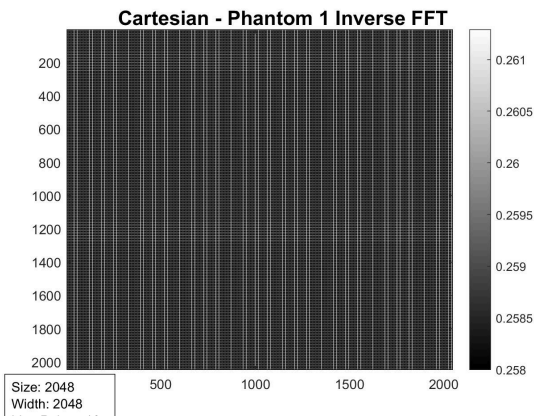


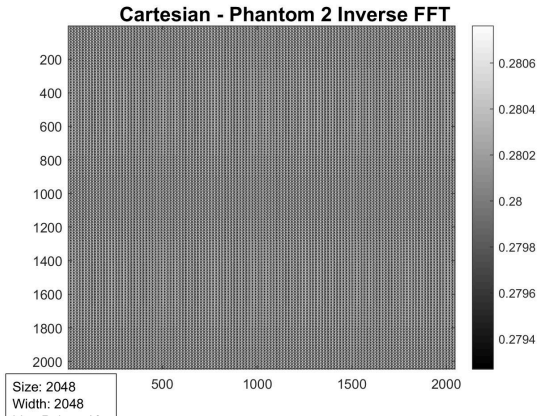


N = 256

Line Density = 16

Point Density = 16



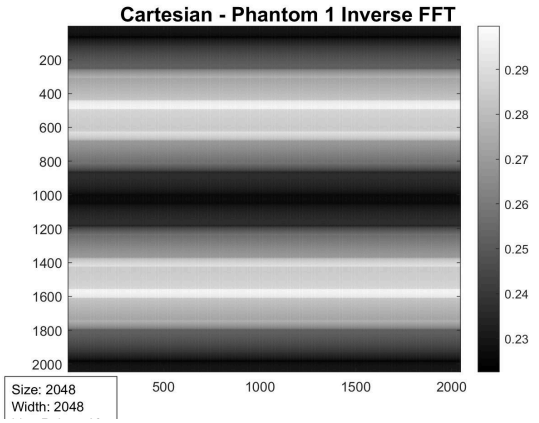


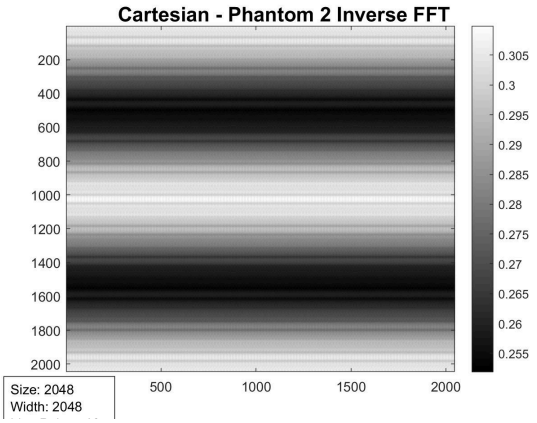
When increasing the parameters, having one greater than the other will have the following effects.

N = 256

Line Density = 200

Point Density = 16



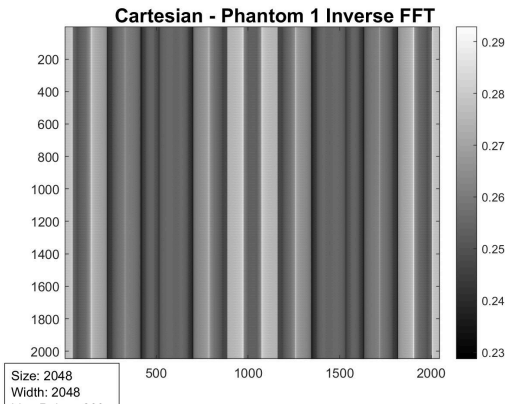


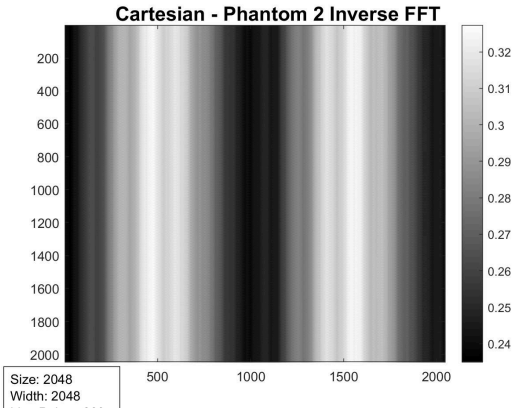
Increasing the line density to 200 and setting the point density to 16 enhances the structural detail in the image, but the resolution may still appear limited due to the lower point density per line.

N = 256

Line Density = 16

Point Density = 200





With a line density of 16 and a point density of 200, the image shows smoother details along individual lines, but the overall structure may appear undersampled due to the lower line density.

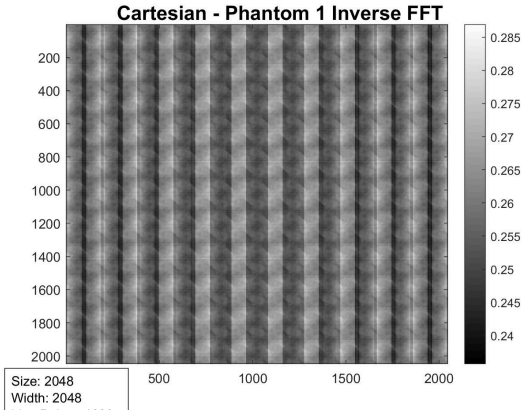
Now we will observe an output with the same parameters and the inclusion of acquisition time.

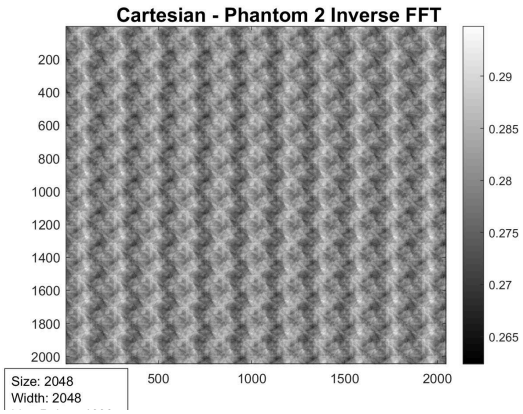
N = 256

Line Density = 1000

Point Density = 1000

Acquisition time = 0.1



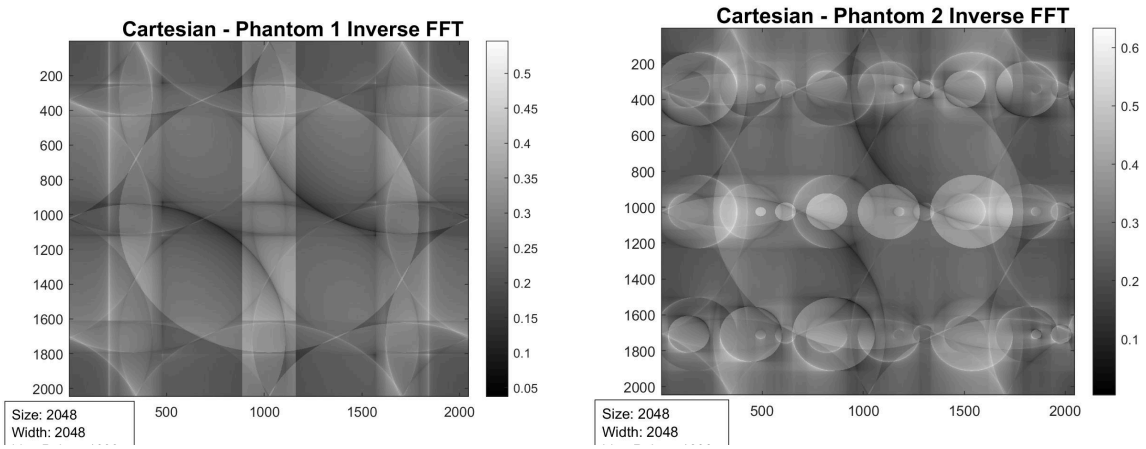


N = 256

Line Density = 1000

Point Density = 1000

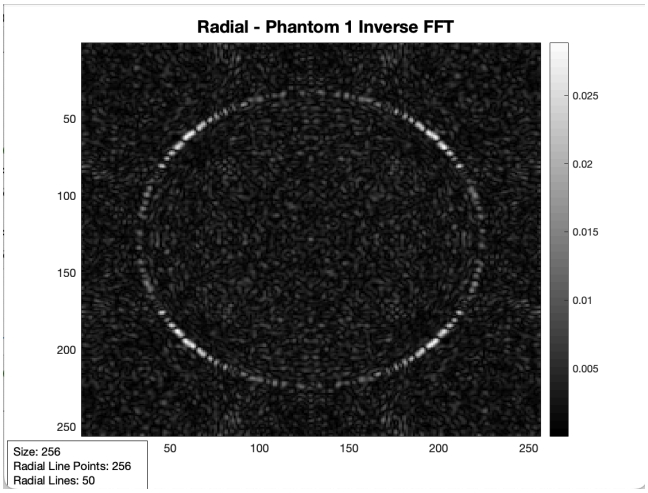
Acquisition time = 0.9

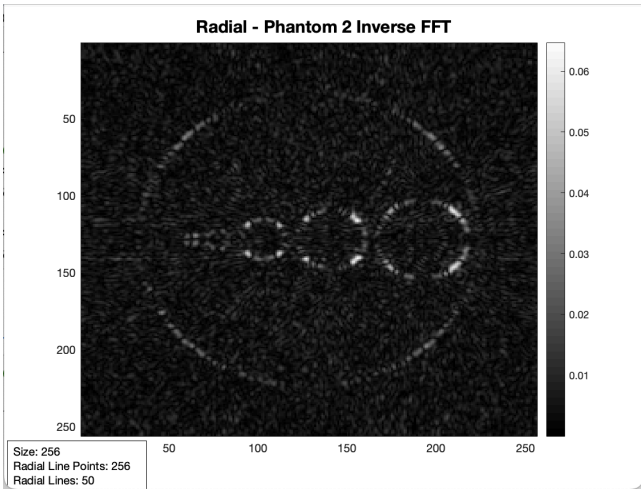


When comparing the images, it becomes evident that as the acquisition time approaches 1, the clarity of the generated image improves significantly.

Radial Acquisition:

Original Phantom using code:

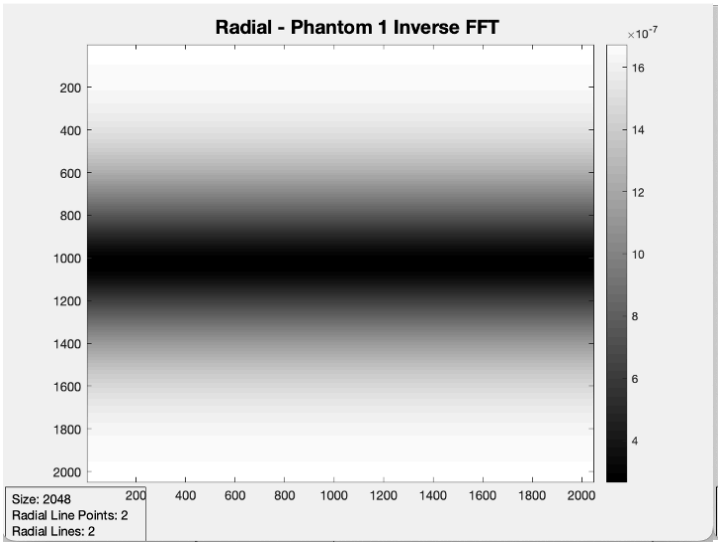


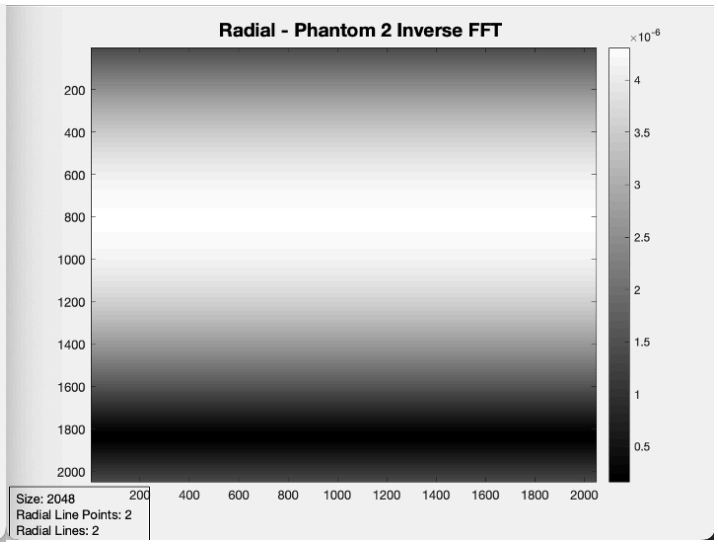


Using the GUI, we are able to change the parameters and identify how they affect the generated images. View the following images and their respective parameters.

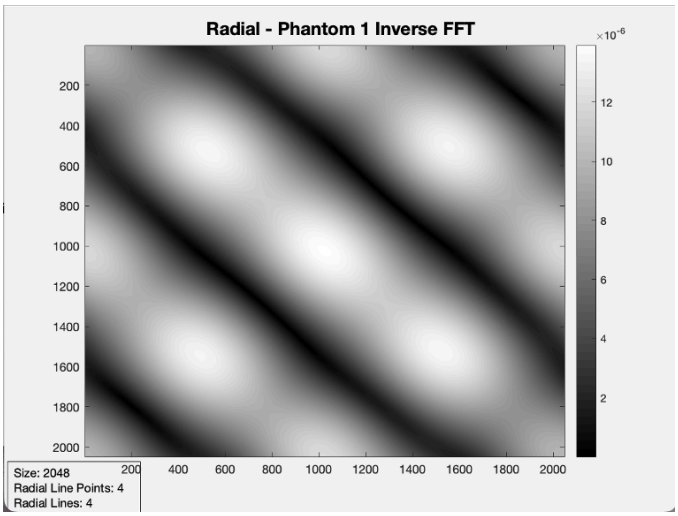
N=2048

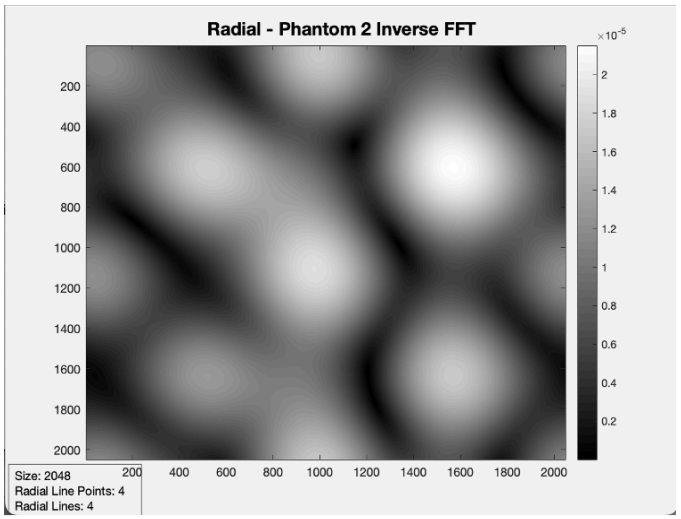
Point Density = 2
Radial Density = 2



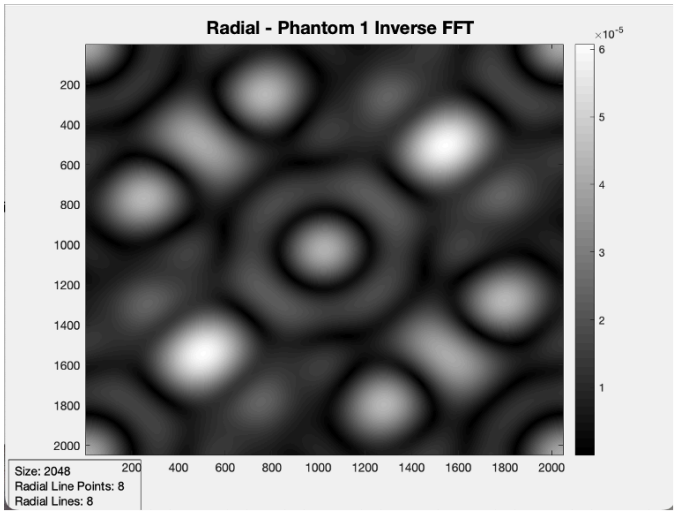


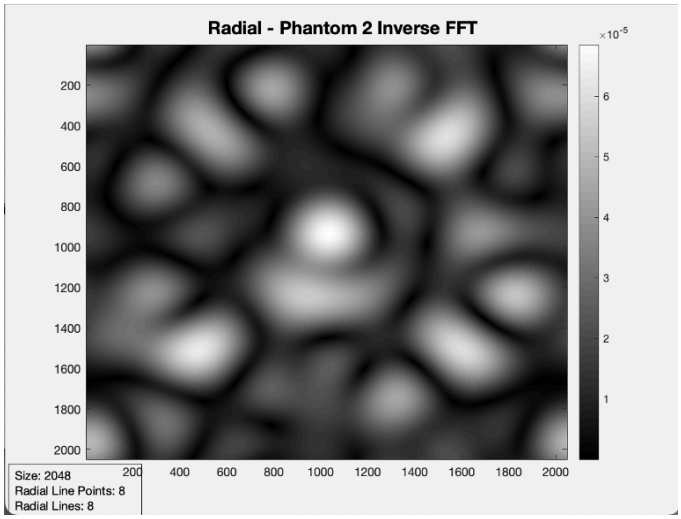
Point Density = 4
Radial Density = 4



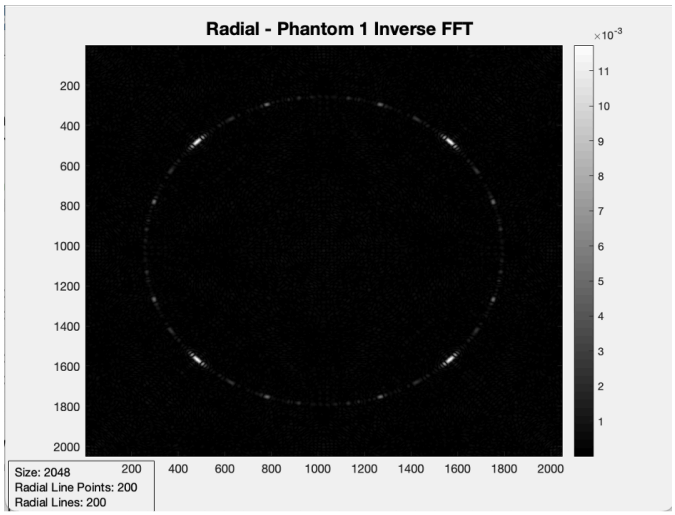


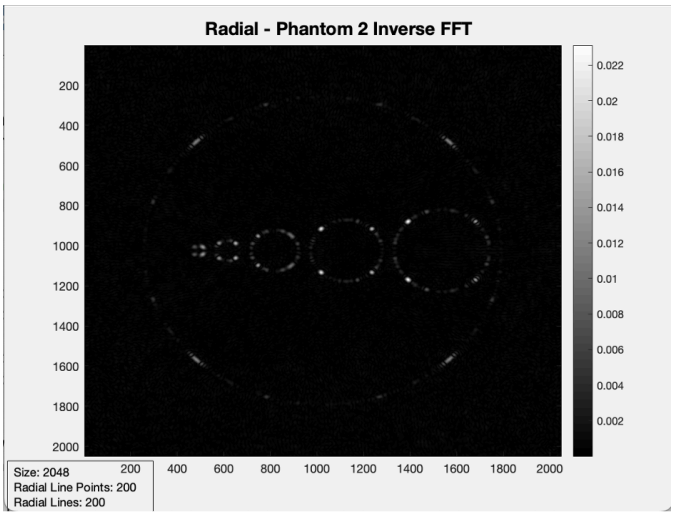
Point Density = 8
Radial Density = 8





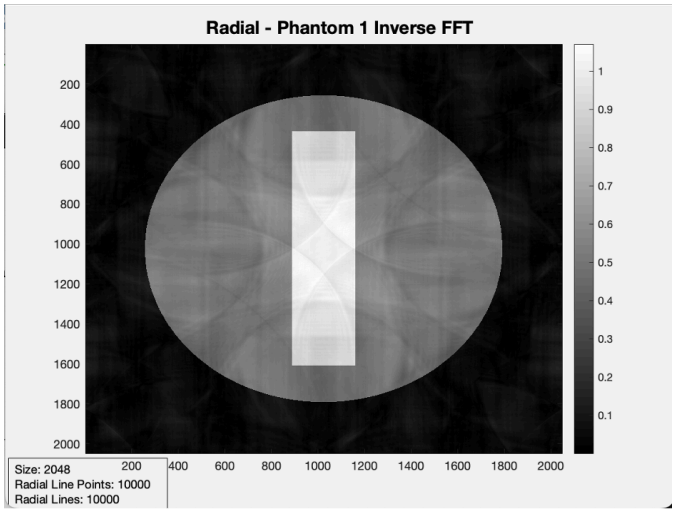
Point Density = 16
Radial Density = 16

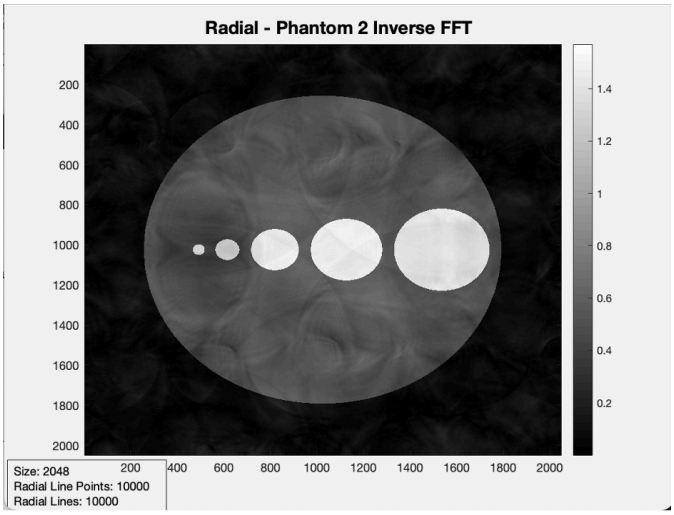




When increasing the values in both parameters, we see that the quality of the image becomes better. If you were to raise the value to a high enough number, at one point the image becomes very clear, for example:

Point Density = 10000
Radial Density = 10000

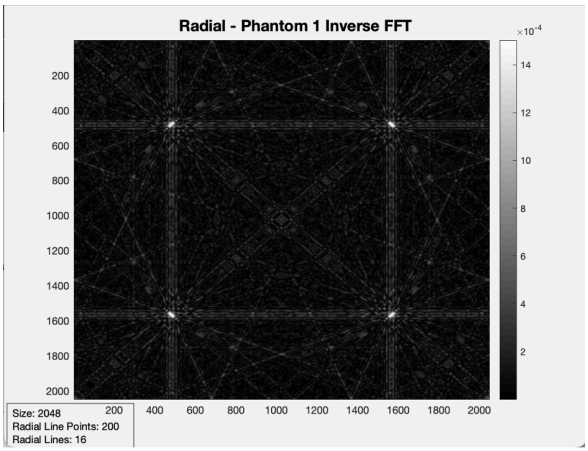


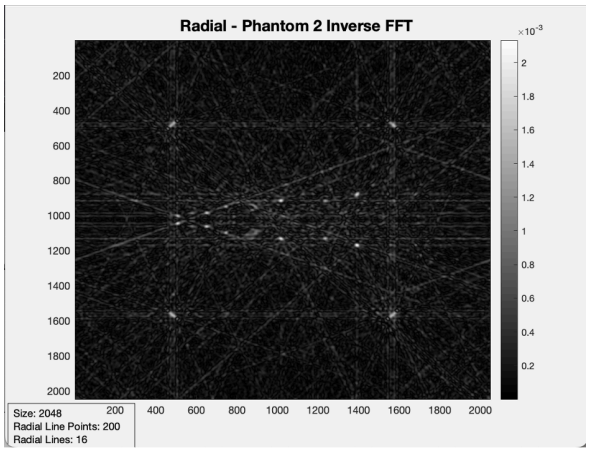


As viewed above, increasing both parameters will give you the best possible image.

When increasing the parameters, having one greater than the other, you will see what effect each field has.
For example, when using the two presets:

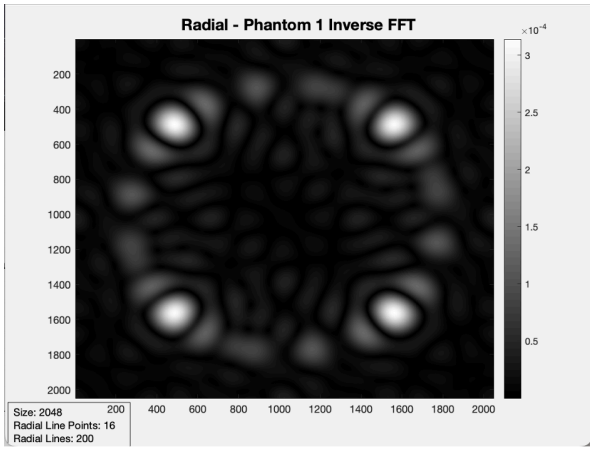
Point Density = 200
Radial Density = 16

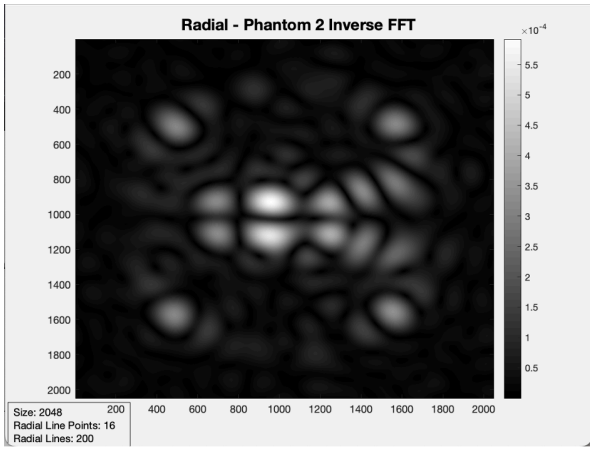




When increasing the Point Density, we see that this parameter helps with the clarity of the structure. We see multiple lines forming geometric structures.

Point Density = 16
Radial Density = 200



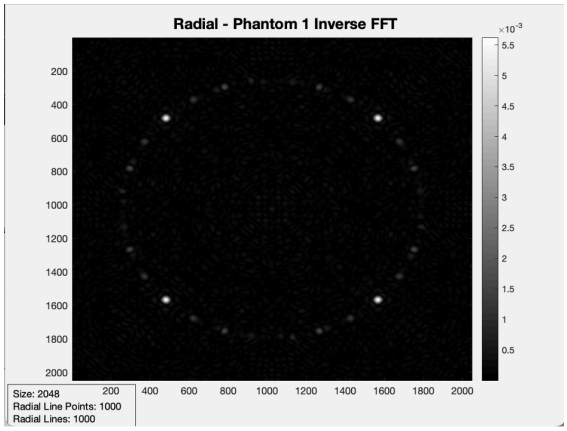


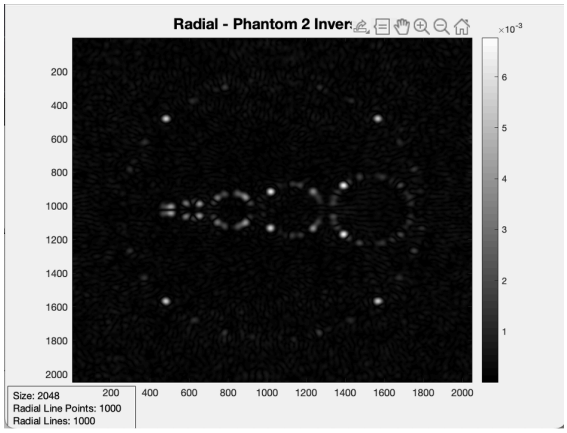
When increasing Radial Density, we see that the image seems to be more blurry and that the objects in the image contain more rounder shapes. As seen on both of the phantoms, but more clearly in the Phantom 1 Inverse, we see a circle taking shape.

Now when running with acquisition time, we will notice the effects that it has for the collected images.
We will be using the same Point Density and Radial Densityy while changing the acquisition time. The acquisition time can only have a value from 0 to 1.

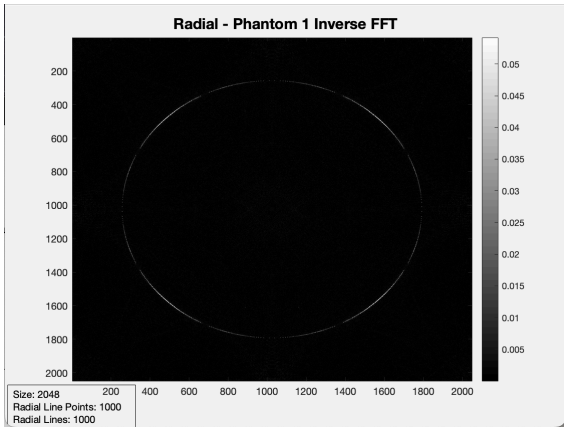
Point Density = 1000
Radial Density = 1000

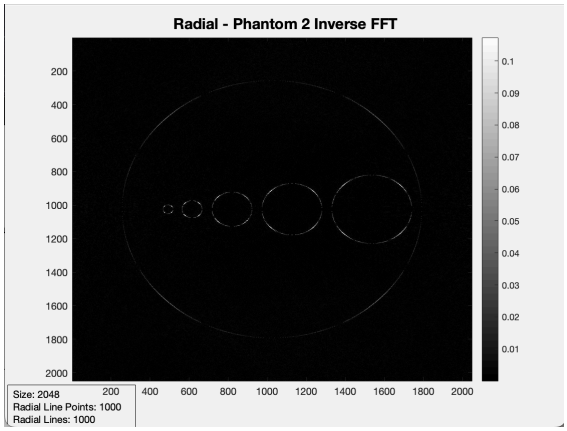
Acquisition Time: 0.1





Acquisition Time: 0.9





When viewing both images, it is clear that when the value of the acquisition time nears 1, the clarity of the generated image increases immensely.