 **Introduction**

 **Calculating Mueller matrices of stratified media**

* Building a model
* Spectroscopic data
  + Berreman method
  + Partial wave method
* K-space maps

 **Plotting classes**

* MPlot
* MPlot3D

 **Measurement classes**

* CRRmeasurement
* muellerData
* Custom indexing and object-arrays
* Methods
* Plotting

**1. Introduction**

The introduction section in the document provides a brief overview of the materials and code contained in the text. It explains that the document serves as a supplement to the author's PhD thesis, titled “Coherence in Polarimetry.” The relevant models for calculating Mueller matrices of stratified media are described in the referenced work. The user is expected to have familiarity with these models before using the provided programs.

Additionally, the document contains custom MATLAB classes that facilitate the production of plots, such as error bars, suitable for both measured values and simulations. Furthermore, the document also introduces two measurement classes designed to assist in reading and processing data from a custom polarimeter developed by the research group. The documentation for these tools is also published alongside the author’s thesis work.

In summary, the introduction sets the stage for using the provided MATLAB code to calculate and visualize Mueller matrices of stratified media while emphasizing that some background knowledge in the field is essential for effective use of the programs

**2. Calculating a mueller matrix of a stratified media**

**2.1** **Building a Model**,

It focuses on constructing a physical model of layered materials for the purpose of calculating their optical properties (specifically, the Mueller matrix). Each layer in the multilayer system is characterized by five key parameters:

1. **Material**: This is a string that specifies the name of the material used for the layer. The material name must exist in predefined MATLAB scripts (materialLib.m or materialLibIso.m).
2. **Thickness (d)**: The thickness of the layer, specified in nanometers. A value of Inf can be used to model a semi-infinite substrate, which indicates that light does not pass through the layer.
3. **Euler Angles (eul)**: These angles represent the rotation of the layer in 3D space using a ZXZ Euler rotation, which helps in determining the layer’s orientation.
4. **Thick Layer Boolean (bool\_thick)**: This is a boolean flag that specifies whether the layer is considered "thick" for certain calculations, such as the partial wave method.
5. **Isotropic Boolean (bool\_isotropic)**: This boolean indicates whether the material is isotropic (the same in all directions). A value of true means the material behaves uniformly, regardless of direction.

**How Layers Are Structured:**

Each layer is represented as a cell array (layer) containing these parameters. Multiple layers are then combined into an array (layerArray) to represent the entire multilayer structure. The first and last layers represent the incident and exit media, respectively, which are often air but can be any isotropic material.

**Examples:**

* **A system with air on both sides and a 1 mm thick quartz layer with a gold coating on both sides:**

**layerArray{1} = {'air', 0, [0 0 0], 0, 1};**

**layerArray{2} = {'AuFilm', 20, [0 0 0], 0, 1};**

**layerArray{3} = {'+quartz', 1000000, [24 0.3 0], 1, 0};**

**layerArray{4} = {'AuFilm', 20, [0 0 0], 0, 1};**

**layerArray{5} = {'air', 0, [0 0 0], 0, 1};**

* **A 30 nm TiO2 film on a silicon wafer, with the optic axis of TiO2 in-plane:**

**layerArray{1} = {'air', 0, [0 0 0], 0, 1};**

**layerArray{2} = {'TiO2', 30, [0 90 0], 0, 0};**

**layerArray{3} = {'Si100', Inf, [0 0 0], 0, 1};**

Euler angles in this context are used to describe the rotation of the optical axes of anisotropic materials in a multilayer structure. These angles help orient the crystal axes of materials like quartz or TiO₂, where optical properties change depending on the direction of light propagation.

In the ZXZ Euler rotation convention, the orientation of the optical axis of a layer is described by three angles. These are set in the [alpha beta gamma] format, representing rotations about different axes:

1. Alpha (α): Rotation about the z-axis.
2. Beta (β): Rotation about the x-axis (after the z-axis rotation).
3. Gamma (γ): A second rotation about the z-axis (after the previous rotations).

**Setting Euler Angles:**

* The Euler angles are given in degrees and specify how much the optical axis of the material is rotated from its default orientation.

**Example:**

In the following example:

layerArray{2} = {'TiO2', 30, [0 90 0], 0, 0};

* [0 90 0] means:
  + **Alpha (0°)**: No initial rotation around the z-axis.
  + **Beta (90°)**: The optic axis is rotated by 90 degrees around the x-axis, which means the optic axis of TiO₂ is now oriented in the plane of the layer (in-plane).
  + **Gamma (0°)**: No further rotation around the z-axis after the first two steps.

This specific set of angles aligns the TiO₂ optic axis parallel to the plane of the material, which is typical when modeling materials that have different optical properties depending on their orientation.

**Another Example:**

For a quartz layer with a slight tilt:

layerArray{3} = {'+quartz', 1000000, [24 0.3 0], 1, 0};

* [24 0.3 0] means:
  + **Alpha (24°)**: The optical axis is rotated by 24 degrees around the z-axis.
  + **Beta (0.3°)**: A very slight tilt around the x-axis.
  + **Gamma (0°)**: No additional rotation around the z-axis.

This setup might be used when the quartz crystal is oriented at a specific angle to study its birefringence effects in the material.

Euler angles [alpha beta gamma] control the orientation of the optical axis of the material in 3D space. These angles are critical when dealing with **anisotropic materials**, where the direction of the optical axis influences the interaction of light with the material.

**2.2.1 Berreman Method**

It provides details on calculating spectroscopic Mueller matrices using the **fully coherent Berreman method**. This method is particularly useful for modeling thin-film optics, especially in cases where the layers are considered thin, as it does not use the bool\_thick parameter for "thick" layers.

**Key Inputs:**

1. **layerArray**: This defines the structure of the layers as described in Section 2.1. Each layer's material, thickness, orientation (using Euler angles), and whether it is isotropic or anisotropic are described here.
2. **wavelengths**: A numeric array representing the wavelengths (in nanometers) at which the Mueller matrix calculations should be performed.
3. **aoi (Angle of Incidence)**: The angle in degrees at which the incident light interacts with the layer stack.
4. **bReflect**: A boolean value that determines whether the calculation is for **reflection** (true) or **transmission** (false).
5. **bNormalize**: A boolean option that, if true, normalizes the output Mueller matrices by dividing each element by M11, the first element of the Mueller matrix.

**Output:**

* The output **M** is a 4x4xN array, where N is the number of wavelengths. This array represents the Mueller matrices at each specified wavelength.

**Computational Approach:**

* The Berreman method relies on matrix and Kronecker products, which are **fully vectorized** to optimize speed. Most of the computation time is spent constructing and calculating the Berreman layer matrices.
* The code is already highly efficient, with typical calculations taking less than one second, although further improvements could be made by using analytical solutions for the matrix exponential (if applicable) and leveraging point group symmetries to optimize the construction of the delta matrix.

In summary, the **Berreman method** is a fast and efficient way to compute the optical response (Mueller matrices) of stratified media, particularly for thin layers. The function supports reflection and transmission configurations, and its output can be used to understand the interaction of light with multilayer structures

The provided code implements the **Berreman method** for calculating the **Mueller matrices** of a multilayer structure, which can be used for both reflection and transmission cases. Let's go through the code step by step to explain how it works.

**Key Variables and Concepts:**

1. **layerArray**: This input defines the multilayer structure, with each layer having specific properties (material, thickness, Euler angles, etc.). The first and last elements represent the incident and exit media (often air).
2. **wavelengths**: This array represents the wavelengths at which the Mueller matrix is calculated.
3. **aoi (angle of incidence)**: This is the angle at which light hits the structure. The code converts it from degrees to radians.
4. **bReflect**: Boolean flag to determine if the calculation is for reflection (true) or transmission (false).
5. **bNormalize**: Boolean flag indicating whether to normalize the Mueller matrix by the first element (M11).

**Major Steps in the Code:**

**1. Initialization**

aoi = aoi\*pi/180;

A = [1,0,0,1;1,0,0,-1;0,1,1,0;0,1i,-1i,0];

Ainv = [0.5,0.5,0,0;0,0,0.5,-0.5\*1i;0,0,0.5,0.5\*1i;0.5,-0.5,0,0];

 The angle of incidence (aoi) is converted from degrees to radians.

 Matrices A and Ainv are transformation matrices used in the Berreman method. These help transform between different representations of the electric field.

**2. Incident Medium Setup**

if strcmp(layerArray{1}{1}, 'air')

t1 = 1 / (2 \* cos(aoi));

Psi\_i = [t1, 1/2, 0, 0; 0, 0, 1/2, t1; -t1, 1/2, 0, 0; 0, 0, 1/2, -t1];

n0 = ones(1, length(wavelengths));

else

[Psi\_i, n0] = psiAmbientInv(layerArray{1}, aoi, wavelengths);

End

* **Incident Medium (Psi\_i)**: If the first layer is air, the transfer matrix (Psi\_i) is calculated based on the angle of incidence. Otherwise, the function psiAmbientInv is used to calculate Psi\_i and the refractive index n0 for the given medium.
* **n0:** The refractive index of the incident medium (set to 1 for air).

**3. Wavevector Calculation**

kx = n0 .\* sin(aoi); % x-component of the wavevector

**kx** is the x-component of the wavevector, which depends on the refractive index n0 and the angle of incidence.

**4. Exit Medium Setup**

if strcmp(layerArray{Nlayers}{1}, 'air')

t1 = cos(aoi);

Psi\_e = [t1,0,-t1,0;1,0,1,0;0,1,0,1;0,t1,0,-t1];

elseif layerArray{Nlayers}{5} == 0

Psi\_e = psiAniso(layerArray{Nlayers}, wavelengths, kx);

elseif layerArray{Nlayers}{2} == Inf

Psi\_e = psiIso(layerArray{Nlayers}, wavelengths, kx);

else

Psi\_e = psiAmbient(layerArray{Nlayers}, aoi, wavelengths);

End

**Exit Medium (Psi\_e)**: Similar to the incident medium, the exit medium transfer matrix is calculated. If the last layer is air, it uses a simple matrix. Otherwise, functions like psiAniso (for anisotropic materials) and psiIso (for isotropic materials) are used to calculate Psi\_e.

**5. Layer Matrices Calculation**

for m = 1:Nlayers-2

if layerArray{m+1}{5} == 0

layerMat(:,:,:,m) = layerBerreman(layerArray{m+1}, wavelengths, kx);

else

layerMat(:,:,:,m) = layerBerremanIso(layerArray{m+1}, wavelengths, kx);

end

end

**Layer Transfer Matrices**: For each layer (except the first and last), a 4x4 transfer matrix is calculated using the **Berreman method**. If the layer is anisotropic (bool\_isotropic = 0), the layerBerreman function is used; otherwise, layerBerremanIso is used for isotropic layers.

**6. Mueller Matrix Calculation**

if Nlayers == 2

P = multiprod(Psi\_i, Psi\_e);

elseif Nlayers > 3

P = layerMat(:,:,:,Nlayers-2);

for k = 1:(Nlayers-3)

P = multiprod(layerMat(:,:,:,Nlayers-2-k), P);

end

P = multiprod( multiprod(Psi\_i, P), Psi\_e);

else

P = layerMat(:,:,:,1);

P = multiprod( multiprod(Psi\_i, P), Psi\_e);

End

* The total transfer matrix **P** is computed by multiplying the transfer matrices of the layers. If there are only two layers (incident and exit), the calculation is straightforward. For more layers, a loop iteratively multiplies the transfer matrices of the intermediate layers.
* **multiprod** is a helper function to efficiently multiply matrices in 3D.

**7. Reflection or Transmission Calculation**

if bReflect

J = multiprod(P([3 4], [1 2], :), invert2x2(P([1 2], [1 2], :)));

else

J = invert2x2(P([1 2], [1 2], :));

End

 If the reflection is requested (bReflect = true), a specific block of the matrix **P** is used to compute the reflection. Otherwise, the transmission is calculated.

 **invert2x2** is a helper function to invert a 2x2 matrix block.

**8. Final Mueller Matrix Construction**

M = real(multiprod( multiprod(A, bigKron(J)), Ainv)) ;

The final Mueller matrix **M** is constructed by multiplying the result with transformation matrices (A and Ainv) to convert the matrix back to a real-valued representation.

**9. Normalization (Optional)**

if bNormalize

M = M ./ M(1, 1, :);

End

* If normalization is requested (bNormalize = true), each Mueller matrix is normalized by dividing by its first element (M11).

**10. Conclusion**

* This code calculates the **Mueller matrix** for a multilayer optical system using the **Berreman method**, taking into account reflection or transmission and different media.
* It efficiently handles thin layers and supports isotropic and anisotropic materials.
* The output is a real-valued Mueller matrix, which can be used to describe the optical properties of the system for the specified wavelengths and angles of incidence.

**Example of Full Script:**

Clc;

Clear all;

% Define the layer structure (air -> TiO2 -> Silicon)

layerArray{1} = {'air', 0, [0 0 0], 0, 1}; % Incident medium (air)

layerArray{2} = {'TiO2', 30, [0 90 0], 0, 0}; % TiO2 layer (30 nm)

layerArray{3} = {'Si100', Inf, [0 0 0], 0, 1}; % Silicon substrate (semi-infinite)

% Define wavelength range (in nanometers)

wavelengths = 300:10:800; % Wavelengths from 300 nm to 800 nm

% Set angle of incidence

aoi = 45; % Angle of incidence = 45 degrees

% Set calculation mode (reflection or transmission)

bReflect = false; % Set to false for transmission

% Set normalization option

bNormalize = true; % Normalize Mueller matrix by M11

% Run the Berreman method

M = mmBerreman(layerArray, wavelengths, aoi, bReflect, bNormalize);

% Display or further analyze the Mueller matrices

disp(M); % Display the result (you can also plot it as needed)

**Helper Functions:**

You need helper functions like multiprod, invert2x2, psiAniso, psiIso, and layerBerreman for this script to run successfully. Make sure these functions are available in your working directory or implement them based on your needs.

If you have all the necessary functions, the script will compute the Mueller matrices for your layered structure over the given range of wavelengths and angles of incidence.

**2.2.2** **Partial Wave Method**

It is introduced as an alternative to the Berreman method for calculating the spectroscopic Mueller matrices for multilayer systems with a "thick" layer. Here's a more detailed explanation:

**Key Points of the Partial Wave Method:**

1. **Applicability**:
   * The Partial Wave Method is specifically designed for scenarios where the multilayer system contains one "thick" layer, meaning a layer whose thickness is large compared to the wavelength of light.
   * Unlike the Berreman method, which assumes that all layers are thin (coherent interactions), the partial wave method can handle cases where a single thick layer requires incoherent summation of contributions from different points within the layer.
2. **How the Method Works**:
   * For a system with multiple layers, the bool\_thick flag is used to designate one of the layers as "thick." This parameter should be set to true for exactly one layer in the multilayer stack.
   * The input model for this method is almost identical to the Berreman method, including the description of the material layers and the physical properties (wavelengths, angle of incidence, etc.). However, the way the calculations are handled differs significantly due to the presence of a thick layer.
3. **Layer Description**:
   * Each layer in the model is described by its material properties, thickness, Euler angles for orientation, and whether it is isotropic or anisotropic. The partial wave method processes the thick layer in a way that accounts for the variation in the optical path length within the layer.
4. **Vectorization**:
   * Just like the Berreman method, the partial wave method uses vectorized operations (through the multiprod toolbox) to efficiently compute the Mueller matrices. This allows for fast calculations despite the complexity introduced by the thick layer.
5. **Output**:
   * The function mmPartialWave outputs the spectroscopic Mueller matrices for the multilayer system. The output format is similar to the Berreman method's result, providing a 4x4xN array of Mueller matrices, where N corresponds to the number of wavelengths used in the calculation.
6. **Use Cases**:
   * The partial wave method is particularly useful for systems that involve thick, anisotropic layers, such as crystals or other complex media. This method provides a more accurate representation of how light interacts with such materials compared to the Berreman method, which is more appropriate for thin layers.
7. **Efficiency**:
   * While both the Berreman and partial wave methods are vectorized for efficient computation, the partial wave method requires more complex handling of the thick layer. However, this approach remains computationally feasible for most practical applications in spectroscopy.

In summary, the **Partial Wave Method** is an enhancement over the Berreman method, enabling accurate computation of Mueller matrices for systems that include one thick layer, which cannot be assumed to exhibit fully coherent behavior .

**COMPARISON** between the **Berreman method** and the **Partial Wave method**

**1. Layer Thickness Assumptions:**

* **Berreman Method**:
  + Assumes **all layers** are thin relative to the wavelength of light. This means that light interacts with the layers coherently, with all parts of the layer contributing equally to the calculation.
  + No thick layer is allowed.
* **Partial Wave Method**:
  + Allows for **one thick layer**, meaning that this layer is thick compared to the wavelength of light. The interaction of light with this layer is treated differently, with incoherent summation of contributions from within the layer.
  + Suitable when one layer is thick and cannot be treated coherently like the others.

**2. Handling of Coherency:**

* **Berreman Method**:
  + Fully coherent. It assumes that light waves maintain phase relationships across all the layers, resulting in interference effects that are calculated based on all layers being thin.
* **Partial Wave Method**:
  + Partially incoherent. The thick layer introduces incoherence, meaning that different parts of the thick layer contribute incoherently to the final result. The rest of the system can still involve coherent interactions between the thin layers.

**3. bool\_thick Parameter:**

* **Berreman Method**:
  + Does not use the bool\_thick parameter. It treats all layers as thin by default.
* **Partial Wave Method**:
  + Specifically requires the bool\_thick parameter to be set to true for one layer, indicating that it is thick. This is crucial for the method to distinguish the thick layer from thin ones.

**4. Use Cases:**

* **Berreman Method**:
  + Best suited for systems where all the layers are thin (comparable to or smaller than the wavelength of light). Typical use cases include multilayer films, coatings, and stacks where the thicknesses of all layers are small.
* **Partial Wave Method**:
  + Designed for systems where one of the layers is thick, such as a bulk material or a substrate, while the other layers are thin. This is particularly useful in situations involving crystals, thick films, or bulk materials.

**5. Accuracy in Thick Layer Calculations:**

* **Berreman Method**:
  + It is not designed to handle thick layers. When used in systems with thick layers, it may give inaccurate results due to its assumption of full coherence.
* **Partial Wave Method**:
  + Provides accurate results for systems with one thick layer, as it takes into account the different contributions of light interacting with the thick layer at different depths. This makes it more suitable for such cases.

**6. Computational Approach:**

* **Berreman Method**:
  + It uses **fully vectorized matrix operations**, efficiently computing the Mueller matrices across all layers using matrix multiplication. The approach is fast due to its simplicity in dealing with thin layers only.
* **Partial Wave Method**:
  + Also uses vectorized operations, but it introduces additional complexity in handling the thick layer. Despite this, it remains efficient for most practical purposes by using optimized matrix multiplication routines.

**7. Speed and Efficiency:**

* **Berreman Method**:
  + Generally faster due to its assumption of thin layers and simpler matrix operations. It is ideal for systems where computational speed is a priority and the thin-layer approximation is valid.
* **Partial Wave Method**:
  + Slightly more computationally intensive than the Berreman method, as it must account for the thick layer. However, it is still optimized for fast computation using vectorized operations.

**8. Reflection and Transmission Calculations:**

* **Both Methods**:
  + Both methods can handle reflection and transmission calculations. The choice between reflection and transmission is specified by the bReflect parameter in both methods.

**9. Material Handling:**

* **Berreman Method**:
  + Works well with isotropic and anisotropic materials as long as they are thin. Each layer's material properties (e.g., refractive index) are incorporated into the matrix calculations.
* **Partial Wave Method**:
  + Handles isotropic and anisotropic materials, but is particularly useful when there is a thick anisotropic layer that requires special handling due to its significant thickness.

**10. Normalization:**

* **Both Methods**:
  + Both methods offer an option (bNormalize) to normalize the Mueller matrix by dividing by the first element (M11). This can be used in both methods for easier comparison of results.

**Summary of Differences:**

| **Aspect** | **Berreman Method** | **Partial Wave Method** |
| --- | --- | --- |
| **Layer Thickness Assumption** | Assumes all layers are thin | Allows one thick layer |
| **Coherency** | Fully coherent | Partially incoherent (thick layer) |
| **bool\_thick Parameter** | Not used | Used to designate one thick layer |
| **Best Use Case** | Thin-film systems | Systems with one thick layer |
| **Handling Thick Layers** | Inaccurate for thick layers | Accurate for systems with thick layers |
| **Computational Approach** | Fast, vectorized, handles thin layers | More complex due to thick layer but still vectorized |
| **Speed** | Generally faster due to simplicity | Slightly slower but optimized |
| **Reflection/Transmission** | Supports both | Supports both |
| **Material Handling** | Thin isotropic/anisotropic layers | Thick anisotropic layers handled accurately |
| **Normalization** | Available | Available |

**Conclusion:**

* Use the **Berreman method** when dealing with thin-film systems where all layers are thin, and fully coherent light interactions can be assumed.
* Use the **Partial Wave method** when you have one thick layer in your multilayer system and need to accurately model its contribution to the overall Mueller matrix, especially for systems involving bulk materials or thick films.

The mmPartialWave function is the **Partial Wave Method** for calculating the Mueller matrices of a multilayer system that contains one thick layer. This method is particularly useful when one of the layers is significantly thicker than the wavelength of light, requiring a different approach from the Berreman method. Let's break down the function step by step:

**Function Overview:**

* **Inputs**:
  1. **layerArray**: Describes the multilayer system, where each layer's material, thickness, orientation, and isotropy are specified.
  2. **wavelengths**: Array of wavelengths in nanometers at which the Mueller matrices are calculated.
  3. **aoi**: The angle of incidence of light in degrees.
  4. **bReflect**: Boolean to specify whether the calculation is for reflection (true) or transmission (false).
  5. **bNorm**: Boolean to specify whether the output Mueller matrix should be normalized by its first element (M11).
* **Step-by-Step Breakdown:**

**1. Convert Angle of Incidence to Radians:**

aoi = aoi.\*pi./180;

* Converts the angle of incidence (aoi) from degrees to radians, as required for trigonometric calculations.

**2. Determine the Location of the Thick Layer:**

g = 1;

while layerArray{g}{4} == 0

g = g + 1;

end

The while loop is used to find the first thick layer in the system. The flag layerArray{g}{4} checks the thickness condition for each layer. The loop iterates until it finds a thick layer, and the index of this layer is stored in g.

3. **Incident Medium Setup**:

if strcmp(layerArray{1}{1}, 'air')

t1 = cos(aoi);

Psi0 = [t1,0,-t1,0;1,0,1,0;0,1,0,1;0,t1,0,-t1];

n0 = ones(1, length(wavelengths));

else

[Psi0, n0] = psiAmbient(layerArray{1}, aoi, wavelengths);

End

If the incident medium is air, the transfer matrix **Psi0** is initialized with values based on the angle of incidence. If the medium is not air, the function psiAmbient is called to compute the transfer matrix and refractive index (n0).

4. **Wavevector Calculation**:

kx = n0 .\* sin(aoi); % x-component of the wavevector

The x-component of the wavevector (kx) is calculated using the refractive index n0 and the sine of the angle of incidence (aoi).

5. **Exit Medium Setup**:

if strcmp(layerArray{Nlayers}{1}, 'air')

t1 = cos(aoi);

Psi2 = [t1,0,-t1,0;1,0,1,0;0,1,0,1;0,t1,0,-t1];

elseif layerArray{Nlayers}{5} == 0

Psi2 = psiAniso(layerArray{Nlayers}, wavelengths, kx);

elseif layerArray{Nlayers}{2} == Inf

Psi2 = psiIso(layerArray{Nlayers}, wavelengths, kx);

else

Psi2 = psiAmbient(layerArray{Nlayers}, aoi, wavelengths);

End

The function computes the transfer matrix for the exit medium (Psi2). If the exit medium is air, a pre-defined matrix is used. For anisotropic materials, psiAniso is used, and for isotropic or semi-infinite layers, psiIso or psiAmbient is used.

6. **Handle Thin Layers Before the Thick Layer**:

if g > 2 %if there are thin layers before thick one, compute layer matrices

for m = 2:(g-1)

layerArray{m}{2} = -layerArray{m}{2}; %change sign of d to invert layer matrix

if layerArray{m}{5} == 0 %check if layer is anisotropic

Psi0 = multiprod(layerBerreman(layerArray{m}, wavelengths, kx), Psi0);

else

Psi0 = multiprod(layerBerremanIso(layerArray{m}, wavelengths, kx), Psi0);

end

end

end

This loop processes **thin layers** that appear **before** the thick layer. The layer matrix for each thin layer is computed using either the **anisotropic** (layerBerreman) or **isotropic** (layerBerremanIso) function. The transfer matrix Psi0 is updated iteratively by multiplying it with each layer matrix.

7. **Handle Thin Layers After the Thick Layer**:

if Nlayers - g > 1 %if there are thin layers after thick one, compute layer matrices

for m = (Nlayers-1):-1:(g+1)

if layerArray{m}{5} == 0 %check if layer is anisotropic

Psi2 = multiprod(layerBerreman(layerArray{m}, wavelengths, kx), Psi2);

else

Psi2 = multiprod(layerBerremanIso(layerArray{m}, wavelengths, kx), Psi2);

end

end

end

This loop processes **thin layers** that appear **after** the thick layer. As before, it checks if each layer is anisotropic or isotropic and computes the corresponding layer matrix, updating the transfer matrix Psi2.

8. **Calculate Mueller Matrix Using the Partial Wave Method**:

M = partialWave(Psi0, Psi2, layerArray{g}, wavelengths, kx, bReflect);

Once the thin layers before and after the thick layer have been processed, the **partial wave method** is applied by calling the partialWave function. This function computes the Mueller matrix (M) based on the thick layer's properties, the pre-processed transfer matrices (Psi0 and Psi2), and other parameters like the angle of incidence and whether the calculation is for reflection or transmission.

9. **Normalization (Optional)**:

if bNorm

M = M ./ M(1,1,:);

End

If the normalization flag (bNorm) is set to true, the Mueller matrix is normalized by dividing it by its first element (M11).

**Summary of How the Partial Wave Method Works:**

* The **Partial Wave Method** is designed for cases where there is one thick layer in a multilayer system. The function:
  + First identifies the thick layer (g) and sets up the incident and exit media.
  + It processes thin layers that appear before and after the thick layer, calculating the transfer matrices using the **Berreman method** or its isotropic variant.
  + Finally, the **partial wave method** is applied to the thick layer to compute the final Mueller matrix.
  + Optional normalization of the Mueller matrix is performed at the end if requested.

**How to Run This Function:**

* You would call this function similarly to the Berreman method but for a system with one thick layer. Ensure that the layerArray defines the correct structure of your system, with one thick layer flagged as such using bool\_thick.

**2.3 K-space Maps**

**K-space maps**, which are essential for visualizing and analyzing the propagation of waves, especially in optics and materials science. These maps are built by calculating the Mueller matrix for a regular grid of in-plane wave vector components rather than for a single angle of incidence. Here's an in-depth explanation of this process:

**Concept of K-space Maps:**

* **K-space** is essentially a reciprocal space where each point corresponds to a particular wave vector. In optics, it can be used to map how light propagates through or reflects off a material.
* Instead of computing a Mueller matrix for a single incident angle, **K-space maps** calculate it for a grid of in-plane incident wave vectors. This provides a more comprehensive view of the light interaction with the material.

**Types of K-Space Maps:**

Two types of K-space maps are outlined in this section:

1. **Conoscopic Maps:**
   * The specimen (material under study) remains **stationary**, and the azimuth (the angle describing the horizontal direction of the incoming wave vector) is also stationary relative to the instrument.
   * Conoscopic maps are directly measured in **back-focal plane K-space imaging systems** (i.e., systems designed to capture such maps).
2. **Polar Maps:**
   * In polar maps, the **specimen is physically rotated** with respect to the instrument. This is done to measure the Mueller matrix at various azimuthal angles.
   * Polar maps are typically obtained using traditional **ellipsometers**, which reorient the specimen with a motorized goniometer to change the direction of the incoming beam.

**Practical Usage:**

* The document explains that in media where there is little diffraction or scattering, **conoscopic maps** and **polar maps** contain equivalent information. These can be interconverted with simple matrix operations.

For **transmission** measurements:

Mcono​=R(−θ)Mpolar​R(θ)

For **reflection** measurements:

Mcono​=R(θ)Mpolar​R(θ)

Where R(θ) is the **rotation matrix** that accounts for the azimuthal rotation of the specimen or instrument.

**Coordinate Systems:**

* The document distinguishes between the **instrument axes** (denoted as x and y) and the **polar coordinates** used in the maps, where:
  + The radius corresponds to sin(θ), where θ is the angle of incidence.
  + The azimuthal angle ϕ represents the horizontal direction of the incoming wave.

This helps in interpreting the maps visually, as the spatial arrangement of wave vectors and the sample’s rotation are critical in understanding the resulting K-space maps.

**Example of K-Space Maps:**

* The document provides examples of simulated K-space maps for a 300 µm thick section of α-quartz, showing both transmission and reflection data. The **numerical aperture (NA)** of the system is specified as 0.85, which describes the range of angles the system can collect.

These maps are generated using both the **Berreman** and **Partial Wave** methods described earlier, depending on the complexity and thickness of the layers involved.

**Output of the Function:**

* Functions like mmBerremanMap and mmPartialWaveMap are used to compute these maps. Their inputs and outputs are structured as follows:

The **K-space maps** enable researchers to understand how light behaves over a range of incident angles and azimuths, providing deeper insights into the material's optical properties.

For further understanding of this topic, refer to additional sources such as back-focal plane imaging systems or ellipsometry methods, which can help in generating K-space maps experimentally.

**1. Understanding K-Space in Optical Calculations**

In optics, **K-space** (or reciprocal space) is the space where wave vectors (or momenta) are described. It is crucial in understanding wave phenomena such as diffraction, reflection, and transmission in layered media. In simple terms:

* A **wave vector (k)** is a vector that points in the direction of wave propagation and has a magnitude proportional to the wavelength of light.
* A **K-space map** provides a visual representation of how light interacts with the material at different angles of incidence (direction of propagation) and azimuths (rotation around the axis).

When performing these simulations, instead of computing for a single incident angle or wavelength, **K-space maps** allow you to compute how the entire spectrum of light interacts with the material across a grid of angles and directions. This gives a broader understanding of the material's optical behavior.

**2. Conoscopic and Polar Maps: Key Differences**

The document explains two different types of K-space maps—**Conoscopic Maps** and **Polar Maps**—both of which give valuable insights but are obtained and interpreted differently.

**Conoscopic Maps:**

* **What is it?**: In a conoscopic map, the incident light angles (azimuth and polar) are varied while the specimen itself remains stationary relative to the instrument.
* **Where is it used?**: This type of map is used in **back-focal plane imaging systems**, such as in **conoscopy** (a method used in polarized light microscopy). In this system, the entire angular spectrum of light that passes through or reflects from the sample is collected and displayed in the focal plane.
* **What does it represent?**: Each point in the K-space corresponds to a specific direction of light incidence in the system. The map shows how light behaves when it hits the material at various angles.

**Polar Maps:**

* **What is it?**: A polar map, unlike the conoscopic map, is created by physically rotating the sample and measuring its optical response at different azimuths.
* **Where is it used?**: Polar maps are commonly used in **ellipsometry**, where the sample is rotated and light is shone at different incident angles to determine the optical constants (like refractive index and extinction coefficient) and thickness of layers.
* **What does it represent?**: Instead of varying the incoming light direction like in conoscopic maps, polar maps alter the orientation of the sample with respect to the light.

**3. Why Both Maps are Useful**

* **Conoscopic Maps** are useful in systems where the light's incidence is fixed, but the full range of angles in a single focal plane is analyzed. This is common in microscopes or imaging systems.
* **Polar Maps** are useful when the physical orientation of the material itself changes relative to the light source, which is common in characterization techniques like **spectroscopic ellipsometry**.

In systems with minimal diffraction, both types of maps contain equivalent information about the material's optical properties. In such cases, you can mathematically convert a polar map into a conoscopic map and vice versa using simple rotation matrices.

**4. The Mueller Matrix in K-space Maps**

The **Mueller matrix** is a 4x4 matrix that describes how light interacts with a material. It is a comprehensive description of the optical properties of a medium, including polarization changes.

* In **K-space maps**, the Mueller matrix is calculated for each combination of wave vectors (which correspond to different angles of incidence and azimuthal angles) rather than just one angle of incidence.
* This results in a **grid** of Mueller matrices, each describing how light is transmitted or reflected from the material at that particular angle.

For example, if you're analyzing how a polarizing film affects light, you can calculate the Mueller matrix at various incident angles to get a full picture of its behavior across the angular spectrum.

**5. How K-space Maps Are Calculated**

In this section, two primary functions—**mmBerremanMap** and **mmPartialWaveMap**—are introduced. These are extensions of the Berreman and Partial Wave methods described in earlier sections, but instead of computing the Mueller matrix for a single angle of incidence, they compute it across a grid of wave vectors.

**Inputs for Calculating K-Space Maps:**

* **Layer Array**: Describes the materials, thicknesses, orientations, and other properties of the layers in the multilayer stack.
* **Wavelengths**: The array of wavelengths at which the Mueller matrix is calculated. Typically, this will be a range, such as 300 nm to 800 nm.
* **Npts**: The number of grid points in X and Y, determining the resolution of the K-space map.
* **MaxAOI**: The maximum angle of incidence in degrees. This defines the outer boundary of the K-space map, as you can't calculate angles of incidence beyond this value.
* **bReflect**: A boolean flag that specifies whether the calculation is for reflection (true) or transmission (false).
* **bNorm**: A boolean flag that, when set to true, normalizes the Mueller matrices by their M11 element.
* **bConoscopic**: A boolean flag that specifies whether to calculate a conoscopic map (true) or a polar map (false).

**Outputs:**

The output of these functions is a 4x4xN×N×M array, where N is the number of grid points in X and Y, and M is the number of wavelengths. This array represents the Mueller matrix at each grid point and wavelength.

**6. Visualizing and Interpreting K-Space Maps**

When you generate K-space maps, the result is often displayed as a **2D plot** or a series of 2D plots. Each pixel in the plot corresponds to a particular angle of incidence and azimuth, and the color at that pixel represents the value of the Mueller matrix at that point.

* **Transmission Maps**: These maps show how much light is transmitted through the material at different angles and wavelengths.
* **Reflection Maps**: These maps show how much light is reflected by the material.

By analyzing these maps, you can gain insight into the material's optical properties, such as how it polarizes light, how it interacts with different wavelengths, and how these properties change with the angle of incidence.

**7. Examples and Applications:**

* In the example given in the document, K-space maps are simulated for a 300 µm thick section of **α-quartz**. The simulated data includes both **transmission** and **reflection** maps, computed using the Berreman and Partial Wave methods, respectively.
* The numerical aperture (NA) of the system is specified as **0.85**, which defines the range of angles that the system can collect.
* Such maps are critical in studying materials with complex optical properties, such as birefringent crystals (like quartz), polarizing films, or metamaterials.

**8. Applications of K-Space Maps in Research**

K-space maps are extensively used in materials science, photonics, and optics research. Common applications include:

* **Characterization of birefringent materials**: Materials that have different refractive indices depending on the direction of light can be studied using K-space maps.
* **Design of optical coatings and filters**: By understanding how light interacts with multilayer structures, researchers can design coatings that reflect or transmit light at specific angles or wavelengths.
* **Metamaterials and photonic crystals**: K-space maps are vital for studying artificial materials that have been engineered to control light in novel ways.

**Conclusion:**

**K-space maps** provide a rich, multidimensional view of how light interacts with materials across a range of angles and wavelengths. By using **conoscopic** or **polar maps**, researchers can obtain comprehensive data about a material's optical properties. These maps are essential tools in modern optics, allowing for detailed analysis and design of advanced optical materials and devices.

In the context of **K-space maps**, the angles **theta (θ)** and **phi (φ)** play crucial roles in defining the direction of light incidence on the material surface. Let’s explore these angles in greater detail to understand how they are used in K-space calculations and what they represent physically.

**1. Theta (θ): Polar Angle or Angle of Incidence**

* **What is Theta (θ)?**:
  + Theta (θ) is the **polar angle** or the **angle of incidence**. It represents the angle between the **incident light ray** and the **surface normal** (the perpendicular line to the material surface).
  + Mathematically, theta is often written as θ\thetaθ, and it ranges from **0° to 90°**:
    - **0°** means light is hitting the surface at **normal incidence** (perpendicular to the surface).
    - **90°** means light is hitting the surface at a **grazing incidence** (parallel to the surface).
* **Physical Meaning**:
  + When light strikes a surface, its behavior (reflection, refraction, or transmission) depends heavily on the **angle of incidence**. For example:
    - At **normal incidence** (θ = 0°), light tends to behave more predictably, with fewer polarization effects.
    - At more **oblique angles** (θ approaching 90°), light can undergo more complex interactions, such as greater reflection, polarization changes, and optical phase shifts.
* **Role in K-Space Maps**:
  + In a **K-space map**, the **radius** of the map often represents sin(θ), which increases as the angle of incidence increases. The further out you go from the center of the map, the larger the angle of incidence becomes.
  + In a **conoscopic map**, theta determines how far from the center a point lies, with the center corresponding to **normal incidence** (θ = 0°) and the outer edge representing a large angle of incidence (e.g., θ = MaxAOI).

**2. Phi (φ): Azimuthal Angle**

* **What is Phi (φ)?**:
  + Phi (φ) is the **azimuthal angle**, representing the rotation of the incident light around the surface normal (the z-axis).
  + It is the angle in the **XY-plane**, measuring how the incoming wave vector is rotated relative to a reference direction, typically the x-axis. The azimuthal angle ranges from **0° to 360°**.
* **Physical Meaning**:
  + The azimuthal angle defines the **direction from which the light is coming** relative to the sample’s surface:
    - **0°** might correspond to light coming from the x-axis.
    - **90°** corresponds to light coming from the y-axis.
    - **180°** is light coming from the opposite direction (relative to 0°).
  + In practical terms, varying φ helps to investigate the **symmetry** or **anisotropy** of a material. For example, certain materials might reflect or transmit light differently depending on the azimuthal angle due to their anisotropic properties (such as crystals).
* **Role in K-Space Maps**:
  + In a **K-space map**, phi determines the **angular position** around the center of the map. As you move around in a circular path in the map, you're changing the azimuthal angle.
  + The map shows how the material behaves as you rotate the direction of the incident light (azimuth) while keeping the angle of incidence (θ) fixed.

**3. Relationship Between Theta (θ) and Phi (φ)**

Together, theta and phi provide a full **spherical coordinate system** for defining the direction of light in 3D space.

* **Theta (θ)** determines how much the light ray is tilted from the normal to the surface (the z-axis).
* **Phi (φ)** determines the rotation of the light ray around the surface normal.

In optical simulations, both angles are needed to describe the full geometry of light incidence:

* **Theta (θ)** gives the elevation of the incoming light.
* **Phi (φ)** gives the direction of the light within the horizontal plane.

In practical terms, when creating K-space maps, you compute how the light interacts with the material for various combinations of **theta** and **phi**, resulting in a 2D plot where:

* The **radial distance** represents sin(θ) (related to the angle of incidence).
* The **angular position** around the plot represents ϕ (the azimuthal angle).

**4. Use of Theta and Phi in K-Space Calculations**

* **In Conoscopic Maps**:
  + These maps are typically plotted in polar coordinates, where:
    - **Theta (θ)** defines the radial distance from the center (closer to the edge means larger angles of incidence).
    - **Phi (φ)** defines the angular position around the center (different azimuthal directions).
  + A conoscopic map shows how the Mueller matrix (or other optical property) changes as a function of both θ and φ.
* **In Polar Maps**:
  + These maps also rely on **theta** and **phi**, but they may involve physically rotating the sample to vary the azimuthal angle (φ). The Mueller matrix is measured at different angles of incidence (θ) and azimuthal angles (φ).

**5. Visual Representation in K-Space Maps**

To visualize this:

* Imagine a **circle** (the K-space map). The center of the circle corresponds to **normal incidence** (θ= 0°).
* Moving outward from the center represents increasing θ (more oblique incidence).
* Moving **around** the circle changes φ, rotating the direction of the incident light.
* A **K-space map** for a material could show how its optical properties change as both θ and φ vary, revealing patterns related to the material’s structure, anisotropy, or other optical characteristics.

**6. Example:**

Let’s say we are studying light interacting with a thin film using a K-space map:

* If **θ = 0°** (normal incidence), the light is coming straight down onto the film. The map’s center would represent this.
* As θ increases, the light comes at more oblique angles. The points farther from the center of the map represent these more oblique angles.
* **Phi (φ)** might start at 0° when the light comes from the x-axis direction. As φ increases, the light’s azimuth rotates around the z-axis (the normal to the surface), sweeping out a full circle if φ goes from 0° to 360°.

**Summary:**

* **Theta (θ)**: Polar or angle of incidence, controlling how much the light ray is tilted relative to the surface normal.
* **Phi (φ)**: Azimuthal angle, controlling the direction of the light ray in the plane of the surface.
* In a **K-space map**, these angles are used together to map how light interacts with a material across a range of incident directions. Theta influences the radial position in the map, and phi influences the angular position around the center.

A screenshot of a graph

Description automatically generated

In the image provided, **theta (θ)** and **phi (φ)** are used as polar coordinates to describe the direction of light incidence in both **polar** and **conoscopic** coordinate systems. Let's break down their roles and how they appear in the plots.

**Theta (θ):**

* **What it represents**: Theta (θ) is the **polar angle** or **angle of incidence**. It represents how much the incoming light is tilted away from the normal (the z-axis, perpendicular to the surface).
* **How it is visualized**:
  + In both the **Polar** and **Conoscopic** maps, the **radial distance** from the center of the plot corresponds to **theta**. The center of the plot (the origin) represents **normal incidence** (θ = 0°), and as you move outward, theta increases. The outermost points on the grid correspond to the maximum angle of incidence, which is determined by the system's **Numerical Aperture (NA)**.
  + In the diagrams above the plots, theta is marked as radiating outward from the center in both the polar and conoscopic coordinate systems. For example, in both coordinate systems, the concentric circles represent increasing theta, with the outermost circle corresponding to the maximum θ allowed by the NA of the system.

**Phi (φ):**

* **What it represents**: Phi (φ) is the **azimuthal angle**, which defines the **rotation** of the incident light around the surface normal (z-axis). Essentially, it describes the direction from which the light is coming relative to a reference direction (usually the x-axis).
* **How it is visualized**:
  + In both **Polar** and **Conoscopic** maps, the azimuthal angle **phi** is shown as the angle around the center of the plot (the angular position in the circular grid).
  + The **phi** angles are labeled around the perimeter of the polar plots, starting from 0° on the right and increasing counterclockwise (e.g., 90° at the top, 180° on the left, 270° at the bottom, and back to 0°/360° at the right).

**Polar vs. Conoscopic Representation:**

* **Polar Maps**: In these maps, the specimen is rotated physically, so the **phi (azimuthal angle)** varies as the sample rotates. This type of map is often used in **ellipsometry**.
  + The azimuthal rotation is shown in the coordinate system diagram, and phi changes as the specimen rotates.
* **Conoscopic Maps**: In these maps, the specimen remains stationary, and instead, the incident light angles vary across the wave vector. These maps are typically generated in **back-focal plane imaging** systems, where the entire range of incident angles (theta and phi) can be captured simultaneously.
  + In this case, the light is varied across the wavevector space without rotating the sample itself.

**Key Differences Between Polar and Conoscopic Maps:**

* In the **polar coordinate system**, the specimen is rotated to measure the optical response at different azimuthal angles. The K-space map then shows how the sample behaves as you change the angle of incidence (theta) and rotate the azimuth (phi).
* In the **conoscopic coordinate system**, the sample remains stationary, and the incident light is analyzed at different angles (theta and phi) without rotating the sample. Here, the azimuthal angle (phi) corresponds to different directions of the incident light in the focal plane, without rotating the material.

**Transmission and Reflection Maps:**

* In the **Transmission** K-space maps, the top set of images in the grid shows how the material transmits light at various angles of incidence (theta) and azimuths (phi). The patterns seen in the concentric rings and spots indicate how transmission varies across the angular spectrum.
* In the **Reflection** K-space maps (bottom set of images), similar to transmission maps, the patterns reflect how light is reflected at different angles of incidence (theta) and azimuths (phi). The color and intensity of the spots indicate the reflective behavior of the material for various combinations of theta and phi.

**Conclusion:**

* **Theta (θ)** corresponds to the radial distance from the center of the map, with 0° (normal incidence) at the center and increasing theta values moving outward.
* **Phi (φ)** corresponds to the angular position around the center, representing the direction from which the light is incident (azimuth). It ranges from 0° to 360° in both coordinate systems, but in the polar map, phi varies by physically rotating the sample, while in the conoscopic map, it varies by changing the light's direction without moving the sample.

In both cases, these angles together describe the full 3D direction of the incoming light, which is crucial for understanding the material’s optical behavior.

The provided function **mmBerremanMap** implements a **K-space map** calculation for multilayer optical systems using the Berreman method. This function is designed to compute the Mueller matrix for a grid of in-plane wave vectors over a range of angles of incidence (in K-space). Let’s break it down step-by-step to understand how it works.

Purpose:

The function computes Mueller matrices in K-space for different wavelengths and angles of incidence, allowing for either reflection or transmission configurations. The results can be used to generate conoscopic or polar maps, depending on the setting of the bConoscopic flag.

**Inputs:**

1. **layerArray:**
   * **A cell array that defines the multilayer structure, with information about each layer (material, thickness, optical properties).**
2. **wavelengths:**
   * **An array of wavelengths (in nanometers) for which the Mueller matrix is to be calculated.**
3. **Npts:**
   * **The number of grid points in the K-space map (both X and Y directions). This defines the resolution of the map.**
4. **maxAOI:**
   * **The maximum angle of incidence (in degrees) that defines the boundary of the K-space map. This determines the range of angles to compute.**
5. **bReflect:**
   * **A boolean value to specify whether the calculation is for reflection (true) or transmission (false).**
6. **bNorm:**
   * **A boolean value to specify whether the output Mueller matrix should be normalized by its first element (M11).**
7. **bConoscopic:**
   * **A boolean value to specify whether to compute a conoscopic map (true) or a polar map (false).**

**Outputs:**

* **MM:**
  + **The output is a 4x4xNptsxNptsxN array (Mueller matrix) where Npts is the number of grid points in X and Y, and N is the number of wavelengths.**

**Step-by-Step Explanation:**

**1. Initialization:**

Nlayers = size(layerArray,2);

A = [1,0,0,1;1,0,0,-1;0,1,1,0;0,1i,-1i,0];

Ainv = [0.5,0.5,0,0;0,0,0.5,-0.5\*1i;0,0,0.5,0.5\*1i;0.5,-0.5,0,0];

 **Nlayers**: The number of layers in the layerArray.

 **A** and **Ainv**: Transformation matrices used in the Berreman method for converting between different representations of the electric field.

**2. Generate the K-space Grid:**

kmax = sin(maxAOI\*pi/180);

kx\_t = [0;0];

idx = [0;0];

kmax = 2\*kmax/Npts;

 **kmax**: The maximum value of the in-plane wave vector components based on the maximum angle of incidence (maxAOI), converted from degrees to radians.

 A grid of K-space vectors (kx, ky) is created using **Npts** to divide the space. The maximum wave vector is divided by Npts to generate a regular grid.

for X = 1:Npts

for Y = 1:Npts

r = sqrt((X-Npts/2).^2 + (Y-Npts/2).^2);

if r <= Npts/2

kx\_t = [kx\_t , [r\*kmax; atan2(X-Npts/2, Y-Npts/2)]];

idx = [idx, [X;Y]];

end

end

end

 This loop creates a grid of **wave vectors (kx, ky)** in the in-plane direction of the surface. The radius r ensures that only wave vectors within the valid range (inside the circle defined by maxAOI) are included.

 **kx\_t** stores the wave vectors (r, φ), where r is the magnitude of the wave vector and φ is the azimuthal angle (calculated using atan2).

 **idx** stores the corresponding grid positions (X, Y) in the 2D map.

3. **Calculate the Initial Transfer Matrix for the Ambient Medium**:

[Psi0, n0] = psiAmbientInvMap(layerArray{1}, kx\_t, wavelengths);

The initial transfer matrix **Psi0** and refractive index **n0** for the **incident medium** are computed using the **psiAmbientInvMap** function.

4. **Calculate the Exit Transfer Matrix**:

if layerArray{Nlayers}{5} == 0

Psi2 = psiAnisoMap(layerArray{Nlayers}, wavelengths, kx\_t, n0);

elseif layerArray{Nlayers}{2} == Inf

Psi2 = psiIsoMap(layerArray{Nlayers}, wavelengths, kx\_t, n0);

else

Psi2 = psiAmbientMap(layerArray{Nlayers}, kx\_t, wavelengths);

End

The function determines if the **last layer** is anisotropic or isotropic, or if it is a semi-infinite medium. Based on this, it calculates the transfer matrix **Psi2** for the exit medium using the appropriate function.

5. **Multiply Transfer Matrices for Intermediate Layers**:

if Nlayers > 2

for k = (Nlayers-1):-1:2

if layerArray{k}{5} == 0

Psi2 = multiprod(layerBerremanMap(layerArray{k}, wavelengths, kx\_t, n0), Psi2);

else

Psi2 = multiprod(layerBerremanIsoMap(layerArray{k}, wavelengths, kx\_t, n0), Psi2);

end

end

end

This loop handles **intermediate layers** between the first and last layers. It computes their transfer matrices using either the **anisotropic** (layerBerremanMap) or **isotropic** (layerBerremanIsoMap) methods, depending on the layer's properties, and multiplies them with **Psi2** to build up the final transfer matrix.

6. **Calculate the Full Transfer Matrix**:

Psi2 = multiprod(Psi0, Psi2);

The full transfer matrix is computed by multiplying the initial transfer matrix **Psi0** with the accumulated transfer matrix **Psi2**.

**7. Compute Reflection or Transmission:**

if bReflect

J = multiprod(Psi2([3 4], [1 2], :, :, :), invert2x2(Psi2([1 2], [1 2], :, :, :)));

else

J = invert2x2(Psi2([1 2], [1 2], :, :, :));

End

Depending on whether reflection or transmission is being calculated (bReflect), the appropriate parts of the transfer matrix are used to compute **J**, which represents the reflection or transmission matrix.

8. **Convert to Mueller Matrix**:

temp = real(multiprod( multiprod(A, bigKron(J)), Ainv));

The **Mueller matrix** is calculated by converting **J** using the transformation matrices **A** and **Ainv**.

9. **Normalize if Required**:

if bNorm

temp = temp ./ temp(1,1,:,:);

end

If **normalization** is requested (bNorm = true), the Mueller matrix is normalized by its first element (M11).

10. **Apply Conoscopic Rotation if Required**:

if bConoscopic

if bReflect

for i=1:length(wavelengths)

temp(:,:,:,i) = mmRotateRefl(temp(:,:,:,i), -kx\_t(2,:));

end

else

for i=1:length(wavelengths)

temp(:,:,:,i) = mmRotate(temp(:,:,:,i), -kx\_t(2,:));

end

end

end

If a **conoscopic map** is requested (bConoscopic = true), the Mueller matrix is rotated according to the azimuthal angle **kx\_t(2,:)** to account for the direction of the incident light in conoscopic systems.

11. **Assemble the Full Mueller Matrix for the K-space Map**:

MM = zeros(4, 4, Npts, Npts, length(wavelengths));

for i=1:size(kx\_t, 2)

MM(:, :, idx(1,i), idx(2,i), :) = temp(:,:,i,:);

End

The computed Mueller matrix is inserted into its corresponding position in the K-space map. The grid points **idx** are used to map each calculated matrix to the correct (X, Y) position in the output array **MM**.

**Summary:**

* **mmBerremanMap** calculates a K-space map of Mueller matrices for a multilayer structure over a grid of in-plane wave vectors (angles of incidence) and wavelengths.
* The function supports both **reflection** and **transmission** calculations and can output either **conoscopic** or **polar maps** based on the input flags.
* This function is part of a broader toolset for analyzing the optical properties of multilayer systems using the **Berreman method**.

% Define the layer structure (air -> TiO2 -> glass)

layerArray{1} = {'air', 0, [0 0 0], 0, 1}; % Incident medium (air)

layerArray{2} = {'TiO2', 300, [0 0 0], 0, 0}; % TiO2 layer, 300 nm thick

layerArray{3} = {'glass', Inf, [0 0 0], 0, 1}; % Glass substrate (semi-infinite)

% Define wavelengths (in nm)

wavelengths = 500; % Single wavelength at 500 nm (you can also use a range like 400:10:800)

% Set the grid size and maxAOI

Npts = 100; % 100x100 grid points

maxAOI = 60; % Maximum angle of incidence of 60 degrees

% Set the flags for reflection and normalization

bReflect = false; % Set to false for transmission

bNorm = true; % Normalize the Mueller matrix

% Set whether to compute a conoscopic map

bConoscopic = false; % Set to false for a polar map (you can set it to true for conoscopic)

% Call the mmBerremanMap function to compute the Mueller matrix

MM = mmBerremanMap(layerArray, wavelengths, Npts, maxAOI, bReflect, bNorm, bConoscopic);

% Now plot all the elements of the Mueller matrix in K-space

% Create a figure to hold the subplots for all 16 Mueller matrix elements

figure;

% Loop through each element of the 4x4 Mueller matrix

for i = 1:4

for j = 1:4

% Extract the (i,j) element of the Mueller matrix at the first wavelength

Mij = squeeze(MM(i, j, :, :, 1)); % Extract the (i,j) element

% Create a subplot for each element

subplot(4, 4, (i-1)\*4 + j);

% Plot the element using imagesc

imagesc(Mij);

axis equal;

colorbar;

% Set title for each subplot

title(['M' num2str(i) num2str(j)]);

% Remove axis labels for a cleaner look

set(gca, 'XTick', [], 'YTick', []);

end

end

% Add a global title for the entire figure

sgtitle('All 16 Elements of the Mueller Matrix (Transmission)');

**Explanation:**

1. **Layer Structure**: We use a simple three-layer structure (air -> TiO2 -> glass), as before.
2. **Wavelength**: We use a single wavelength at 500 nm. You can modify this to use a range of wavelengths if desired.
3. **Grid Resolution and maxAOI**: We use 100x100 grid points (Npts = 100) and a maximum angle of incidence of 60 degrees (maxAOI = 60°).
4. **Mueller Matrix Calculation**: The mmBerremanMap function is called to compute the Mueller matrix over the K-space grid for the given wavelength.
5. **Plotting**:
   * We loop over all 16 elements of the 4x4 Mueller matrix.
   * For each element, we use imagesc to plot the corresponding element as a 2D image.
   * The results are arranged in a 4x4 grid of subplots.
   * Each subplot is labeled with the corresponding element of the Mueller matrix (M11, M12, M21, etc.).

**Result:**

This code generates a **4x4 grid of plots**, where each plot corresponds to one element of the **Mueller matrix** (M11 to M44). The plot shows how each element of the Mueller matrix varies across K-space (over a range of angles of incidence).

* **M11, M12, M13, ..., M44**: Each subplot represents one of the 16 elements of the Mueller matrix at a specific wavelength and shows the variation of that element over the K-space grid.

**Customization:**

* **Wavelength**: You can modify the wavelengths variable to include a range of wavelengths, and then loop over them to generate a sequence of Mueller matrix plots for different wavelengths.
* **Conoscopic vs. Polar Map**: You can change bConoscopic = true to compute a conoscopic map instead of a polar map.

This code provides a complete overview of the Mueller matrix elements over K-space, helping you understand how each element behaves as a function of the angle of incidence and azimuth.

The **mmPartialWaveMap** function calculates the **Mueller matrix** for a multilayer structure over a grid of wave vectors in **K-space** using the **Partial Wave method**. This method is designed for systems that contain one **thick layer** (in addition to other thin layers). The function allows for the generation of either **transmission** or **reflection** maps over the **K-space** grid and provides the option to plot the results using a user-defined function.

Let’s break down the key components of this function and how it works.

**Function Overview:**

* **Inputs**:
  1. **layerArray**: A cell array describing the layers of the multilayer system. Each element includes details such as material properties, thickness, and anisotropy.
  2. **wavelengths**: Array of wavelengths (in nanometers) for which the Mueller matrix will be calculated.
  3. **Npts**: Number of grid points in both X and Y directions of the K-space map (resolution of the grid).
  4. **maxAOI**: Maximum angle of incidence (in degrees). This determines the boundary of the K-space map.
  5. **bReflect**: Boolean flag to indicate if the computation is for **reflection** (true) or **transmission** (false).
  6. **bNorm**: Boolean flag to indicate if the Mueller matrix should be normalized by its M11 element.
  7. **bConoscopic**: Boolean flag to indicate whether to compute a **conoscopic** map (true) or a **polar** map (false).
  8. **varargin**: Additional optional input arguments for plotting.
* **Outputs**:
  1. **MM**: The calculated Mueller matrix over the K-space grid, a 5D array of dimensions [4, 4, Npts, Npts, length(wavelengths)].
  2. **MPlot\_object**: An object returned by a user-defined plotting function, if provided in varargin.

**Steps in the Code:**

**1. Initialization:**

Nlayers = size(layerArray,2);

thick = zeros(Nlayers,1);

for k=1:Nlayers

thick(k) = layerArray{k}{4};

end

g = find(thick == true);

 **Nlayers**: The number of layers in the layerArray.

 **thick**: A vector that checks each layer's thickness. The flag **true** marks the thick layer in the system.

 **g**: Index of the thick layer in the system.

**2. Generate the K-space Grid:**

kmax = sin(maxAOI\*pi/180);

kx\_t = [0;0];

idx = [0;0];

kmax = 2\*kmax/Npts;

for X = 1:Npts

for Y = 1:Npts

r = sqrt((X-Npts/2).^2 + (Y-Npts/2).^2);

if r <= Npts/2

kx\_t = [kx\_t , [r\*kmax; atan2(X-Npts/2, Y-Npts/2)]];

idx = [idx, [X;Y]];

end

end

end

kx\_t = kx\_t(:, 2:end);

idx = idx(:, 2:end);

 **kmax**: Maximum in-plane wave vector determined by the **maximum angle of incidence (maxAOI)**, converted from degrees to radians.

 The loop generates the grid of **wave vectors (kx, ky)** over a 2D space defined by **Npts** (grid resolution). The wave vectors are stored in **kx\_t**, and their corresponding grid locations are stored in **idx**.

* **kx\_t** stores the wave vectors in polar coordinates: r (magnitude) and phi (azimuthal angle).
* **idx** stores the corresponding X, Y indices in the grid.

3. **Initial Transfer Matrix Calculation for the Ambient Medium**:

[Psi0, n0] = psiAmbientMap(layerArray{1}, kx\_t, wavelengths);

This computes the initial transfer matrix **Psi0** and refractive index **n0** for the incident medium (typically air).

4. **Transfer Matrix for the Exit Medium**:

if layerArray{Nlayers}{5} == 0

Psi2 = psiAnisoMap(layerArray{Nlayers}, wavelengths, kx\_t, n0);

elseif layerArray{Nlayers}{2} == Inf

Psi2 = psiIsoMap(layerArray{Nlayers}, wavelengths, kx\_t, n0);

else

Psi2 = psiAmbientMap(layerArray{Nlayers}, kx\_t, wavelengths);

End

The function computes the transfer matrix **Psi2** for the **exit medium** (the last layer in the system).

* If the exit medium is anisotropic, it calls **psiAnisoMap**.
* If it’s isotropic or semi-infinite, it calls **psiIsoMap** or **psiAmbientMap** accordingly.

5. **Multiply Transfer Matrices for Thin Layers Before the Thick Layer**:

if g > 2

for m = 2:(g-1)

layerArray{m}{2} = -layerArray{m}{2}; %change sign of d to invert layer matrix

if layerArray{m}{5} == 0

Psi0 = multiprod(layerBerremanMap(layerArray{m}, wavelengths, kx\_t, n0), Psi0);

else

Psi0 = multiprod(layerBerremanIsoMap(layerArray{m}, wavelengths, kx\_t, n0), Psi0);

end

end

end

If there are **thin layers before the thick layer** (those with g > 2), the function computes their transfer matrices and multiplies them into **Psi0**. It uses:

* **layerBerremanMap** for anisotropic layers.
* **layerBerremanIsoMap** for isotropic layers.

6. **Multiply Transfer Matrices for Thin Layers After the Thick Layer**:

if Nlayers - g > 1

for m = (Nlayers-1):-1:(g+1)

if layerArray{m}{5} == 0

Psi2 = multiprod(layerBerremanMap(layerArray{m}, wavelengths, kx\_t, n0), Psi2);

else

Psi2 = multiprod(layerBerremanIsoMap(layerArray{m}, wavelengths, kx\_t, n0), Psi2);

end

end

end

If there are **thin layers after the thick layer**, their transfer matrices are computed and multiplied into **Psi2**.

7. **Partial Wave Calculation**:

temp = partialWaveMap(Psi0, Psi2, layerArray{g}, wavelengths, kx\_t, n0, bReflect);

The **Partial Wave method** is used to compute the Mueller matrix for the system. This method handles the thick layer differently from the Berreman method. The function **partialWaveMap** performs this calculation using the transfer matrices for the ambient and exit media (**Psi0** and **Psi2**) and the properties of the thick layer.

8. **Normalization (Optional)**:

if bNorm

temp = temp ./ temp(1,1,:,:);

end

If **normalization** is enabled (bNorm = true), the Mueller matrix is normalized by its first element **M11**.

9. **Conoscopic Map Rotation (Optional)**:

if bConoscopic

if bReflect

for i=1:length(wavelengths)

temp(:,:,:,i) = mmRotateRefl(temp(:,:,:,i), -kx\_t(2,:));

end

else

for i=1:length(wavelengths)

temp(:,:,:,i) = mmRotate(temp(:,:,:,i), -kx\_t(2,:));

end

end

end

If a **conoscopic map** is requested (bConoscopic = true), the Mueller matrix is rotated according to the azimuthal angle to account for the direction of the incoming light in conoscopic imaging systems.

10. **Construct the Final Mueller Matrix**:

MM = zeros(4, 4, Npts, Npts, length(wavelengths));

for i=1:size(kx\_t, 2)

MM(:, :, idx(1,i), idx(2,i), :) = temp(:,:,i,:);

End

The calculated Mueller matrix is placed into a 4x4xNptsxNptsxN array **MM**, which stores the full Mueller matrix over the grid points in K-space.

11. **Plotting (Optional)**:

if ~isempty(varargin)

fun = varargin{1};

MPlot\_object = fun(MM, varargin{2:end});

End

* If a plotting function is provided as part of the **varargin** input, this function is called to generate plots or visualizations of the computed Mueller matrix **MM**. The result is stored in **MPlot\_object**.

**Summary:**

* **mmPartialWaveMap** calculates a K-space map of Mueller matrices for a multilayer system with one thick layer, using the **Partial Wave method**.
* The function can handle both transmission and reflection calculations and supports optional normalization and conoscopic map rotations.
* The output includes the full 4x4 Mueller matrix over a grid of angles of incidence and azimuth, and optionally allows for plotting using user-defined functions.

**Section 3** in the document appears to focus on **Berreman and Partial Wave Mueller Matrix Map Methods**, diving into the technical details of how the Mueller matrices are computed for layered structures. Let's explain this section in an **exhaustive manner**, breaking down the key concepts involved.

**Overview of the Section**

This section describes two methods used to compute **Mueller matrices** for optical systems that consist of **multiple layers**: the **Berreman method** and the **Partial Wave method**. Both methods aim to model how light interacts with these layers, but they do so using different mathematical approaches. The section also explains how these methods can be extended to compute **Mueller matrix maps** in **K-space** over a range of angles of incidence.

**1. Berreman Method**

* The **Berreman method** is a well-established technique for modeling light propagation through **anisotropic media** (materials that have different properties in different directions).
* **Core idea**: The method is based on solving the **Berreman 4x4 matrix equations**, which describe how light (in the form of electromagnetic waves) propagates through stratified (layered) media. Each layer in the system is characterized by a **4x4 matrix** that describes how the light's electric and magnetic fields are transformed as they pass through the layer.

**Key Points:**

* **Electromagnetic Field Propagation**: The Berreman method solves for the electric and magnetic fields as light propagates through each layer. These fields are represented in a **4x4 matrix** form.
* **Anisotropy**: The method is particularly useful for **anisotropic materials**, where the refractive index depends on the direction of the light. This is common in crystals, where light behaves differently along different crystal axes.
* **Transfer Matrices**: For each layer, the method computes a **transfer matrix** that describes how the light changes as it passes through the layer. The transfer matrices of all the layers are then multiplied together to get the overall transfer matrix for the entire multilayer system.
* **Mueller Matrix Calculation**: Once the overall transfer matrix is computed, the **Mueller matrix** (which describes the polarization state of light) is derived from it. The Mueller matrix provides a full description of how the layered system transforms the polarization of light.

**K-Space Mapping in the Berreman Method:**

* **K-space maps** are generated by calculating the Mueller matrix for a grid of **wave vectors** (or angles of incidence) and azimuthal angles.
* The Berreman method allows for the **conoscopic** and **polar maps** discussed earlier, where the light's direction is varied to cover a range of angles.

**2. Partial Wave Method**

The **Partial Wave method** is another approach for calculating the Mueller matrix in stratified media. It differs from the Berreman method in how it handles the computation of light propagation through the system.

**Key Points:**

* **Thick and Thin Layers**: The Partial Wave method is especially useful when the system contains a **thick layer** sandwiched between **thin layers**. In this case, the Berreman method can be computationally inefficient, so the Partial Wave method is preferred.
* **Core idea**: This method breaks the problem into smaller pieces, computing the propagation of light through **thin layers** separately from the propagation through the **thick layer**. This leads to more efficient calculations in systems with mixed thin and thick layers.
* **Transfer Matrices for Thin Layers**: For thin layers, the method computes the transfer matrices in a manner similar to the Berreman method.
* **Partial Wave Solution for Thick Layers**: For the thick layer, the Partial Wave method computes a different kind of solution, one that is more efficient for layers with large thicknesses. This is the key difference between the Berreman and Partial Wave methods.

**Advantages of the Partial Wave Method:**

* The Partial Wave method is more efficient when there is at least one **thick layer** in the system. Thick layers can introduce **computational bottlenecks** in the Berreman method, as the transfer matrix can become large and difficult to handle. The Partial Wave method overcomes this by treating the thick layer separately.

**3. Mueller Matrix Maps**

Both the Berreman and Partial Wave methods can be extended to generate **Mueller matrix maps** in **K-space**.

**K-Space Maps:**

* **K-space** refers to the space of **wave vectors** (or angles of incidence) that describe the direction of incoming light.
* **Mueller matrix maps** are generated by calculating the Mueller matrix at multiple points in K-space, which correspond to different combinations of angles of incidence and azimuthal angles.
* These maps show how the material's optical properties (such as polarization changes) vary with the angle of incidence and azimuth.

**Transmission and Reflection:**

* The Mueller matrix can be computed for both **transmission** and **reflection** cases, depending on whether the light is transmitted through the layered structure or reflected back from it.

**4. Numerical Considerations**

* **Numerical Aperture (NA)**: The document likely discusses the **numerical aperture** of the system, which determines the range of angles that can be captured in the K-space map. A higher NA allows for a wider range of angles to be included in the map.
* **Resolution**: The number of **grid points (Npts)** in the K-space map determines the resolution of the map. Higher resolution requires more computational power but provides more detailed information about how the system behaves over different angles of incidence.

**Conclusion:**

* The **Berreman method** is a powerful tool for calculating the optical properties of **anisotropic multilayer systems**, especially when the layers are **thin**.
* The **Partial Wave method** is more efficient for systems that include **thick layers**, making it the preferred method for mixed systems with both thick and thin layers.
* Both methods can be used to compute **Mueller matrix maps** over **K-space**, providing a comprehensive view of how the system behaves over a range of angles of incidence and azimuths.

This section, **Section 3**, provides the theoretical foundation for the subsequent computations of Mueller matrices in various scenarios. It introduces the underlying methods, explains when to use each method, and sets the stage for the practical applications that follow.

**3.1 MPlot**

**MPlot** class is designed to facilitate the plotting of spectral Mueller matrix data as a 4x4 grid of line plots. This class simplifies the visualization of Mueller matrices for various purposes, such as comparing data at different angles of incidence or different wavelengths.

Here's an in-depth breakdown of its components and functionalities:

**Constructor (pObj = MPlot(wavelengths , mmData , mmError , lineSpec , PROPERTIES);)**

* **wavelengths**: A numeric array representing the wavelengths at which the Mueller matrices were computed. It is used as the x-axis in the plots.
* **mmData**: A 4x4xN-dimensional array representing the Mueller matrix data where N is the number of wavelengths. Each element of the Mueller matrix is plotted as a line plot.
* **mmError** (Optional): An array that holds the error values corresponding to the Mueller matrix data. This is useful when plotting error bars.
* **lineSpec** (Optional): A Matlab LineSpec (line specification) string that controls the appearance of the lines in the plot (e.g., line style, color, marker).
* **PROPERTIES**: These are optional **Name-Value pairs** that allow customization of the plot, such as the size, font size, axis properties, and more.

**Class Properties**

* **figHandle**: Handle to the figure object (read-only). This allows interaction with the figure after it is created.
* **axesHandles**: A 4x4 array of axis handles (read-only). Each of these corresponds to one subplot in the 4x4 grid of Mueller matrix elements.
* **size**: Sets the size of the figure in inches (default: [1000, 700]).
* **fontsize**: Sets the font size for all text in the figure (default: 12 points).
* **limy**: Sets the minimum allowed range for the y-axes of the plots (default: 1E-4).
* **title**: A string used as the title of the figure.
* **legend**: Configures a legend for the plot. It can accept either a numeric array or a cell array of strings.
* **axNV**: Name-Value pairs that apply to the **Axes Properties** to format the axes.
* **lineNV**: Name-Value pairs that apply to the **Line Properties** to format the plot lines.
* **vSpace**: Adds extra vertical space between plots (in pixels).
* **borderFactor**: Adds extra white space between the plot area and the figure borders. It is proportional to the largest line width on the plots.
* **ev**: If true, the x-axis is converted from nanometers to electronvolts for energy-based plots.

Let’s go into detail about the **class properties** of **MPlot** and how they contribute to the customization and visualization of spectral Mueller matrix data.

**1. figHandle (Protected)**

* **Type**: Handle to the figure object.
* **Description**: This property stores a reference to the figure window created by the MPlot class. The figure handle allows you to modify the figure after it has been created (e.g., changing its size, adding additional elements, etc.).
* **Usage**: When the MPlot object is initialized, this handle is used to control and manage the figure's properties, including size and position.

**2. axesHandles (Protected)**

* **Type**: 4x4 matrix of axes handles.
* **Description**: This property stores the handles to the 16 subplots (axes) created in the figure. Each handle corresponds to one element of the 4x4 Mueller matrix. This allows individual customization of each subplot (e.g., setting axis limits, labels, grid lines, etc.).
* **Usage**: The axes handles allow you to directly manipulate individual subplots. For example, you could use the axesHandles to change the title, gridlines, or axis labels of a specific subplot.

**3. size**

* **Type**: 2-element array (default: [1000, 700]).
* **Description**: This property sets the size of the figure window in pixels. The first element controls the width, and the second element controls the height of the figure.
* **Usage**: The size is passed during the initialization of the MPlot object to control how big or small the overall figure will appear. This is especially useful when producing plots for publication where specific figure dimensions are required.

**Example**:

matlab

Copy code

'size', [800, 500] % A figure size of 800x500 pixels.

**4. fontsize**

* **Type**: Scalar (default: 12).
* **Description**: This property controls the font size of all text in the figure, including axis labels, titles, legends, and tick labels.
* **Usage**: Adjusting this property allows you to change the text size across all subplots in the figure.

**Example**:

matlab

Copy code

'fontsize', 14 % Set font size to 14 points.

**5. limy**

* **Type**: Scalar (default: 1e-4).
* **Description**: This property sets the minimum allowed range for the y-axes across all subplots. If any subplot’s y-axis range is smaller than limy, the axis is adjusted to ensure that it meets the minimum range.
* **Usage**: This ensures that all y-axes in the subplots have a sufficient range, preventing axes from being too narrow and improving the readability of plots with very small data variations.

**Example**:

matlab

Copy code

'limy', 0.01 % Set the minimum y-axis range to 0.01.

**6. title**

* **Type**: String (default: empty).
* **Description**: This property sets the title of the entire figure. The title is placed at the top of the figure and gives a description of the content of the plot.
* **Usage**: Use the title property to give your figure a meaningful label. This is particularly useful in scientific visualizations where you want to describe what the plot represents.

**Example**:

matlab

Copy code

'title', '1 mm c-cut quartz in transmission' % Set the title of the figure.

**7. legend**

* **Type**: Cell array or numeric array (default: empty).
* **Description**: This property is used to create a legend for the plot. The first element of the cell array defines the label for the legend (e.g., 'AOI' for angle of incidence), and the second element specifies the values or labels corresponding to the plotted data.
* **Usage**: Legends help identify which data corresponds to different conditions, such as different angles of incidence, wavelengths, or other experimental parameters.

**Example**:

matlab

Copy code

'legend', {'AOI', [0 10 20]} % Create a legend for angles of incidence (0°, 10°, 20°).

**8. axNV (Axes Name-Value Pairs)**

* **Type**: Cell array (default: empty).
* **Description**: This property allows you to modify the properties of the axes (subplots) in the figure. These can be any valid name-value pairs accepted by MATLAB’s axes function. For example, you can change the color order for different plot lines, modify tick marks, set grid visibility, and more.
* **Usage**: Customize the appearance of the axes in the subplots, such as setting the color scheme or turning on grid lines.

**Example**:

matlab

Copy code

'axNV', {'ColorOrder', cbrewer('qual', 'Set2', 3)} % Set color order using ColorBrewer.

**9. lineNV (Line Name-Value Pairs)**

* **Type**: Cell array (default: empty).
* **Description**: This property allows you to modify the properties of the plot lines. These can be any valid name-value pairs accepted by MATLAB’s plot function, such as setting line width, line style, or marker type.
* **Usage**: Customize the appearance of the plot lines. For example, you can make the lines thicker, change their color, or modify the line style (solid, dashed, etc.).

**Example**:

matlab

Copy code

'lineNV', {'LineWidth', 2} % Set the line width to 2 points.

**10. vSpace**

* **Type**: Scalar (default: 0).
* **Description**: This property sets the vertical space between subplots in pixels. It allows you to create more separation between the rows of subplots.
* **Usage**: Useful when you have multiple subplots and want to avoid crowding them vertically.

**Example**:

matlab

Copy code

'vSpace', 10 % Add 10 pixels of vertical space between subplots.

**11. borderFactor**

* **Type**: Scalar (default: 0).
* **Description**: This property increases the white space (or padding) around the plot area in the figure. The value is a multiple of the largest line width in the figure, ensuring that the lines don't appear too close to the figure's edges.
* **Usage**: Helps ensure that the plotted lines are not too close to the borders of the plot area, improving the visual appearance and readability of the plot.

**Example**:

matlab

Copy code

'borderFactor', 3 % Add extra white space around the plot area.

**12. ev**

* **Type**: Boolean (default: false).
* **Description**: If this property is set to true, the x-axis is automatically converted from nanometers (wavelengths) to electronvolts (eV), which is another common unit for photon energy.
* **Usage**: Useful when you want to display the x-axis in units of electronvolts rather than nanometers.

**Example**:

matlab

Copy code

'ev', true % Convert the x-axis from wavelength (nm) to energy (eV).

**How These Properties Work Together:**

These class properties give you full control over how the spectral Mueller matrix data is visualized:

* **Figure size**: You can adjust the size to fit publication standards or screen dimensions.
* **Axes customization**: You can modify the appearance of axes (e.g., color order, grid lines) for better presentation.
* **Line customization**: You can change the appearance of lines to distinguish between data sets more clearly.
* **Title and legend**: You can add meaningful descriptions to your plot to make it self-explanatory.
* **Spacing and borders**: You can ensure the plot is not cramped by adjusting spacing between subplots and adding padding around the plot area.

These properties together make **MPlot** a highly customizable and flexible tool for visualizing spectral Mueller matrix data, particularly when working with multi-dimensional datasets such as those generated in optical experiments or quantum simulations.

% Clear workspace and command window

clc;

clear all;

% Step 1: Build the model

layerArray{1} = {'air', 0, [0 0 0], 0, 1}; % Incident medium (air)

layerArray{2} = {'+quartz', 1000000, [0 0 0], 1, 0}; % Quartz layer (1 mm thick)

layerArray{3} = {'air', 0, [0 0 0], 0, 1}; % Exit medium (air)

% Step 2: Calculate transmission Mueller matrix spectra

M0 = mmPartialWave(layerArray, 300:750, 0, 1, 0); % Angle of incidence 0 degrees

M10 = mmPartialWave(layerArray, 300:750, 10, 1, 0); % Angle of incidence 10 degrees

M20 = mmPartialWave(layerArray, 300:750, 20, 1, 0); % Angle of incidence 20 degrees

% Step 3: Plot the data using MPlot

pObj = MPlot(300:750, cat(4, M0, M10, M20), ...

'size', [800 500], ... % Set the figure size

'borderFactor', 3, 'vSpace', 10, ... % Adjust spacing

'title', '1 mm c-cut quartz in transmission', ... % Add a title

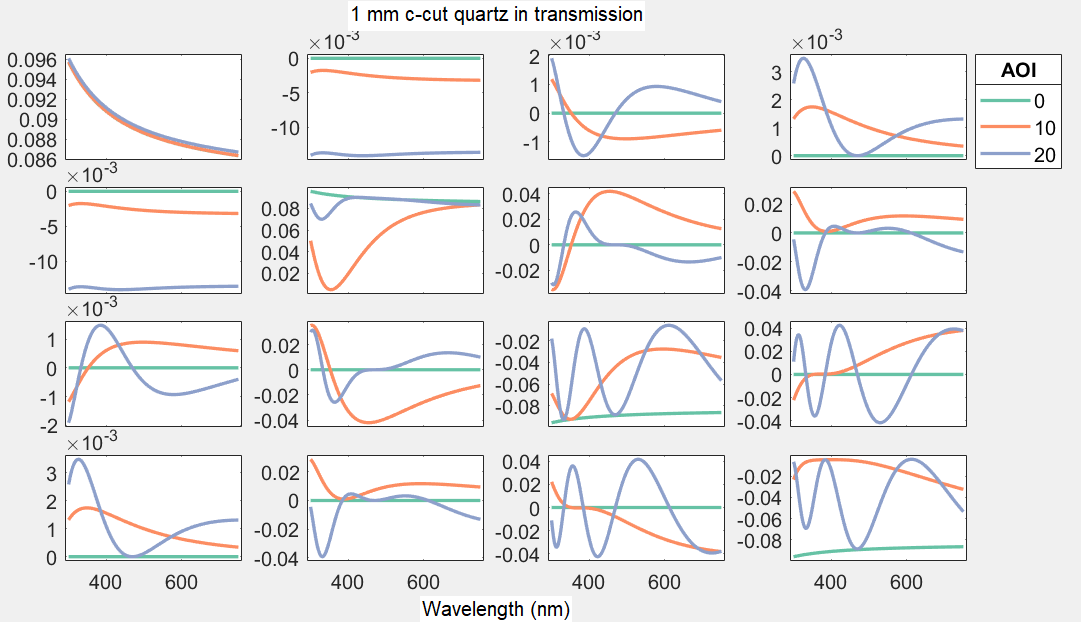
'legend', {'AOI', [0 10 20]}, ... % Create a legend

'lineNV', {'LineWidth', 2}, ... % Set line width

'axNV', {'ColorOrder', cbrewer('qual', 'Set2', 3)}); % Modify line colors using ColorBrewer

% Step 4: Save the figure to a file

pObj.print('MPlot\_example1.eps'); % Save the plot as an EPS file



This code defines a class called **MPlot** in MATLAB. It is used to create a 4x4 grid of plots, specifically tailored for visualizing spectral Mueller matrix data or similar multi-dimensional datasets. The **MPlot** class allows for various customization options such as figure size, font size, line specifications, axis properties, and legends. Let me explain the different sections of the code in detail.

**Class Properties**

1. **size**:
   * **Type**: Array.
   * **Default**: [1000 700].
   * **Purpose**: Sets the size of the figure in pixels. The first element is the width, and the second is the height.
2. **fontsize**:
   * **Type**: Scalar.
   * **Default**: 12.
   * **Purpose**: Defines the font size for all labels and text in the figure.
3. **limy**:
   * **Type**: Scalar.
   * **Default**: 1e-4.
   * **Purpose**: Sets the minimum y-axis range for the plots. If the plot range is smaller than this value, it will be expanded to meet this threshold.
4. **title**:
   * **Type**: String.
   * **Default**: Empty string ('').
   * **Purpose**: Sets the title of the plot.
5. **legend**:
   * **Type**: Cell array or numeric array.
   * **Default**: Empty ({}).
   * **Purpose**: Creates a legend for the plot. The first element is a string that is the title of the legend, and the second element can be a numeric or cell array of strings representing the legend labels.
6. **axNV**:
   * **Type**: Cell array.
   * **Default**: Empty ({}).
   * **Purpose**: Stores the axes name-value pairs. These can be any valid axes properties like axis limits, tick marks, etc.
7. **lineNV**:
   * **Type**: Cell array.
   * **Default**: Empty ({}).
   * **Purpose**: Stores the line name-value pairs. These are used to control the line style, color, and width in the plots.
8. **ev**:
   * **Type**: Boolean.
   * **Default**: false.
   * **Purpose**: If true, the x-axis will display the energy in electronvolts (eV) rather than wavelength in nanometers.
9. **vSpace**:
   * **Type**: Scalar.
   * **Default**: 0.
   * **Purpose**: Adds vertical space between the rows of the subplots in pixels.
10. **borderFactor**:
    * **Type**: Scalar.
    * **Default**: 0.
    * **Purpose**: Adds extra padding (white space) around the plot. It is a multiple of the largest line width in the figure.
11. **figHandle** (Protected):
    * **Type**: Handle.
    * **Purpose**: Stores the handle to the figure object. This allows for manipulating the figure later.
12. **axesHandles** (Protected):
    * **Type**: 4x4 matrix of axes handles.
    * **Purpose**: Stores the axes handles for the 4x4 grid of subplots. Each handle corresponds to one of the subplots in the grid.

**Methods**

**1. Constructor (MPlot)**

matlab

Copy code

function obj = MPlot(varargin)

obj.figHandle = figure;

makePlots(obj, varargin{:});

end

* **Purpose**: The constructor initializes a figure using MATLAB's figure function and calls the makePlots method to set up the 4x4 grid of plots based on the input arguments.
* **varargin**: It accepts a variable number of input arguments, which can be a mix of data, error bars, and other options for plotting.

**2. add Method**

matlab

Copy code

function add(obj, wavelengths, mmData, varargin)

[mmError, lineSpec] = parseVariableInputs(obj, varargin{:});

plot(obj, wavelengths, mmData, mmError, lineSpec);

end

* **Purpose**: This method is used to add additional data to an existing plot. It parses the input arguments (via parseVariableInputs), and then it calls the plot method to actually add the data to the plot.

**3. print Method**

matlab

Copy code

function print(obj, filepath)

print(obj.figHandle, filepath, '-depsc');

end

* **Purpose**: Saves the current figure to a file, typically in EPS format (-depsc stands for color Encapsulated PostScript). The filepath argument specifies the location and filename.

**4. parseVariableInputs Method**

matlab

Copy code

function [mmError, lineSpec] = parseVariableInputs(obj, varargin)

* **Purpose**: Parses the input arguments to handle optional error bars, line specifications, and other properties like font size, figure size, title, and legends.
* **Arguments**:
  + **mmError**: Error bars corresponding to the data being plotted.
  + **lineSpec**: Line style specification for the plot (solid, dashed, etc.).
  + **ev**: Converts x-axis from wavelength to electronvolts if true.
  + **fontsize**, **limy**, **lineNV**, **axNV**, etc. are extracted from the varargin input to control various properties of the figure.

**5. makePlots Method**

matlab

Copy code

function makePlots(obj, wavelengths ,mmData, varargin)

* **Purpose**: This method creates the figure layout and prepares the 4x4 grid of subplots for the Mueller matrix or other data. It calculates appropriate positions for each subplot and sets their properties (e.g., size, spacing, axis limits).
* **Steps**:
  1. **Check Screen Size**: Ensures the figure fits within the available screen dimensions.
  2. **Set Figure Properties**: Adjusts the size, position, and title of the figure.
  3. **Create Axes Handles**: Creates a 4x4 grid of subplot axes and stores their handles in axesHandles.
  4. **Plot Data**: Calls the plot method to plot the data in the subplots.

**6. plot Method**

matlab

Copy code

function plot(obj, wavelengths, mmData, mmError, lineSpec)

* **Purpose**: Plots the given data in the 4x4 subplot grid. It handles different scenarios, such as whether error bars are provided, and adjusts the y-axis limits based on the limy property.
* **Key Functionality**:
  + **Axes Properties**: Applies any axes-related settings (like tick marks, colors, etc.) through the axNV property.
  + **Line Plotting**: Uses the provided data and line style to draw the lines on each subplot.
  + **Error Bars**: If mmError is provided, error bars are plotted using MATLAB's errorbar function.
  + **Adjust Axes Limits**: If the y-axis range is smaller than the limy threshold, it adjusts the limits to ensure sufficient range.

**Summary:**

The **MPlot** class is designed for efficiently visualizing multi-dimensional data like Mueller matrix spectra. It sets up a 4x4 grid of subplots and provides various customization options like figure size, font size, line styles, error bars, axis properties, and legends. This class allows the user to easily plot and customize spectral data with high flexibility, making it ideal for scientific visualization.

**3.2 MPlot3D**

The MPlot3D class is designed to create high-quality publication-ready plots of Mueller matrix data in the form of image plots. This is particularly useful when you have spatially resolved Mueller matrix data, meaning the data varies across both horizontal (X) and vertical (Y) directions.

**Key Features and Properties**

1. **figHandle**: This is a handle to the figure object, which contains the plot. It allows users to manipulate the figure after it's created, such as adding new elements or modifying existing ones.
2. **axesHandles**: A 4x4 array of axes handles. Each handle corresponds to one of the 16 elements of the Mueller matrix. The matrix elements are arranged in a 4x4 grid, and each element is represented as a separate plot within this grid.
3. **colorbarHandles**: This is also a 4x4 array, where each handle refers to the colorbar associated with the respective axes in the axesHandles. Colorbars help in visualizing the scale of the data, particularly for spatially resolved data.
4. **uniquezero**: A boolean property that, when set to true, ensures that values of zero are mapped to the center of the colormap. This is useful when visualizing Mueller matrices that may have positive and negative values. By centering zero, you can clearly see where sign changes occur in the data. The default value is true.
5. **palette**: A string that specifies the name of the colormap to be used for the plots. You can use any colormap available in MATLAB or custom colormaps included in the class (such as HotCold Bright). The colormap defines how data values are translated into colors for visualization.
6. **gs (Global Scale)**: A numeric array that defines the global scale for the plots. If this is set to 0, each plot has its own independent colorbar. If it’s set to an array like [min max], all plots use a global colorbar with the specified range.
7. **width**: Specifies the width of the figure in inches, allowing you to control the size of the output plot.
8. **fontsize**: This controls the font size of all text in the figure, such as axis labels and titles. The default font size is 14.
9. **limz**: A numeric property that defines the minimum range of the colorbars. This helps avoid very small ranges in the colorbars, which can make the data difficult to interpret.
10. **norm**: A boolean that determines whether the Mueller matrix should be normalized by its M11 element. Normalization helps standardize the matrix so that it's easier to compare results across different samples or experiments. The default value is true.
11. **hSpacing**: Specifies the horizontal spacing between the plots in the 4x4 grid, measured in points. The default value is 3 points.
12. **vSpacing**: Specifies the vertical spacing between the plots, also in points. Like hSpacing, this property helps control the appearance of the plot grid.
13. **cbw (Color Bar Width)**: Defines the width of the colorbars in points. The default value is 10 points.

**Methods**

* **plot(pObj, mmData, PROPERTIES)**: This method allows you to plot new data on an existing figure, defined by pObj. You can also update the figure's properties using optional PROPERTIES.
* **update(pObj, PROPERTIES)**: This method updates the properties of the existing MPlot3D object and redraws the figure using the existing data.
* **replacePlotData(pObj, mmData)**: This method replaces the plot data without resizing or changing any properties. It is an efficient way to update the data in the plot without modifying other settings.
* **getPlotData(pObj)**: This method extracts the data from the plots in the figure.

**Colormap Control**

The MPlot3D class provides a range of colormaps for visualizing the data. When the uniquezero option is enabled, values of zero are mapped to the center of the colormap, making it easier to spot changes in the sign of the Mueller matrix elements. This is especially useful when using colormaps like HotCold Bright, which has white at the center, to make sign changes more visible.

In summary, the MPlot3D class is a powerful tool for visualizing spatially resolved Mueller matrix data, offering detailed control over the appearance of the plots, from color scaling and spacing to normalization and font size. It helps in producing consistent and visually appealing plots for publications.

The MPlot3D class is designed to create visually appealing, publication-ready 3D plots of Mueller matrix data, where each element of the Mueller matrix is plotted as a separate color-coded 2D image (or heatmap) in a 4x4 grid. This is particularly useful when handling multidimensional Mueller matrix data that varies across both the horizontal (X) and vertical (Y) dimensions.

The key features, properties, and methods are explained below:

**Properties**

1. **uniquezero (default: true)**
   * This boolean property controls whether zero values in the data are mapped to the center of the colormap.
   * When set to true, zero values are shown as a specific, neutral color (usually white or black, depending on the colormap), making sign changes in the data more obvious.
2. **palette (default: 'HotCold Bright')**
   * This property defines the colormap used for plotting the data. The default value is 'HotCold Bright', but any valid MATLAB colormap or custom colormap can be used.
   * Colormaps determine how data values are mapped to colors in the plot.
3. **gs (Global Scale)**
   * Defines the scale for all subplots. If this is set to 0 (the default), each subplot has its own independent color scale. If it’s set to an array like [min max], all subplots share the same global color scale, making comparisons easier.
4. **width**
   * This property controls the width of the figure (in inches). The height is automatically computed based on the width to ensure the plots are not stretched.
5. **fontsize (default: 14)**
   * This property sets the font size for all text in the figure, such as colorbar labels and titles. Adjusting this allows better control of plot readability.
6. **limz (default: 1e-3)**
   * This property limits how small the range of the colorbar (Z-scale) can be. It ensures that even if the range of data values is very small, a minimum range is used for better visualization.
7. **norm (default: true)**
   * If true, the Mueller matrix data is normalized by its first element (M11). This standardizes the data across different samples or experiments, making them easier to compare.
8. **hSpacing (default: 3)**
   * Specifies the horizontal spacing (in pixels) between the subplots. This helps adjust the layout and appearance of the plot grid.
9. **vSpacing (default: 3)**
   * Specifies the vertical spacing (in pixels) between the subplots, similar to hSpacing.
10. **cbw (default: 10)**
    * Defines the width of the colorbars (in pixels) associated with each subplot.
11. **figHandle**
    * Stores the handle to the figure object created by the class. This allows for later manipulation of the figure.
12. **axesHandles**
    * A 4x4 array of axes handles. Each handle corresponds to one element of the Mueller matrix. These handles allow for individual manipulation of each subplot.
13. **colorbarHandles**
    * A 4x4 array of colorbar handles. Each handle refers to the colorbar associated with the respective subplot. Colorbars provide a scale for interpreting the data values in the subplots.
14. **maskHandles**
    * Hidden property that stores handles related to masking operations, where specific regions of the plot can be masked or highlighted using an ellipse.

**Methods**

1. **MPlot3D Constructor**
   * The constructor initializes a new MPlot3D object, creates a figure, and plots the data based on the provided input. It accepts various optional parameters to customize the plot’s appearance, such as colormap, fontsize, and spacing.
2. **plot(obj, data, varargin)**
   * This method creates the actual plots. It takes a 4x4 Mueller matrix array (data) as input and generates a 4x4 grid of color-coded subplots. It supports various optional name-value pairs to customize the appearance (e.g., colormap, fontsize, spacing).
   * If gs is provided, all subplots use a global scale; otherwise, each subplot gets its own scale.
3. **getPlotData(obj)**
   * This method extracts and returns the data currently being displayed in the plots. It retrieves the color data (CData) from each subplot.
4. **replacePlotData(obj, mmdata)**
   * Replaces the plot data without changing the layout or settings of the plot. This is an efficient way to update the content of the plots without redrawing the entire figure.
5. **update(obj, varargin)**
   * Updates the properties of the existing MPlot3D object and redraws the figure with the same data but new settings. It’s useful for adjusting the appearance of an existing plot without changing the data.
6. **drawMask(obj, i, j)**
   * Draws an ellipse (mask) on the subplot located at position (i, j). This method is useful for highlighting or masking specific regions of a plot.
7. **setElipse(obj, position)**
   * Sets the position of the ellipse (mask) on the plot. You can use this to adjust the shape and location of the mask.
8. **applyMask(obj)**
   * Applies the mask to the plot, modifying the data in the selected region according to the mask.
9. **applyMaskWithTrim(obj)**
   * Applies the mask to the plot, but also trims the data to the region inside the mask. This is useful for focusing on a specific area of the plot and removing the rest.
10. **print(obj, filepath)**
    * Exports the plot to an EPS file. EPS (Encapsulated PostScript) is a vector graphics format that is commonly used in publications.
11. **flipX(obj)**
    * Flips the data along the X-axis, effectively mirroring the plot.

**Colormap Handling**

* **makeColormap(clim, b\_uniqueZero, palette)**
  + Generates a custom colormap based on the given limits (clim), whether zero is unique (b\_uniqueZero), and the specified colormap (palette).
  + It ensures that zero values are displayed prominently if uniquezero is set to true.
* **colPalette(palette)**
  + Defines various custom colormaps that can be used in the plot, such as 'HotCold Bright', 'Fireice', 'Spectral', and others. This function provides flexibility in selecting visually distinct colormaps.