PROJECT DOCUMENTATION: SAR IMAGE PROCESSING

Overview

This project aims to process Synthetic Aperture Radar (SAR) data to generate calibrated images. SAR is a remote sensing technique that creates high-resolution images of objects on the Earth's surface by transmitting and receiving microwave signals. The code provided processes raw SAR data, aligns signals, and generates pseudo-color images.

Requirements

To run the code, you need the following:

- OpenCV Library: The code utilizes OpenCV for image processing functionalities.
- Compiler with ISO C17 Support: The code uses ISO C17 (2018) standard features. Ensure your compiler supports this standard (e.g., /std:c17 flag for MSVC).

Setup Environment:

Install the OpenCV library.

- Configure your compiler to support ISO C17 standard.

Compile and Run:

- Compile the code using your preferred compiler with the necessary flags.
- Execute the compiled binary.

Input Data:

- Ensure your SAR data files are stored in the specified directory structure.
- Update the paths in the code if necessary.

Output:

- The code generates pseudo color images of the processed SAR data displayed during runtime.

Code Structure

This code is a C++ program for processing Synthetic Aperture Radar (SAR) data. Let's break down each module and its purpose:

Workflow Explanation

File Paths: The paths to the ambient data, coin horizontal data, and time data files are defined.

Read Data: The data from these files are read into vectors.

Interpolate Time Data: The time data is interpolated by a factor of 4.

Subtract Base Value: The base value (from the middle of the time data) is subtracted from the time data starting from 1/3rd of its length.

Apply Moving Average and Interpolation: Both ambient and coin horizontal data are smoothed using a moving average filter and then interpolated by a factor of 4.

Preprocess Data: The data is preprocessed to remove mutual coupling and align the signals.

Define Grid Points: The grid points for the x and z coordinates are defined.

Back Projection: The back-projection algorithm is used to create the radar image.

Create Image Mat: The resulting image data is stored in an OpenCV Mat object.

Normalize Image: The image is normalized to a range of 0 to 255 and a color map is applied.

Display and Save Image: The image is displayed using OpenCV and saved to a file.

Header Includes

This section includes necessary header files for the program, such as <iostream> for input-output operations, <opencv2/opencv.hpp> and <opencv2/highgui.hpp> for image processing, and others for file handling, numerical operations, etc.

```
#include <iostream>
#include <vector>
#include <cmath>
#include <opencv2/opencv.hpp>
#include <opencv2/highgui.hpp>
#include <codecvt>
#include <filesystem>
#include <algorithm>
#include <numeric>
#include <locale>
#include <fstream>
#include <sstream>
#include <string>
#include <iterator>
#include <opencv2/core.hpp>
#include <opencv2/imgproc.hpp>
```

Utility Functions

Reading Data from Text Files:

read1DTXTFile: Reads a single-column text file into a vector<double>.

```
// Function to read a 1D array from a text file
vector<double> read1DTXTFile(const string& filename) {
       vector<double> data;

       // Open the text file
    ifstream file(filename);
    if (!file.is_open()) {
        cerr << "Error opening text file: " << filename << endl;
        return data;
    }
    string line;
    while (getline(file, line)) {
             double value;
             stringstream ss(line);
        if (ss >> value) {
                    data.push_back(value);
             }
       }
    file.close(); // Close the file
    return data;
}
```

readTXTFile: Reads a multi-column text file into a vector<vector<double>>.

```
// Function to read a 2D array from a text file
vector<vector<double>> readTXTFile(const string& filename) {
    vector<vector<double>> data;

    // Open the text file
    ifstream file(filename);
    if (!file.is_open()) {
        cerr << "Error opening text file: " << filename << endl;
        return data;
    }
    // Read the file line by line
    string line;
    while (getline(file, line)) {
        vector<double> row;
        stringstream ss(line);
        double value;

        // Split each line by whitespace (space, tab, etc.) and read values
        while (ss >> value) {
            row.push_back(value);
        }
        // Add row to data vector only if it's not empty (avoid empty lines)
        if (!row.empty()) {
            data.push_back(row);
        }

           }
    file.close(); // Close the file

    return data;
}
```

Readfile: Reads multiple text files from a directory and processes each file to extract the second column after a specific header line.

```
vector<vector<double>> Readfile(const string& Folder) {
    if (!fs::exists(Folder) || !fs::is_directory(Folder)) {
        throw std::runtime_error("Provided path is not a valid directory.");
    }
    vector<vector<double>> dataCollection;
    for (const auto& entry : fs::directory_iterator(Folder)) {
        if (entry.is_regular_file() && entry.path().extension() == ".txt") {
            ifstream inFile(entry.path() );
            if (!inFile) {
                cerr << "Could not open file: " << entry.path() << endl;
                continue; // Skip to the next file if we can't open this one
            }
            string line;
            bool dataSection = false;
            vector<double> fileData;

            while (getline(inFile, line)) {
                if (line.find("Data,") != string::npos) {
                    dataSection = true;
                    continue;
                }

                if (dataSection) {
                    istringstream iss(line);
                    string value;
                    double y;

                    // Attempt to extract x and y values
                    getline(iss, value, ',');
                    if (getline(iss, value, ',')) {
                        y = stod(value);
                        fileData.push_back(y); // Assuming you want to collect only
the second column
                    }
                }
            }
            if (!fileData.empty()) {
                dataCollection.push_back(fileData); // Add the file's data to the
collection
            }
            inFile.close();
        }
    }

    return data collection;
}
```

Data Processing:

Moving Average: Applies a moving average filter to smooth the data, removing noise by adjusting the window size.

```
vector<double> movingAverage(const vector<double>& signal, size_t windowSize) {
    vector<double> smoothedSignal(signal.size(), 0.0);
    size_t N = signal.size();
    for (size_t i = 0; i < N; ++i) {
        size_t start = (i >= windowSize / 2) ? i - windowSize / 2 : 0;
```

```
        size_t end = (i + windowSize / 2 < N) ? i + windowSize / 2 : N - 1;
        double sum = 0.0;
        for (size_t j = start; j <= end; ++j) {
            sum += signal[j];
        }
        smoothedSignal[i] = sum / (end - start + 1);
    }
    return smoothedSignal;
}
```

Interpolation: Interpolates the signal by a factor of 4 as there are 2048 samples. By interpolating, it goes to 8189 samples in each signal.

```
vector<double> interpolateBy4(const std::vector<double>& signal) {
    size_t N = signal.size();
    vector<double> interpolatedSignal(4 * N - 3); // (4 - 1) * N + 1

    for (size_t i = 0; i < N - 1; ++i) {
        interpolatedSignal[4 * i] = signal[i];
        interpolatedSignal[4 * i + 1] = signal[i] + 0.25 * (signal[i + 1] -
signal[i]);
        interpolatedSignal[4 * i + 2] = signal[i] + 0.50 * (signal[i + 1] -
signal[i]);
        interpolatedSignal[4 * i + 3] = signal[i] + 0.75 * (signal[i + 1] -
signal[i]);
    }

    interpolatedSignal.back() = signal.back();

    return interpolatedSignal;
}
```

Normalization: Normalizes the signal to a range between -1 and 1 to cancel out the maximums so the mutual coupling signal is removed.

```
vector<double> normalize(const vector<double>& signal) {
    vector<double> normalizedSignal = signal;
    double minVal = *min_element(signal.begin(), signal.end());
    double maxVal = *max_element(signal.begin(), signal.end());
    for (size_t i = 0; i < signal.size(); ++i) {
        normalizedSignal[i] = 2 * (signal[i] - minVal) / (maxVal - minVal) - 1;
    }
    return normalizedSignal;
}
```

Algorithm Functions

alignAndAverage Function

This function aligns multiple ambient signals based on their maximum peak values and computes their average. It is designed to normalize the signals, find the peak index, shift the signals for alignment, and then average them.

Steps in alignAndAverage Function

Input Validation and Initialization:

Check if the input ambient vector is empty.

Initializes variables: M (number of vectors), N (length of each vector), amb_signal (to store the averaged signal), maxIndices (to store the index of the maximum peak in each signal), and alignedSignals (to store the aligned signals).

Normalization: Normalizes each signal in ambient using the normalize function.

Find Maximum Peak Indices and Align Signals: Iterates over each signal to find the index of the maximum peak value.

Prints the maximum peak index for debugging.

Align each signal based on the maximum peak index by shifting the signals so that their peaks are aligned with the peak of the first signal.

Averaging the Aligned Signals: Computes the average of the aligned signals by summing the corresponding elements and dividing by the number of signals.

Return the Averaged Signal

```
vector<double> alignAndAverage(vector<vector<double>> ambient, const string& method
= "Max Peak") {
    if (ambient.empty() || ambient[0].empty()) return {};

    size_t  M = ambient.size();     // Number of vectors
    size_t N = ambient[0].size();  // Length of each vector
    vector<double> amb_signal(N, 0.0);
    vector<int> maxIndices(M, 0);
    vector<vector<double>> alignedSignals(M,vector<double>(N, 0.0));
    for (size_t i = 0; i < M; ++i) {
        ambient[i] = normalize(ambient[i]);
    }
    for (size_t i = 0; i < M; ++i) {

        auto it = max_element(ambient[i].begin(), ambient[i].end(),
            [](double a, double b) { return abs(a) < abs(b); });
        int maxIndex = distance(ambient[i].begin(), it);
        maxIndices[i] = maxIndex;
        cout << "Maximum peak index in ambient[" << i << "]: " << maxIndex << endl;
        cout << "max" << ambient[i][maxIndex] << endl;
        // Shift/align signals based on max peak index

        int shiftAmount = maxIndices[0] - maxIndex; // Aligning to the first
signal's max peak index
        for (size_t j = 0; j < N; ++j) {
            int newIndex = j + shiftAmount;
            if (newIndex >= 0 && newIndex < N) {
                alignedSignals[i][newIndex] = ambient[i][j];
            }
        }
    }
    // 2. Average the aligned signals
    vector<double> averageSignal(N, 0.0);
    for (size_t j = 0; j < N; ++j) {
        double sum = 0.0;
```

```
        for (size_t i = 0; i < M; ++i) {
            sum += alignedSignals[i][j];
        }
        averageSignal[j] =sum/ M;
    }

    return averageSignal;
}
```

align Function

This function aligns Coin_Horizontal signals concerning signal_amb and subtracts the ambient signal to minimize mutual coupling. It includes normalization, alignment based on peak values, subtraction of the ambient signal, and optional plotting of results.

Steps in align Function

Input Validation and Initialization: Checks if the input Coin_Horizontal vector is empty.

Initializes variables: M (number of vectors), N (length of each vector), alignedSignals (to store aligned signals), new_Coin_Horizontal (to store normalized signals), normal_Coin_Horizontal (to store signals after ambient subtraction), signal_tar (to store the final target signal), new_signal_amb (to store aligned ambient signal), and max coin (to store the index of the maximum peak in each signal).

Normalization: Normalizes each signal in Coin_Horizontal.

Find Maximum Peak Index in signal_amb: Finds the index of the maximum peak value in signal_amb.

Find Maximum Peak Indices in Coin_Horizontal and Align Signals: Iterates over each signal in Coin_Horizontal to find the index of the maximum peak value.

Prints the maximum peak index for debugging.

Align each signal based on the maximum peak index by shifting the signals so that their peaks are aligned with the peak of the first signal.

Align signal_amb: Aligns signal_amb based on the shift amount calculated from the maximum peak index of the first signal in Coin_Horizontal.

Subtract signal_amb from Coin_Horizontal: Subtracts the aligned ambient signal from the aligned Coin_Horizontal signals to minimize mutual coupling.

Optionally extracts a portion of the signal to focus on the target signal.

Plot and Return Aligned Signals: Plots the results for visualization.

Returns the aligned and processed signals.

```
vector<vector<double>> align(vector<vector<double>> Coin_Horizontal, vector<double>
signal_amb, const string& method = "Max Peak") {
    if (Coin_Horizontal.empty() || Coin_Horizontal[0].empty()) return {};
```

```
    size_t M = Coin_Horizontal.size();     // Number of vectors
    size_t N = Coin_Horizontal[0].size();  // Length of each vector
    vector<vector<double>> alignedSignals(M, vector<double>(N, 0.0));
    vector<vector<double>> new_Coin_Horizontal(M, vector<double>(N, 0.0));
    vector<vector<double>> normal_Coin_Horizontal(M, vector<double>(N, 0.0));
    vector<vector<double>> signal_tar(M, vector<double>(N, 0.0));
    vector<double> new_signal_amb(N, 0.0);
    vector<int> maxcoin(M, 0);
    for (size_t i = 0; i < M; ++i) {
        new_Coin_Horizontal[i] = normalize(Coin_Horizontal[i]);
    }
    plotColumn(new_Coin_Horizontal[0],"coin1");
    auto it = max_element(signal_amb.begin(), signal_amb.end(),
        [](double a, double b) { return abs(a) < abs(b); });
    int maxamb = distance(signal_amb.begin(), it);
    cout << "Maximum peak index in signal_amb: " << maxamb << endl;
    cout << "max" << signal_amb[maxamb] << endl;
    for (size_t i = 0; i < M; ++i) {
        auto it = max_element(new_Coin_Horizontal[i].begin(),
new_Coin_Horizontal[i].end(),
            [](double a, double b) { return abs(a) < abs(b); });
        int maxIndex = distance(new_Coin_Horizontal[i].begin(), it);
        maxcoin[i] = maxIndex;
        cout << "Maximum peak index in new_coin_horizontal[" << i << "]: " <<
maxIndex << endl;
        cout << "max: " << new_Coin_Horizontal[i][maxIndex] << endl;

        // Shift/align signals based on max peak index
        int shiftAmount = maxcoin[0] - maxIndex; // Aligning to the first signal's
max peak index
        for (size_t j = 0; j < N; ++j) {
            int newIndex = j + shiftAmount;
            if (newIndex >= 0 && newIndex < N) {
                alignedSignals[i][newIndex] = new_Coin_Horizontal[i][j];
            }
            else {
                        alignedSignals[i][j] = 0;
                }
        }
    }
    int refIndex = maxcoin[0];
    int shiftAmb =  maxamb-refIndex;
    cout << "Shift amount for signal_amb: " << shiftAmb << endl;
    for (size_t j = 0; j < N; ++j) {
        int newIndex = j + shiftAmb;
        if (newIndex >= 0 && newIndex < N) {
            new_signal_amb[j] = signal_amb[newIndex];
        }
        else {new_signal_amb[j] = 0;}
    }
    // Subtract signal_amb from aligned Coin_Horizontal to cancel out mutual
coupling
    for (size_t i = 0; i < M; ++i) {
        for (size_t j = 0; j < N; ++j) {
            normal_Coin_Horizontal[i][j] =alignedSignals[i][j] - new_signal_amb[j];
            if (j > floor(length/3) && j<floor(length-length/15)) { // Time gating
based on the signal
```

```
                        signal_tar[i][j] = normal_Coin_Horizontal[i][j];
                    }
                else {
                        signal_tar[i][j] = 0;
                    }

            }

        }

        return signal_tar;
}
```

Global Back Projection

Purpose

The global_back_projection function is designed to reconstruct an image from a set of data acquired from multiple apertures (or sensors). This technique is commonly used in synthetic aperture radar (SAR) and other imaging systems that involve back-projection algorithms.

Inputs

sc1_data: A 2D vector containing the measured data from the apertures.

X_p: A vector containing the positions of the apertures.

x_grid: A vector containing the x-coordinates of the grid points in the image plane.

z_grid: A vector containing the z-coordinates of the grid points in the image plane.

Imagetotal1: A reference to a 2D vector that will store the accumulated image data.

new_time_data: A vector containing time data used for interpolation.

Outputs

Imagetotal1: The final reconstructed image matrix after normalizing the pixel values.

Key Steps in the Function

Initialization:

Imagetemp and Imagetotal1 are initialized as 2D vectors of size Nx by Nz filled with zeros.

Imagetemp is a temporary matrix to hold intermediate calculations, while Imagetotal1 accumulates the final image data.

Aperture Loop:

The function iterates over each aperture (N_aper times). For each aperture, it calculates the contribution to the image from that specific aperture position.

Grid Points Loop:

For each aperture position x_p (from X_p), the function iterates over all grid points defined by x_grid and z_grid.

The Euclidean distance formula calculates the distance r from the aperture to the grid point.

, where c0 is the speed of the wave propagation medium.

Time Interpolation:

The function uses binary search (lower_bound) to find the closest time index in new_time_data.

The amplitude p of the signal at the grid point is interpolated between the two closest time points using linear interpolation.

Image Accumulation:

The interpolated amplitude p is added to the corresponding position in Imagetotal1.

Normalization:

After processing all apertures and grid points, the function normalizes Imagetotal1 by dividing each element by the maximum value in the matrix. This step ensures the pixel values are within a consistent range.

Detailed Description of the Loops and Calculations

Aperture Loop (for (int n = 0; n < N_aper; ++n)):

For each aperture, x_p is retrieved from X_p.

Grid Points Loop (for (int i = 0; i < Nx; ++i)):

The function iterates over each x-coordinate (x_grid).

Nested within this loop, it iterates over each z-coordinate (z_grid).

Distance and Time Calculation:

Distance (r): The distance from the aperture to the grid point is calculated

Travel Time (t): The round-trip travel time

Interpolation:The function finds the closest index in new_time_data for the calculated travel time t:

auto it = lower_bound(new_time_data.begin(), new_time_data.end(), t);

int index = it - new_time_data.begin() + 3500;

Linear interpolation is used to estimate the amplitude p at the grid point:

double alpha = (t - new_time_data[index]) / (new_time_data[index + 1] - new_time_data[index]);

p = (1 - alpha) * sc1_data[n][index] + alpha * sc1_data[n][index + 1];

Accumulation:The interpolated amplitude p is added to Imagetotal1:

Imagetotal1[i][k] += p;

Normalization: The maximum value in Imagetotal1 is found and used to normalize all values to the range [0, 1].

```
vector<vector<double>> global_back_projection(const vector<vector<double>>&
sc1_data,
    const vector<double>& X_p,
    const vector<double>& x_grid,
    const vector<double>& z_grid,
    vector<vector<double>>& Imagetotal1, vector<double>& new_time_data) {
    // Initialize temporary and final image matrices
    vector<vector<double>> Imagetemp(Nx, vector<double>(Nz, 0));
    Imagetotal1 = vector<vector<double>>(Nx, vector<double>(Nz, 0));
    int length = sc1_data[1].size();
    int N_length = sc1_data.size();
    for (int n =0; n < N_length; ++n) {
        double x_p = X_p[n];

        for (int i = 0; i < Nx; ++i) {
            for (int k = 0; k < Nz; ++k) {
                double r = sqrt(pow(x_grid[i] - x_p, 2) + pow(z_grid[k], 2));
                double t = (2.0 * r) / (c0*100);
                double tau = t * 1e9;

                auto it = lower_bound(new_time_data.begin(), new_time_data.end(),
t);
                int index = it - new_time_data.begin()+floor(length/3);
                double p;

                if (index >= 0 && index < Nr_data*4) {
                    if (index == Nr_data - 1) {
                        p = sc1_data[n][index];
                    }
                    else {
                        double alpha = (t - new_time_data[index]) /
(new_time_data[index + 1] - new_time_data[index]);
                        p = (1 - alpha) * sc1_data[n][index] + alpha *
sc1_data[n][index + 1];
                    }
```

```
                    Imagetotal1[i][k] += p;

                }
            }
        }
    }
    double maxVal = 0.0;
    for (const auto& row : Imagetotal1) {
        for (double val : row) {
            maxVal = max(maxVal, val);
        }
    }
    for (auto& row : Imagetotal1) {
        for (double& val : row) {
            val /= maxVal;
        }
    }

    return Imagetotal1;
}
```

Plotting

Plotcolumn: The plotColumn function orchestrates a sequence of essential steps to create and display a line plot within a graphical window using OpenCV. It establishes a window with a designated name and then calculates the appropriate dimensions to accommodate the line plot. Subsequently, it generates an image canvas with a white background using OpenCV's Mat class, ready to draw the line plot. The function accurately represents the data by defining ranges for the x and y axes based on the input column's size. It further computes scaling factors to map data points to pixel coordinates on the image, facilitating precise plotting. As it iterates through the data points, it calculates the pixel coordinates for each point and connects them with lines, effectively generating the line plot. Axis lines are drawn to provide visual context, and the resulting plot is displayed in the window using the imshow function, allowing users to explore and analyze the data interactively**.**

Main Function

Reading Data: Reads data from two separate files, ambient_data, and coin_hori_data, storing them in vectors named ambient and Coin_Horizontal, respectively.

Changes the paths accordingly

```
const char* ambient_data = "F:\\project\\newsar\\ambient";
const char* coin_hori_data = "F:\\project\\newsar\\Data";
const char* timedata = "F:\\project\\newsar\\Data\\126.txt";
vector<vector<double>>ambient = Readfile(ambient_data);
vector<vector<double>>Coin_Horizontal = Readfile(coin_hori_data);
vector<double>time_data = read1DTXTFile(timedata);
```

Filter: Applies a moving average filter with a window size of 20 to each signal in both ambient and Coin_Horizontal.

Interpolation: Interpolates each signal by a factor of 4 to increase data resolution.

```
for (auto& signal : ambient) {
    signal = movingAverage(signal, 20);
}
// Step 2: Interpolate each signal by 4
for (auto& signal : ambient) {
    signal = interpolateBy4(signal);
}
for (auto& signal : Coin_Horizontal) {
      signal = movingAverage(signal, 20);
}
// Step 2: Interpolate each signal by 4
for (auto& signal : Coin_Horizontal) {
      signal = interpolateBy4(signal);
}
```

Preprocessing: Preprocess data using a function called PreprocessData with ambient, Coin_Horizontal, and para parameters.

Two steps: alignAndAverge and align to run the algorithm

```
vector<vector<double>>sc1_data = PreprocessData(ambient,Coin_Horizontal);
```

```
vector<vector<double>> PreprocessData(const vector<vector<double>>&
ambient,vector<vector<double>>& Coin_Horizontal)
{
    int M = Coin_Horizontal.size();     // Number of vectors
    int N = Coin_Horizontal[0].size();
    setprecision(3);
    vector<double>signal_amb = alignAndAverage(ambient, "Max Peak");
    plotColumn(signal_amb,"signal_amb");
    vector<vector<double>>signal_tar = align(Coin_Horizontal,signal_amb, "Max
Peak");

      return signal_tar;
}
```

Plots the first signal in Coin_Horizontal using the plotColumn function.

Data Analysis: Calculates the size of ambient and Coin_Horizontal data matrices.Computes the sum of processed data (sc1_data) using the Addition function.

Color Mapping: Applies a color map to the normalized image data, enhancing visualization.

Displays the color-mapped images in separate windows and waits for user input before closing.

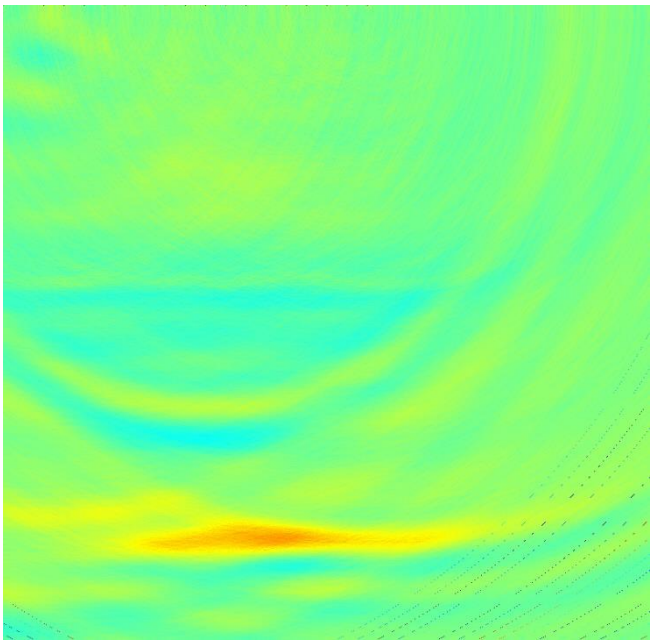
```
    vector<double> X_p(N_aper, 0);
    vector<vector<double>> grid(Nx, vector<double>(Nz, 0));
    vector<double> x_grid(Nx, 0);
    vector<double> z_grid(Nz, 0);
    vector<vector<double>> Imagetotal1;
    for (int i = 0; i < Nx; ++i) x_grid[i] = i * 0.01;
    for (int j = 0; j < Nz; ++j) z_grid[j] = j * 0.01;
    for (int i = 0; i < height; ++i) X_p[i] = 0.25 * (i);

     vector<vector<double>> imagegbp;
     // Call the back projection function
     imagegbp = global_back_projection(sc1_data, X_p, x_grid, z_grid, Imagetotal1,
new_time_data);
```

```
    cout << "Image size: " << imagegbp.size() << " x " << imagegbp[0].size() <<
endl;
     Mat imageMat(Nx, Nz, CV_8UC1);
     for (int i = 0; i < Nx; ++i) {
        for (int j = 0; j < Nz; ++j) {
            imageMat.at<uchar>(j, i) = static_cast<uchar>((imagegbp[i][j] + 1) *
127.5);
        }
     } // Normalize the image to the range 0 to 255
     Mat normalizedImage;
     applyColorMap(imageMat, normalizedImage, COLORMAP_JET);
     // Display the image
     namedWindow("Normalized Image with Colorbar", WINDOW_AUTOSIZE);
     imshow("Normalized Image with Colorbar", normalizedImage);
     imwrite("F:\\project\\newsar\\displayed_imaged_new6.png", normalizedImage);
     waitKey(0);
    return 0;
}
```

Results:

Resulted Image 1: Shows the Coin as the Red color region.



Additional Requirements: To show a gridded display and colorbar, additional libraries such as matplotlib are needed. This library can be used to create more detailed visualizations with grids and colorbars that enhance the interpretability of the radar images.

Source: **https://github.com/lava/matplotlib-cpp.git**