Calculating and plotting Mueller matrices of stratified media

Shane Nichols
snichols27@gmail.com

November 13, 2017

Contents

1 Introduction

This is a collection of code to supplement my PhD thesis "Coherence in Polarimetry" [1]. The relevant models to calculate Mueller matrices of stratified media are described in Ref. [2]. The

reader should be familiar with the principles therein before using these programs. Some custom Matlab classes for producing nice plots are also provided. They include the ability to plot error bars and other tools that make them appropriate for plotting measured values as well as simulations. Finally, two measurement classes that help read in and process data from our custom polarimeter are documented here. These documentation files are also published in [1].

2 Calculating Mueller matrices of stratified media

2.1 Building a model

Each layer in a multilayer is described by the five parameters

`material`	String. Name of a material in materialLib.m or materialLibIso.m.
`d`	Numeric. Thickness of the layer, in nanometers.
`eul`	Numeric array. Rotations angles for a 3D passive ZXZ Euler rotation, in degrees.
`bool_thick`	Boolean. Sets the thick layer for the partial wave calculation.
`bool_isotropic`	Boolean. Set to true if the material is isotropic.

The parameters are arranged in the cell array `layer`, as,

```
layer = {material, d, eul, bool_thick, bool_isotropic};
```

Layers are ordered in a cell array according to their physical order in the multilayer stack, as in

```
layerArray = {layer1, layer2, layer3, ... , layerN};
```

The ambient medium on the incident side should always be the first layer, i.e., `layer1`. Similarly, the exit ambient medium should be the last layer. Ambient media are usually air but can be any isotropic material. For example, if the specimen is immersed in a cuvette of refractive index matching fluid, then the optical functions of the index fluid should be programmed into materialLibIso.m and the `material` parameter of the first and last layers set accordingly. For reflection calculations, it is also possible to treat the last layer as a semi-infinite substrate, which is an appropriate choice if light reflected from the back interface of the last layer does not reach the detector. Both anisotropic and isotropic semi-infinite substrates are supported. To use this model, simply set the thickness parameters of the last layer to infinity, i.e., `d = Inf`.

Example: Model for a partial wave calculation of 1 mm (001) +quartz with 0.3 deg miscut and an azimuthal angle of 24 deg coated on both sides with 20 nm of gold, in air:

```
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};
```

Example: Model for a Berreman calculation of a 30nm TiO_2 film grown on a (100) silicon wafer where the optic axis of TiO_2 is in the plane and along the x axis:

```
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};
```

2.2 Spectroscopic data

2.2.1 Berreman method

The function `mmBerreman` computes spectroscopic Mueller matrices using the fully coherent Berreman method. This function does not read the values of `bool_thick` but rather assumed that all layers are 'thin'. All inputs of this function are required position inputs, and are,

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

`layerArray`	Model of layers, as described in Section 2.1.
`wavelengths`	Numeric array. Wavelengths at which to compute, in nanometers.
`aoi`	Numeric. Free-space angle of incidence, in degrees.
`bReflect`	Boolean. True for reflection, false for transmission.
`bNormalize`	Boolean. If true, each Mueller matrix is normalized by M_{11}.

The output `M` is a $4\times4\times N$ array with N being the number of wavelengths. Most matrix and Kronecker products are fully vectorized and are very efficient. Most of the time is spent constructing and computing the Berreman layer matrices, which is looped over the wavelength. It is certainly possible to improve this part of the code by using analytical solutions to the matrix exponential, when tractable, and by using the point group to optimize the construction of the delta matrix. However, the code is sufficiently fast for our purposes with most calculations taking less than 1 sec.

2.2.2 Partial wave method

The function `mmPartialWave` computes spectroscopic Mueller matrices using the partial wave method. The value of `bool_thick` should be true for exactly one layer in the multilayer stack. Inputs and outputs for this function are identical to `mmBerreman` in the previous section. As for the

Berreman calculation, the partial wave calculation is fully vectorized using the multiprod toolbox [3].

2.3 K-space maps

The two functions described here are very similar to those in the previous section except that the Mueller matrix is calculated for a regular grid of in-plane incident wave vector components, rather than for a single incident direction. The functions output k-space maps of two kinds: conosopic or polar maps, which respectively differ in whether the specimen azimuth is stationary or rotates with the azimuth of the map. Back focal plane k-space imaging systems [4] directly measure conoscopic maps because the specimen is stationary. On the other hand, a traditional ellipsometer can also acquire k-space data by mechanically reorienting the specimen with a motorized goniometer and repeating Mueller matrix measurements in serial [5, 6]. We call these types of maps 'polar maps' to distinguish them from conoscopic maps. The two types of coordinate systems are illustrated on the right side of the first row of Table 3, where x and y refer to the instrument axes and the polar coordinates refer to the in-plane component of the wave vector, i.e., the radius is $\sin(\theta)$ where θ is the angle of incidence, and the azimuthal angle ϕ is the azimuth. In a polar map, the specimen is physically rotated with respect to the instrument to send the beam along different directions of the sample, so that in k-space, the instrument coordinate rotates with the azimuth of the map. Example calculations that compare the two types of maps are shown in Table 3. In media without significant diffraction or scattering, conoscopic maps obtained with an imaging system and polar maps obtained with an ellipsometer contain equivalent information and can be interconverted according to,

$$\text{Transmission:} \quad \mathbf{M}_{\text{cono}} = \mathbf{R}(-\theta)\mathbf{M}_{\text{polar}}\mathbf{R}(\theta) \tag{1}$$

$$\text{Reflection:} \quad \mathbf{M}_{\text{cono}} = \mathbf{R}(\theta)\mathbf{M}_{\text{polar}}\mathbf{R}(\theta) \tag{2}$$

where,

$$\mathbf{R}(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3}$$

for either transmission or reflection measurements.

The function `mmBerremanMap` computes maps using the coherent Berreman method. Inputs to the function are,

```
M = mmBerremanMap(layerArray, wavelengths, Npts, maxAOI, bReflect,
    bNorm, bConoscopic)
```

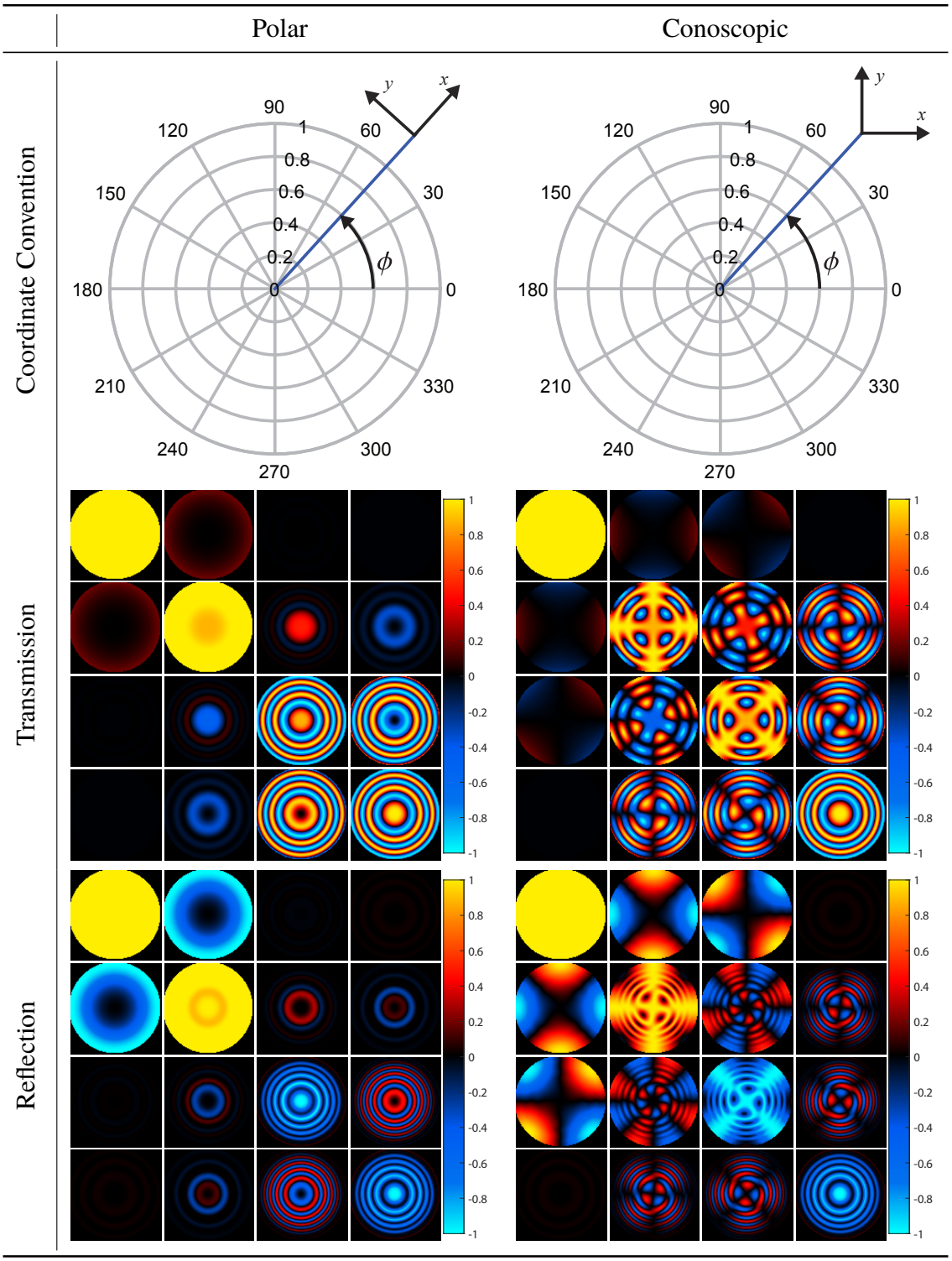


Table 3: Simulated k-space maps of a 300 µm thick (001) section of α-quartz with an NA of 0.85 in both transmission and reflection. Simulations were performed using the partial wave method in with both ambient media set to air.

`layerArray`	Model of layers, as described in Section 2.1.
`wavelengths`	Numeric array. Wavelengths at which to compute, in nanometers.
`Npts`	Integer. Number of grid points in X and Y.
`maxAOI`	Numeric. Maximum angle of incidence to compute, in deg.
`bReflect`	Boolean. True for reflection, false for transmission.
`bNorm`	Boolean. If true, each Mueller matrix is normalized by M_{11}.
`bConoscopic`	Boolean. If true, the polar map is converted to a conoscopic map.

The output `M` is a $4\times4\times N\times N\times M$ array where N is the value of `Npts` and M is the number of wavelengths. Most matrix and Kronecker products are vectorized and do not use for loops. The majority of time is spent computing the Berreman transfer matrices. The function `mmPartialWaveMap` similarly computes a k-space using the partial wave method.

3 Plotting classes

Two handle classes are provided to facilitate producing publication quality plots with little effort. Because these classes are handle classes, a command such as `pObj2 = pObj` does not copy the figure window associated with the object `pObj` or the properties contained within, rather `pObj2` is simply a new reference to the same object. Changing properties of `pObj2` also changes properties of `pObj` and vise-versa.

3.1 MPlot

The `MPlot` class is used for plotting spectral Mueller matrix data as a 4×4 array of line plots. A figure is created by calling the class constructor

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

The inputs `mmData` and `mmError` are n-dimensional arrays in which the first three dimensions have size $4\times4\times N$ where N is the number of wavelengths. Higher dimensions are flattened and plotted as additional lines. `lineSpec` is a Matlab LineSpec (Line Specification) string that is applied to all plots. Inputs `mmError` and `lineSpec` are optional positional inputs. `PROPERTIES` are optional Name-Value pairs that set properties of the MPlot class. The class has the properties

`figHandle`	Handle to the figure. Read only.
`axesHandles`	4×4 array of axes handles. Read only.

`size`	Numeric array. [Width, Height] of figure in inches. Default is [1000 700]
`fontsize`	Numeric. Font size of all text in figure in points. Default is 12.
`limy`	Numeric. Requires range of y axes to be less than `limy`. Default is 1E-4.
`title`	String. Title to place at the top of the figure.
`legend`	Cell array. Configures a legend to add to the plot `title', values'title', values` where `values` is either a cell array of strings, or a numeric array.
`axNV`	Cell array. Name-Value pairs of Axes Properties to format the plot axes.
`lineNV`	Cell array. Name-Value pairs of Line Properties to format the each plot.
`vSpace`	Numeric. Add extra vertical space between plots, in pixels.
`borderFactor`	Numeric. Adds extra whitespace between axis borders and plots. The value is a multiple of the largest line width on the plots.
`ev`	Boolean. If true, the x-axis is converted from nanometers to electronvolts.

In addition to the class constructor that creates the figure, other methods of the class are

`add(pObj, wavelengths, mmData, mmError, lineSpec, PROPERTIES)`	Adds additional lines to an existing MPlot object. The inputs are identical to the class constructor except for the first input `pObj`, which is a handle to a MPlot object.
`print(pObj, filepath)`	Creates an eps file of the figure window. The file is saved to the path filepath.

Example: Plot the transmission Mueller matrix of 1 mm c-cut α-quartz at three angles of incidence. The output is shown in Fig. 1.

```
% build the model
layerArray{1} = {'air', 0, [0 0 0], 0, 1};
layerArray{2} = {'+quartz', 1000000, [0 0 0], 1, 0};
layerArray{3} = {'air', 0, [0 0 0], 0, 1};
% calculate some transmission Mueller matrix spectra
M0 = mmPartialWave(layerArray, 300:750, 0, 1, 0);
M10 = mmPartialWave(layerArray, 300:750, 0, 10, 0);
M20 = mmPartialWave(layerArray, 300:750, 0, 20, 0);
% plot the data
pObj = MPlot(300:750, cat(4, M0,M10,M20), ...
```

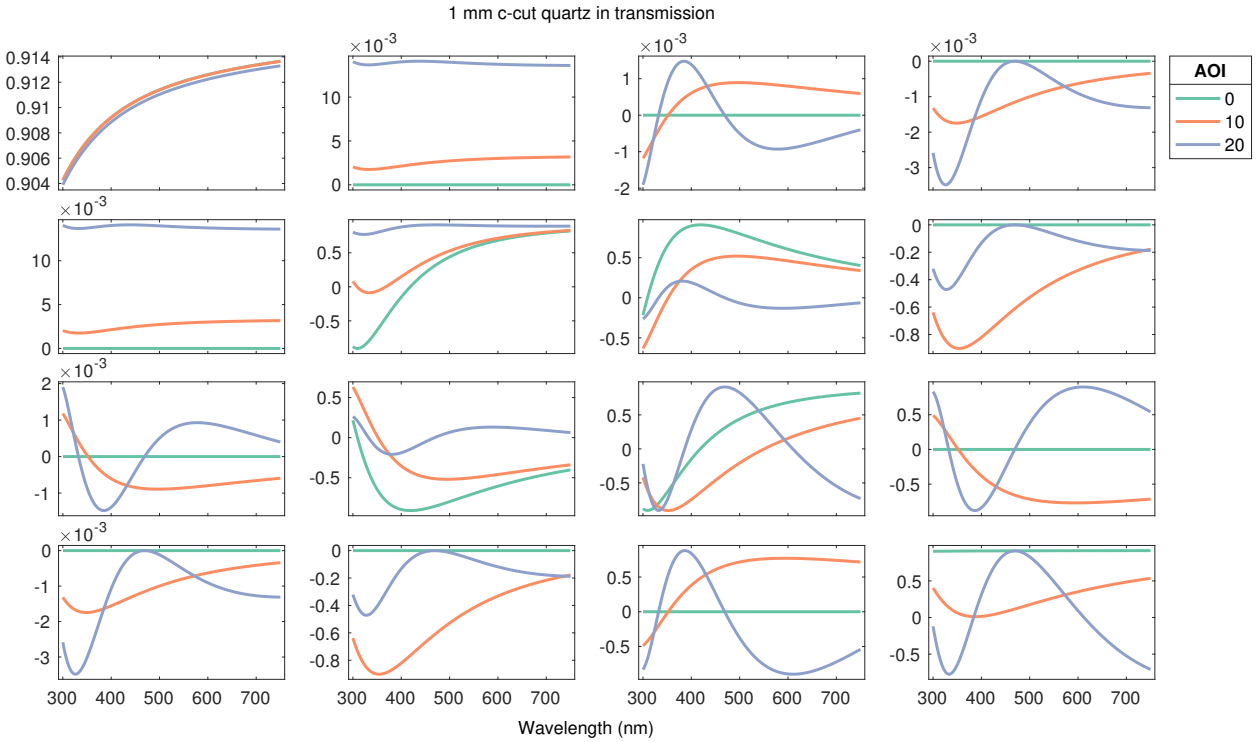


Figure 1: Example of a figure created with MPlot.

```
    'size', [800 500],... % set the size
    'borderFactor', 3, 'vSpace', 10,... % modify spacing
    'title', '1 mm c-cut quartz in transmission',... % add a title
    'legend', {'AOI', [0 10 20]},... % create a legend
    'lineNV', {'LineWidth', 2}, ... % modify line width
    'axNV', {'ColorOrder', cbrewer('qual', 'Set2', 3)}); % modify line
       colors
% print the figure to the current folder
pObj.print('MPlot_example1.eps');
```

Again, all Name-Value pairs are optional, but allow one to quickly customize the plot appearance. Note that the `cbrewer` package [7] was used to pick the plot colors rather than using the default Matlab color order. This package is redistributed here.

3.2 MPlot3D

This handle class facilitates creation of publication quality Mueller matrix image plots. Instances of the class are created by calling the class constructor `pObj = MPlot3D(mmData, PROPERTIES )` where `mmData` is an array of Mueller matrix data of size $4{\times}4{\times}X{\times}Y$, where X and Y are respectively the number of points along the horizontal and vertical directions of the plots. The class properties are,

`figHandle`	Handle to the figure. Read only.
`axesHandles`	4×4 array of graphics objects axes handles. Read only.
`colorbarHandles`	4×4 array of graphics objects colorbar handles. Read only.
`uniquezero`	Boolean. If true, values of zero are always mapped to the center of the full colormap. Default is true.
`palette`	String. Name of a colormap. Accepted values are the name of any function in the Matlab path accepts no inputs and returns a valid colormap, including the Matlab defaults, or the name of the custom colormaps provided within the class (see Table 9). Default is `'HotCold Bright'`.
`gs`	Numeric array. Short for Global Scale. Is set to 0, all plots have their own colorbar. If set to an array of [min max], all plots use a global colorbar. Default is 0.
`width`	Numeric. Width of the figure window in inches.
`fontsize`	Numeric. Font size of all text in figure in points. Default is 14.
`limz`	Numeric. Limits the range of colorbars to be greater than limz. Default is 1e-3.

`norm`	Boolean. If true, Mueller matrices are normalized by M_{11}. Default is true.
`hSpacing`	Numeric. Horizontal spacing between Mueller matrix plots in points. Default is 3.
`vSpacing`	Numeric. Vertical spacing between Mueller matrix plots in points. Default is 3.
`cbw`	Numeric. Short for Color Bar Width. Sets of the width of the colorbars in points. Default is 10 pts.

All writable properties can be set upon calling the constructor using Name-Value pairs. For example as, `pObj = CRRmeasurementObj.plot('palette', 'HotCold Bright', 'gs', [-1 1])`.

The following table list methods that plot new data on and/or update properties of an existing plot object:

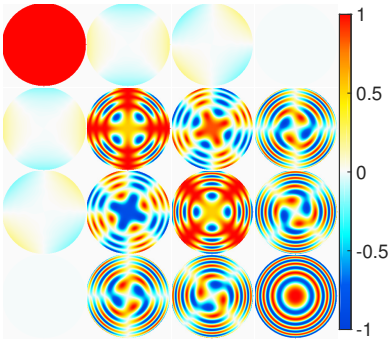
`plot(pObj, mmData, PROPERTIES)`	Plots new data on the same figure defined by `pObj` and optionally updates properties.
`update(pObj, PROPERTIES)`	Updates the properties of `pObj` and redraws the figure using the existing data.
`replacePlotData(pObj, mmData)`	Replaces plot data without resizing, changing properties, or updating colormaps. This method is very fast.
`mmData = getPlotData(pObj)`	Extracts data from the plots.

Colormaps and the uniquezero option . The `palette` property sets the name of a valid colormap. The name can be any of the built-in matlab colormaps, i.e., `'jet'`, but the built-in maps are terrible choices for bisignate data, and most are not colorblind friendly. Twelve custom maps are provided; the maps `'HotCold Bright'`, `'HotCold Dark'`, and `'Fireice'` are coded inside the class and do not have dependent files, while the remaining nine are taken from the cbrewer package [7] and require that package to be on the Matlab path. The package is redistributed in this repository. Table 9 gives example plots using each colormap. When the `uniquezero` flag is set, values of zero are always mapped to the center of the full colormap. In this case, the data will not use the full range of the colormap unless max(data) = -min(data). Using the `uniquezero` option along with a map having white at its center makes seeing sign changes in the Mueller matrix elements very easy. `uniquezero` has no effect when a global scale is used, as in Table 9.

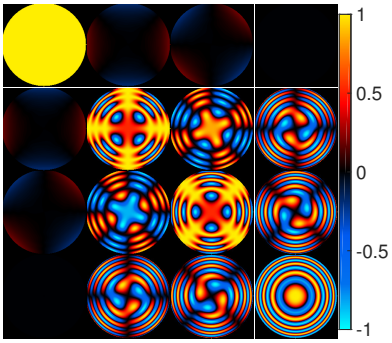
A set of methods are provided for cleaning up conoscopic Mueller matrix maps. Pixel values outside the image of the rear pupil take large and random values because there is no light, or stray light, at these points. These methods set the values to zero and optionally trim whitespace from the

Table 9: Colormaps for MPlot3D

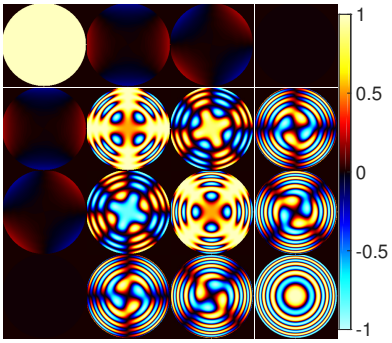
HotCold Bright



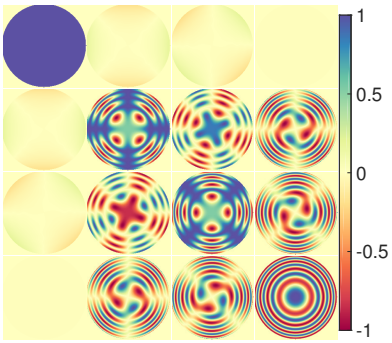
HotCold Dark



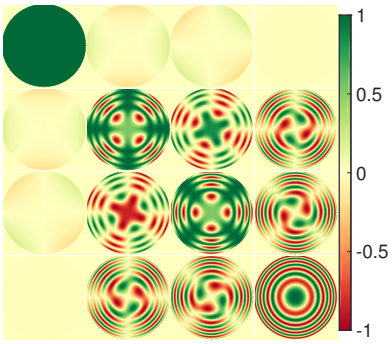
Fireice



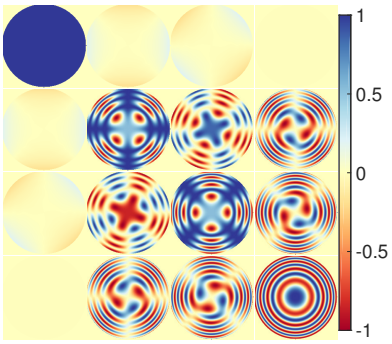
Spectral



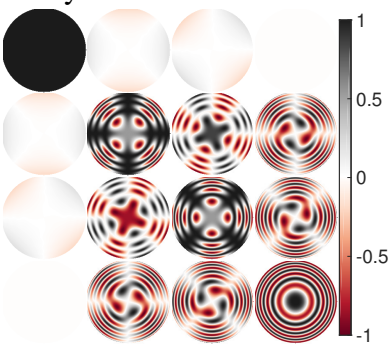
RdYlGn



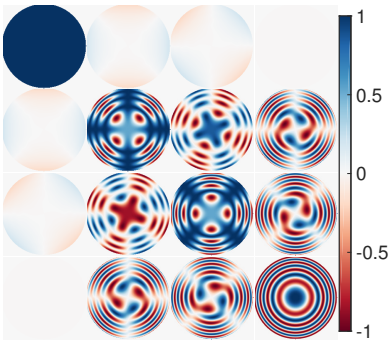
RdYlBu



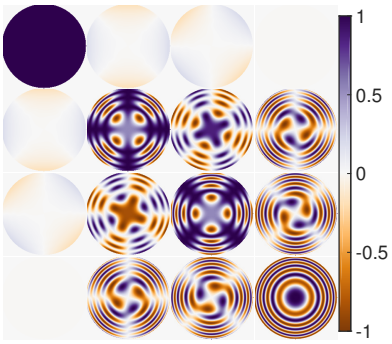
RdGy



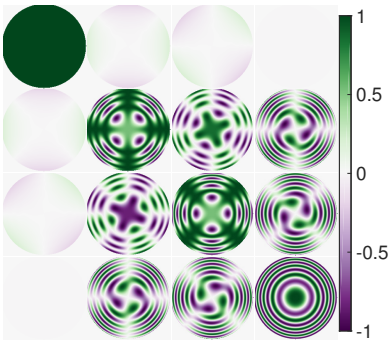
RdBu



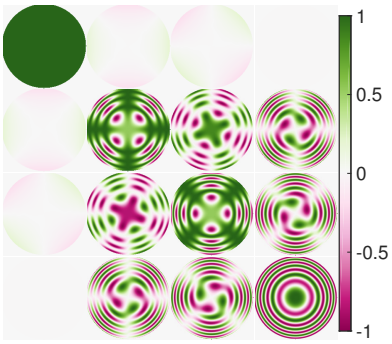
PuOr



PRGn



PiYG



BrBG

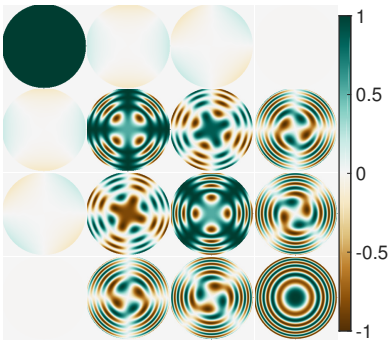
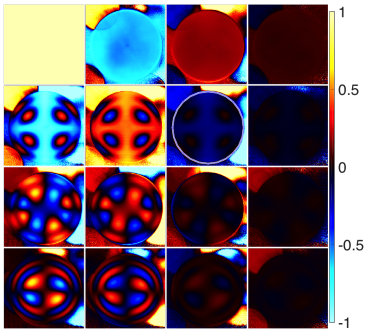
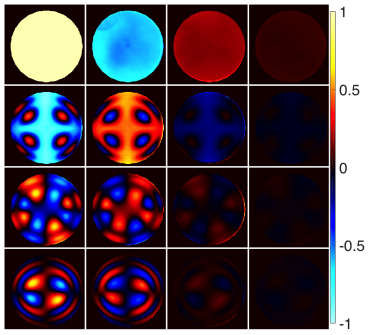
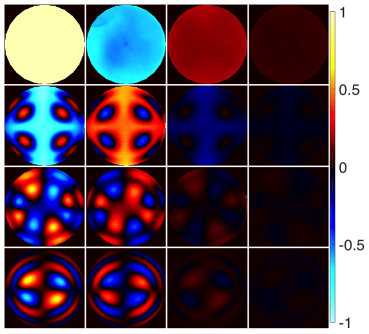


Table 11: Effects of methods used to clean up conoscopic data.

		
`pObj.drawMask(2, 3)`	`pObj.applyMask`	`pObj.applyMaskWithTrim`

resultant plots. The methods are

`pObj.drawMask(2, 3)`	Draws a circular mask on the Mueller matrix element M_{ij}. The ellipse can be manually resized to trace the boundary of the conoscopic data and the background.
`pObj.applyMask`	Set all pixels outside the ellipse to zero.
`pObj.applyMaskWithTrim`	Sets all pixels outside the ellipse to zero, and trims lines containing only zero (whitespace). Use this method instead of `pObj.applyMask`, not after it.

Table 11 shows an example of applying these functions to a dataset.

4 Measurement classes

The classes described here are particular to the custom Mueller matrix measurement devices in the research group of Bart Kahr at New York University. Design and operation principles of these devices are described elsewhere [1].

4.1 CRRmeasurement

This class is used to process and plot data from out imaging polarimeter. Although the LabView programs that run the instrument are self consistent and feature tools to process and plot Mueller matrix data, nearly all processing steps to go from raw frames to the Mueller matrix were also coded in Matlab. This was mainly for development and pedagogic purposes, but also to serve as a tool for carefully working up challenging datasets by hand, rather than using the automated routines

in LabView. This section will show how to perform the calibration procedure in practice using Matlab. Processing of raw frames in Matlab, as opposed to LabView, has all the same features, except that the Matlab functions do not correct for the rolling shutter of the camera. It is a relatively small effect, however, and very particular to the camera used. I elected to keep the Matlab code sufficiently general and assumed that the rather straightforward correction can be included later if needed.

CRRmeasurement is a value class that contains as properties all parameters necessary to compute the Mueller matrix from raw data. It reads in a parameter file produced by Labview, as well as raw frames and/or Mueller matrix elements saved as JPEG 2000 files. An instance of the class, throughout named `obj`, is created by executing the code,

```
obj = CRRmeasurement( path );
obj = obj.loadFrames;
```

with `path` is a string giving the path to the measurement folder. Frames are not loaded by default, and thus the second line is required to load frames into the class instance. The properties of the class are,

`N`	Scalar integer. Number of frames collected.
`periods`	Scalar integer. Number of waveform periods measured.
`dropped_frames`	Scalar integer. Number of dropped frames.
`frame_rate`	Scalar float. Frame rate of the camera in Hz.
`motor_speeds`	Float array with elements [motorSpeed1 motorSpeed2] in Hz.
`binning`	Scalar integer. Binning factor of the camera.
`roi`	Integer array with elements [Xoffset Yoffset Xwidth Ywidth] in pixels.
`exposure`	Scalar float. Exposure time of the camera, in seconds.
`dark_value`	Scalar float. Average dark value of the camera sensor.
`time`	Time of measurement formatted as, `9/3/2017 2:31 PM`
`wavelength`	Scalar float. Wavelength of the light source, in nm.
`mmtype`	String. Gives the type of matrix in properties mmdata.
`p1`	Scalar float. Phase of the first retarder, in rad.
`p2`	Scalar float. Phase of the second retarder, in rad.
`d1`	Scalar float. Retardance of the first retarder, in rad.
`d2`	Scalar float. Retardance of the second retarder, in rad.

`pol1`	Scalar float. Angle of the first polarizer, in rad.
`pol2`	Scalar float. Angle of the second polarizer, in rad.
`illumination`	Imageplane structure array.
`specimen`	Imageplane structure array.
`mmdata`	Array of float. Mueller matrix data of size $4 \times 4 \times X \times Y$
`frames`	Array of u16 integer or single precession float of size $X \times Y \times N$.
`path`	String giving the path to the measurement files.

Each property is accessed by `obj.PROPERTY` and can be treated like a normal variable. To get the Mueller matrix of the first pixel, for example, one uses `obj.mmdata(:,:,1,1)`. The properties `obj.illumination` and `obj.specimen` are structure arrays with the properties,

`image`	Image of the specimen or illumination. Specimen image data is only used for simulation purposes. Illumination image data is used in correcting for precession of the illumination by beam steering in the first rotating retarder (usually not needed).
`amplitude`	Scalar float Amplitude of precession.
`phase`	Scalar float. Phase of the precession.

These properties are accessed by `obj.specimen.PROPERTY`, for example, and store parameters needed to correct for precession artifacts.

Methods of the class used to process data are,

`obj = CRRmeasurement(path)`	Class constructor. Creates obj from data in folder.
`obj = loadFrames(obj)`	Loads frames into `obj.frames`.
`obj = getSpecimenPrecession(obj, frameSkip)`	Uses every *n*th frame, determined by `frameSkip`, to fit the specimen precession function. The method sets the values of `obj.specimen.amplitude` and `obj.specimen.phase`.

`obj = calculateMM(obj)`	Computes the Mueller matrix using analytical harmonic demodulation. The routine here is very fast, being built from the Multiprod toolbox developed by Paolo de Leva. Computing a Mueller matrix from 1000 frames of size 512×512 pixels takes approximately 1 sec on our machine. This method sets the value of `obj.mmdata` and therefore overwrites any Mueller matrix data imported from Labview.
`obj = fliX(obj)`	Flips the X axis of Mueller matrix maps, which is the forth element of the array `obj.mmdata`. This method simply performs `obj.mmdata = flip(obj.mmdata,4)`.

Each of these methods, given above in functional syntax, can also be called in dot syntax as `obj.METHOD(PARAMETERS)`. While the functional syntax is usually faster, the dot syntax allows one to easily string together any methods that return an instance of the class, which all of the above do. Therefore, the sequence of steps that load data, correct for precession, compute the Mueller matrix, and reindex to the standard coordinate system can be executed as the single command,

```
mmdata = CRRmeasurement(path).loadFrames.getSpecimenPrecession(
   frameSkip).calculateMM.flipX.mmdata;
```

The class also includes a few methods to facilitate simulation of raw data, which can be useful when troubleshooting new processing algorithms. These methods are,

`obj = setInstrument(obj, N, periods, p1, p2, d1, d2, pol1, pol2)`	Sets of the corresponding properties of `obj` having names equal to the inputs. These are all the parameters necessary to generate a waveform, except the specimen Mueller matrix.
`obj.specimen = obj.imageplane( PROPERTIES )`	(Static method) Sets one or more properties of `obj.specimen`. Accepted inputs are `obj.imageplane(image)`, `obj.imageplane(amplitude,phase)`, or `obj.imageplane(image,amplitude,phase)`. The same method, following the same input rules, can be used to set `obj.illumination`.

`obj = simulateRawData(obj, mm)`	Populates `obj.frames` by computed each frame according to the instrument theory. The input `mm` is a 4×4 Mueller matrix that is assumed to be homogeneous across the image. Raw data is simulated using all available properties loaded into the class in that, illumination and specimen precessions are simulated only if their respective properties are set.

Simulating data requires at least setting `obj.specimen.image`, and those properties set by `obj.setInstrument( )`. Precession of the image plane is simulated if `obj.amplitude` and `obj.phase` are set. Additionally, if `obj.illumination` is set, then precession of the illumination image is simulated.

4.2 muellerData

The Matlab muellerData class provides some tools for processing and plotting data collected with our 4PEM polarimeter. It is intended to lower the bar for students learning to use the device and work with the data. Most of the included methods are applicable to transmission data and pertain to the kinds of quantities introduced in [1, Chapt. 3].

In object oriented programming, a class is a collection of code having properties, which structure data, and methods, which structure functions that operate on data. Instances of a class are called objects. Consider the following instance of a `muellerData` object,

```
muellerData with properties:

     Label: 'smn-2-150-26'
     Value: [5-D double]
   ErValue: [5-D double]
      Size: [4 4 226 3 4]
      Dims: {[226x1 double]  [40 50 60]  [9.3 54.3 99.3 144.3]}
  DimNames: {'Wavelength'  'AOI'  'Azimuth'}
        HV: [226x3x4 double]
        DC: [226x3x4 double]
reflection: [true]
```

It represents a dataset collected with 226 wavelengths, 3 angles of incidence (AOI), and 4 azimuthal angles, at values stored in property `Dims`. Mueller matrix values and associated errors are stored in the properties `Value` and `ErValue`, respectively, and have sizes stored in `Size`. All properties except `Value` are optional and two, namely `HV` (high-voltage) and `DC` (average value of the waveform), are particular to our 4-PEM ellipsometer. Finally, `reflection` indicates whether the values correspond

to reflection or transmission data. Use function `read4PEMclass` to create a class from 4-PEM measurements. For example, the above example object can be created by,

```
obj = read4PEMclass('smn-2-150-26',{40:10:60,2},{9.3:45:144.3,1});
```

where `smn-2-150-26` is the base filename of the dataset. One could also load the dataset into an array and construct a class instance by manually setting each property by,

```
[MM,MMer,WL,HV,DC] = read4PEM('smn-2-150-26'
    ,1,{40:10:60,2},{9.3:45:144.3,1});
obj = muellerData(MM);
obj.Label = 'smn-2-150-26';
obj.ErValue = MMer;
obj.Dims = {WL  [40 50 60]  [9.3 54.3 99.3 144.3]};
obj.DimNames = {'Wavelength'  'AOI'  'Azimuth'}
obj.HV = HV;
obj.DC = DC;
obj.reflection  = true;
```

Note that `Size` is set automatically by the class constructor.

4.2.1 Custom indexing and object-arrays

MATLAB supports three indexing types: dot `'.'`, parenthesis `'( )'`, and curly braces `'{ }'`. Expressions may contain multiple levels of indexing. MATLAB supports few indexing possibilities by default, but custom indexing is implemented by overloading the built-in function `subsref`. The muellerData supports the following indexing routines on single objects (herein named `obj`):

`obj = obj( )`	parsed `muellerData` object (more info below)
`obj = obj{ }`	parsed `muellerData` object using `obj.Dims` (more info below)
`val = obj.property`	value of `property`
`val = obj.method`	value returned by `method`
`val = obj.property{ }`	value of `property` indexed by `{ }`
`val = obj( ).property`	value of `property` for parsed object
`val = obj( ).method`	value returned by `method` for parsed object
`val = obj{ }.property`	value of `property` for parsed object
`val = obj{ }.method`	value returned by `method` for parsed object

The first two options parse the properties of muellerData objects, (i.e., select a subset), and return

a new instance of the class. To parse the above example instance into data corresponding only to AOI's of 40 and 50, issue `obj(:,:,:,[1,2],:)`, which returns,

```
ans = muellerData with properties:
      Value: [5-D double]
    ErValue: [5-D double]
       Size: [4 4 226 2 4]
       Dims: {[226x1 double]  [40 50]  [9.3 54.3 99.3 144.3]}
  DimNames: {'Wavelength'  'AOI'  'Azimuth'}
         HV: [226x2x4 double]
         DC: [226x2x4 double]
reflection: [true]
```

All properties are automatically parsed. This indexing method is positional, thus the dimension of the requested indices must equal that of `Size`. Object parsing with `'( )'` only requires that `Value` and `Size` be defined, which is guaranteed by the class constructor. When `Dims` is defined, however, a more convenient indexing method is possible. Values used to index objects with `'{ }'` refer to the physical quantities in `Dims`. Hence, `obj(:,:,:,[1,2],:) == obj{:,:,:,[40,50],:}` in our example. The routine selects values from `Dims` nearest to the requested ones, after which duplicate points are automatically removed. Arrays in `Dims` are sorted and thus their entires need not be in any particular order. With this functionality, the user need not consider precisely how the data is organized to select a subset corresponding to prescribed physical values, thereby simplifying indexing of heterogeneous collections of objects where positional indices may vary. Collections of objects are stored in special MATLAB arrays called object-arrays.

Object-arrays Grouping objects into arrays facilitates operating on multiple objects. Creation of object-arrays follows standard notation:

```
objAr = [obj1, obj2, obj3, obj4];
objAr(5) = obj5;
```

A muellerData object-array supports the following indexing options:

`objAr = objAr( )`	positional indexing of `objAr` (usual array indexing)
`objAr = objAr{ }`	returns `objAr` of same size with each `obj` parsed as `obj{ }`
`valAr = objAr.property`	values of `property`
`valAr = objAr.method`	values returned by `method`
`valAr = objAr( ).property`	values of `property` for indexed `objAr`
`valAr = objAr( ).method`	values returned by `method` for indexed `objAr`
`valAr = objAr{ }.property`	values of `property` for parsed `objAr`

`valAr = objAr{ }.method`	values returned by `method` for parsed `objAr`

For properties, `valAr` is always a cell-array containing the value of `property`. For methods, `valAr` is an object array if `method` returns a new instance of the class, otherwise, it is a cell-array. If `objAr` is indexed to length one, as in `objAr(1).method`, the output array is stripped and just the property or result of method is returned.

4.2.2 Methods

Objects are always the first and usually only input variable to methods. Methods may be called in either dot or functional notation, which respectively read `obj.method(arg1,arg2,...,argN)`, and `method(obj,arg1,arg2,...,argN)`. If methods take only an object as input, `obj` need not be handed explicitly when using dot notation; the syntax `obj.method` is sufficient. The following methods return another instance of the class as output:

`lb(obj)`	x-y projection of linear birefringence
`ld(obj)`	x-y projection of linear dichroism
`lbp(obj)`	xy-yx projection of linear birefringence
`ldp(obj)`	xy-yx projection of linear dichroism
`cb(obj)`	circular birefringence
`cd(obj)`	circular dichroism
`a(obj)`	mean absorption
`a_aniso(obj)`	anisotropic part of mean absorption
`a_iso(obj)`	isotropic part of mean absorption
`lbmag(obj)`	magnitude of linear birefringence
`lbang(obj)`	angle of linear birefringence
`ldmag(obj)`	magnitude of linear dichroism
`ldang(obj)`	angle of linear dichroism
`optProp(obj)`	all seven optical properties (LB, LD, LB', LD', CD, CB, A)
`lm(obj)`	analytic matrix logarithm (has Mueller-Jones symmetry)
`logm(obj)`	numerical matrix logarithm (for depolarizing media)

`expm(obj)`	numerical exponentiation of log(**M**)
`di(obj)`	depolarization index
`jones(obj)`	Mueller-Jones to Jones
`nearestjones(obj)`	Mueller to nearest Jones (for depolarizing media)
`mfilter(obj)`	nearest physically realizable matrix
`covar(obj)`	Mueller to Cloude covariance matrix
`mrotate(obj,angle)`	rotate coordinate about the wave-vector

Those in blue overload built-in functions for the class. Note that all other properties are copied to the new object, hence, if circular dichroism were computed for the whole dataset via `CDobj = obj.cd`, the resultant object `CDobj` that contains the values of circular dichroism can still be indexed by `CDobj( )` or `CDobj{ }`.

Overloaded binary operators Basic mathematical operations may be performed directly on objects. Currently, there is no support for object-arrays. The following operations are supported for objects:

`obj = obj1 + obj2;`	New object with value `obj1.Value + obj2.Value`
`obj = obj1 - obj2;`	New object with value `obj1.Value - obj2.Value`
`obj = obj1 .* obj2;`	New object with value `obj1.Value .* obj2.Value`
`obj = obj1 ./ obj2;`	New object with value `obj1.Value ./ obj2.Value`
`obj = obj1 * obj2;`	Matrix product of Mueller matrices in `obj1` and `obj2`
`obj = obj1 / obj2;`	Matrix division of Mueller matrices in `obj1` and `obj2`

Organizational methods These are methods that manipulate objects rather than perform calculations. A method is provided to merge two objects that differ only in one element of `obj.Dims`. For example, say one collects a dataset at various incident angles and azimuthal angles. Later, it is determined that additional incident angles are needed. If another dataset is aquired with different incident angles but the same azimuthal angels and wavelengths, then they can be directly merged as,

```
objMerged = merge(obj1,obj2);
```

Another common situation is having multiple datasets with different dimensions. Given a

dataset with multiple incident angles and azimuths and several other datasets with only multiple azimuths, it may be necessary to pull out a single incident angle from the first higher dimension dataset and add it to an object array of other lower dimensional datasets. Simply indexing the first dataset for one incident angle is not sufficient; there will be a singleton dimension remaining and the dimension will be unchanged. To solve this, the method `squeeze` is supplied. For example, consider our example object given earlier. We wish to index a single incident angle:

```
objNew = obj{:,:,:,50,:}; % returns new object with select properties:
        Size: [4 4 226 1 4]
        Dims: {[226x1 double]  [50]  [9.3 54.3 99.3 144.3]}
objNew = squeeze(obj{:,:,:,50,:}) % returns new object with select
   properties:
        Size: [4 4 226 4]
        Dims: {[226x1 double]  [9.3 54.3 99.3 144.3]}
```

All other properties also effected by removing the singleton dimension are appropriately modified.

Multiple dot-Levels All aforementioned indexing options on objects or object-arrays support multiple levels of dot indexing. This means that the output from any method that returns a muellerData instance may be directly handed to another method using the notation `valAr = objAr.method1.method2` for any number of methods. For example, if you want to filter the Mueller matrix and then compute the depolarization index for all objects in an object array, simply use,

```
valAr = objAr.mfilter.di;
```

Methods that require additional inputs must come last. The syntax,

```
valAr = objAr.method1.method2(arg1,arg2,...,argN)
```

is valid whereas,

```
valAr = objAr.method1(arg1,arg2,...,argN).method2
```

is not. Finally, a call to value may follow any number of methods, as, `obj.methods.Value`.

Static methods Static-methods are methods that do not take an object of the class as an input. With static-methods, one can operate directly on a numeric arrays of data without having to first construct an instance of the class. To declare static-methods of a class, one can either use the name of the class as `muellerData.staticMethod( )`, or use any arbitrary instance of the class as, `obj.staticMethod( )`. Both serve to inform MATLAB from which class to retrieve the function. All of the aforementioned methods may also be called as static-methods, which are named by placing an `s_` before the corresponding method name. For example, the following operations are equivalent, where `MM` is an array of Mueller matrix data (i.e., `MM = obj.Value`):

```
obj.cb.Value    % calculate CB for values in obj.Value (uses method)
muellerData.s_cb(MM) % same operation using static-method
```

4.2.3 Plotting

`muellerData` was written before any of the other code in this documentation. As such, it does not use the MPlot class for plotting but has its own functions built in. They are, however, very similar. The `plot` method may be called with either function or dot notation, which respectively read

```
h = obj.methods.plot(formatSpec,Name1,Value1,...)
h = plot(obj.methods,formatSpec,Name1,Value1,...)
```

This construction is very similar to the built-in plot function, except that X and Y data are obtained from `obj` rather than given as inputs. In the case of a 4×4 plot, the output `h` is a 1×16 axes handle array, otherwise it is a single axes handle. All inputs to plot are optional. The input `formatspec`, if given, must come first and is identical to the formatSpec input of the MATLAB plot function. Following are any number of `Name,Value` pair arguments, which may be arranged in any order. Possible `Name,Value` pair arguments and their default values are,

`'handle',h`	Handle to a single axis, or a 1×16 array of axes handles on which to plot. If not given, new axes are drawn on a new figure. By default, the axes property `NextPlot` is set to `'add'`, thus plotting on existing axes adds additional lines.
`'axNV',axesProperties`	AxesProperties is a cell array of name-value pairs applicable to the Axes Properties of the MATLAB Axes class. Changes to MATLAB defaults include, `{'NextPlot','add','box','on'}`, which remain unless explicit changed in user set `axesProperties`
`'lineNV',lineProperties`	LineProperties is a cell array of name-value pairs applicable to the Primitive Line Properties of the MATLAB Line class. If not given, MATLAB defaults are used.
`'size',[xPixels yPixels]`	Size of the figure windows, not including the legend. Default values: [1000 700] for 4×4 plots, and [700 500] for single plots.
`'fontsize',12`	Font size of all text in figure given in points.
`'title','string'`	If given, a title is created and placed above the plots. The string may contain TeX mark-up.
`'limy',0`	Limits the minimum allowed range for the y-axes of plots.
`'ev',false`	If true, the x data is converted to eV as $x = 1239.8/x$

`'legend',value`	A legend is created by default if `obj.Dims` is defined. Although setting `value = 'none'` disables automatic legend creation. Additionally, the default legend can be overridden by setting `value = {'legendTitle',labels}`, where `'legendTitle'` is the title of the legend, and `labels` is either a cell-array of strings to directly use as labels, or a numeric array.
`'legendLocation','string'`	Only defined for single axis plots. This is identical to the 'Legend Location' property in MATLAB Legend Properties. The default is `'best'`. Only locations on the right side are supported for options that place the legend outside of axes. For 4×4 plots, the legend is fixed to `'northeastoutside'` with respect to $M_{1,4}$.
`'xLabel','string'`	If given, creates a label for the plot x-axis. Only available for single parameter plots.
`'yLabel','string'`	If given, creates a label for the plot y-axis. Only available for single parameter plots.

Below are some examples of using the `plot` method.

```
hAr = obj.plot % plot Mueller matrix elements as 4x4 array of plots.
hAr = obj{:,:,:,[40,50],:}.plot % plot data with AOIs of 40 and 50
h = obj.di.plot % plot DI of all data.
h = obj{:,:,:,[40,50],:}.di.plot % plot DI of data with AOIs of 40 and
    50
h = obj{:,:,:,[40,50],:}.mfilter.di.plot % same but first filter the MM
```

All of the above examples may also be performed on object-arrays, in which case a plot is made for each object in the array.

Printing plots A method called `print` is included that is identical to plot, except that it creates a .EPS file of the figure. The full path to the .EPS file (WITHOUT extension) must be the first input to `print`. The remaining inputs are those already given for `plot`, e.g.,

```
h = obj.methods.print('filepath',formatSpec,Name1,Value1,...)
```

Plot merging A method is provided to merge single axis plots into a single figure. For example, say you have six plots with handles stored in a handle array that you wish to arrange as a 3×2 array. This can be achieved by,

```
hAr = [h1,h2,h3,h4,h5,h6];                % put handles into a handle-
    array
```

```
fig = muellerData.mergeAxes(hAr,[3,2]); % merge into one figure
print(fig,'filename','-depsc')          % save as EPS
```

Here, `fig` is a handle to the new figure. To use this, the plots figures corresponding to the handles in `hAr` must still be open.

References

[1] S. M. Nichols, "Coherence in polarimetry", PhD thesis (New York University, 2018).

[2] S. Nichols, O. Arteaga, A. Martin, and B. Kahr, "Measurement of transmission and reflection from a thick anisotropic crystal modeled by a sum of incoherent partial waves", J. Opt. Soc. Am. A **32**, 2049–2057 (2015).

[3] P. de Leva, *Multiple matrix multiplications, with array expansion enabled*, https://www.mathworks.com/matlabcentral/fileexchange/8773-multiple-matrix-multiplications--with-array-expansion-enabled.

[4] O. Arteaga, S. M. Nichols, and J. Antó, "Back-focal plane Mueller matrix microscopy: Mueller conoscopy and Mueller diffractometry", Appl. Surf. Sci. **421**, 702–706 (2017).

[5] S. Nichols, O. Arteaga, B. Maoz, G. Markovich, and B. Kahr, "Polarimetric analysis of the extraordinary optical transmission through subwavelength hole arrays", Proc. SPIE, 91631W–91631W (2014).

[6] O. Arteaga, B. M. Maoz, S. Nichols, G. Markovich, and B. Kahr, "Complete polarimetry on the asymmetric transmission through subwavelength hole arrays", Optics express **22**, 13719–13732 (2014).

[7] C. Robert, *Cbrewer : colorbrewer schemes for matlab*, https://www.mathworks.com/matlabcentral/fileexchange/34087-cbrewer---colorbrewer-schemes-for-matlab.