Fourier Microscopy and Raman Spectroscopy

Jack Wilson

Advisor: Rashid Zia

Contents

Acknowledgements

I first owe a massive thank you to my advisor, Rashid, for many hours of support, both on this project and just about everything else. He has taught me many things, and he's even showed me how to have a little confidence in myself every once in a while.

I should also thank everyone in the lab, who made working on this project truly enjoyable. I am grateful for Jon's patience, since many of the worst questions in my quiver were shot straight at him. Matthew also often went completely out of his way to help me out. I send several thanks out to Wenhao, for half of his laser, occasionally his camera, and always his humor. Finally, I'll thank my good pal and undergrad lab mate Joe, for absorbing the many complaints that followed my many failures.

1 Abstract

Fourier microscopy is a technique commonly used to study the angular dependence of light emission. While this technique has been used to study fluorescence, it has not yet been applied to Raman scattering. The goal of this so far unsuccessful work was to fill this gap and perform angle-resolved Raman scattering measurements in a Fourier microscope. I first present an intuitive method to model Fourier microscopes and evaluate their performance using Matlab and Zemax. I then move to polarized Raman measurements on bulk and few-layered MoS_2, and finally discuss the experimental difficulties with angle resolved Raman measurements.

2 Fourier Microscopy

Traditional spectroscopy measures the intensity of light as a function of frequency. Analyzing the spectrum of a light matter-interaction–whether it be absorption, emission, or scattering–allows us to determine quite a bit about material properties. Light, however, carries not only energy information (encoded in ω), but also momentum information (encoded in the wave vector, $\mathbf{k}$). Though normal spectroscopy neglects $\mathbf{k}$, this valuable piece of information can be recovered with Fourier optics.

We normally think of lenses as magnification tools–they let us see small things. However, we can also think of lenses as devices that map angles to positions, and positions to angles. When used in this way, lenses in conventional microscopes can image the angular distribution of emitted/scattered light, which is directly related to the momentum distribution of the light. More generally, lenses allow us to take a Fourier transform, mapping position to momenta. A standard microscope can be converted to perform momentum-resolved measurements with the addition of a single lens focused on the back focal plane

(BFP) of the objective. This lens is called a Bertrand lens. (For a detailed diagram of our experiment setup, see Figure 7 in Section 5.)

The extra dimension of information that the momentum distribution provides is often quite useful. In photoluminescence measurements, for example, the momentum pattern of the emitted light can reveal the orientations of the optical transitions that generated that light (see Figure 1).

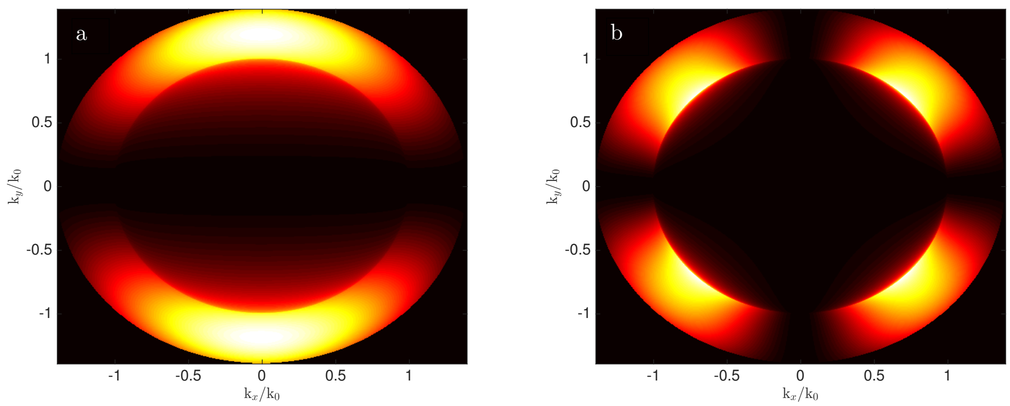

Figure 1: Calculated radiation patterns for two different types of transitions: a) an out-of-plane magnetic dipole transition and b) an in-plane electric dipole transition. The momentum-resolved radiation patterns clearly differentiate different light emission mechanisms.

3 Raman Scattering in Crystals

3.1 Conservation Rules

When photons scatter from matter, they generally do so elastically–the incident and scattered photons have the same energy. In Raman processes, however, light scatters inelastically, and the scattered photons have slightly higher or lower energy than the incident ones. This energy shift is due to the generation or absorption of phonons in the crystal lattice.

Two conservation rules determine the details of this process. If an incoming photon with energy $\hbar\omega$ and momentum $\hbar\mathbf{k}$ engages in a first-order Raman scattering process with a crystal, the scattered photon will have energy $\hbar(\omega \pm \omega_p)$ and momentum $\hbar(\mathbf{k} \pm \mathbf{k_p})$, where ω_p and $\mathbf{k_p}$ characterize the energy and momentum of the phonon. If the scattered photon gains energy, the scattering is called anti-Stokes Raman, and if the photon loses energy, the scattering is called Stokes Raman. A schematic of a Stokes Raman process is shown in Figure 2. It is worth noting that $\mathbf{k}$ for visible light ($\sim \pi/\lambda$) is significantly smaller than a typical phonon wavevector (up to $\sim \pi/a$, where a is the lattice constant), and as a result, Raman scattering only probes phonons near the center of the Brillouin zone.

Measuring the spectrum of the frequency-shifted scattered light (the Raman spectrum) reveals a great deal of information about the host lattice. Since the vibrational behavior of matter is directly related to the symmetry and composition of its constituents, the Raman spectrum contains information about the structure and composition of that matter. Raman measurements are purely optical, so they provide an easy and generally non-destructive way to probe these material properties. As a result, Raman spectroscopy has become an extremely popular and useful tool in chemistry and materials science.

The initial goal of this work was not only to measure frequency shifts (as in traditional spectroscopy), but to measure both the frequency and momentum shifts of Raman scattered light using Fourier microscopy. I discuss the challenges with this goal in section 6.

3.2 Selection Rules

Raman scattering transitions generally obey well-defined selection rules determined by the symmetries of zone-center phonons in the crystal and the polarizations of the incident and scattered light. A group theory analysis of these phonons can identify Raman allowed

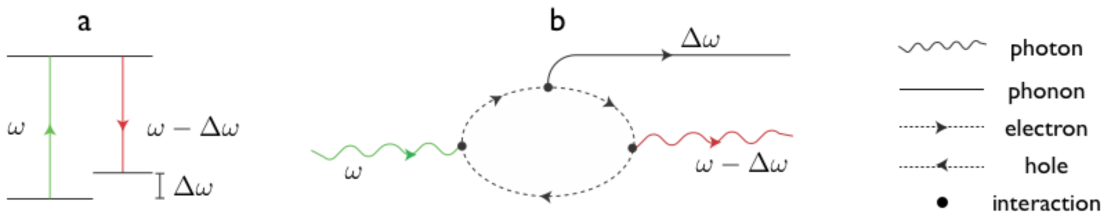

Figure 2: Pictorial illustrations of a Stokes Raman scattering process. a) Energy level diagram. The scattered photon has a lower energy than the incident photon. b) Diagram for one possible Stokes process. The incident photon is absorbed and creates a phonon. A photon at a lower frequency is then emitted.

transitions and the polarization configurations in which they can be measured. Ref. [1] outlines the calculation and simplifies the procedure with a comprehensive set of look-up tables. More perspectives on this process and Raman scattering in general can be found in refs. [2] and [3].

4 Model of Optical System and Ray-Tracing in Zemax

4.1 Zemax Software and Motivation for Optical Modeling

Zemax is a commercial software used to model optical systems and characterize their performance. Even in a simple system of a few lenses, a thorough characterization of an optical system's behavior can be difficult. Zemax handles the problem easily, enabling both comprehensive optimization and a better understanding of the optical system as a whole.

There are three main reasons why optical modeling was useful in this case. The first is that, unlike traditional spectroscopy (which only uses intensity information at each frequency), this technique spreads additional information in space at each frequency. This makes the measurement susceptible to geometric aberrations in the optical system that alter the image in space. To get truly reliable information out of these spatial images, it becomes important to quantify the effects of these geometric aberrations. The second reason for modeling was to evaluate the feasibility of more complex optical systems, such as a microscope with a polarization-splitting Wollaston prism. Finally, a computational model of the optics enabled a better intuitive understanding of the system and made it easier to diagnose problems and limitations during actual experiments.

In the following sections, I'll demonstrate a few cases in which modeling could be helpful, and describe the process I use to do the simulations.

4.2 Simple Modeling Problems

An ideal Fourier microscope would be able to simultaneously record light of perpendicular polarizations for two reasons. First, simultaneous measurements eliminate any potential changes in the sample or setup between measurements of different polarizations. Second, such a setup is more elegant and efficient, halving the amount of measurements necessary for an experiment. One way to accomplish this simultaneous polarization control is to put a Wollaston prism in the beam path before the spectrometer.

In previous iterations of the Fourier imaging setup, the Bertrand lens was chosen to spread the measured radiation pattern across the entire camera chip. With the addition of the Wollaston prism, however, two radiation patterns (one for each linear polarization) hit the camera at the same time. This leaves half as much space on the camera, which means the actual images need to be (1) at least half as small and (2) appropriately separated on the chip. A simulated ideal alignment of two sample images is shown in Figure 3.

Other practical matters also create constraints. For example, there are only a few places in the microscope where the Bertrand lens and prism can actually fit. Also, Wollaston prisms often have small cross sections, and vignetting can become a problem. This was the case with the first prism we tried, and a Zemax model (shown in Figure 4) of this prism

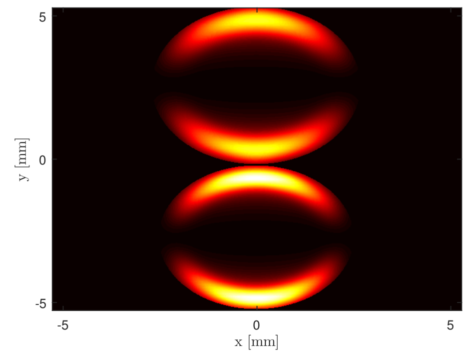

Figure 3: Ideal alignment of 2 radiation patterns on the camera (the actual radiation patterns are not necessarily realistic). The focal length of the Bertrand lens determines the size of each image, and the wedge angle of the prism determines the vertical spacing between the two images.

clearly shows the issue. With some modeling in Zemax, however, we found a combination that works. Based on this modeling, we purchased a custom Wollaston prism from Karl Lambrecht Corporation (part #WQ-19-1 with a 45 degree cut angle, 19 mm clear aperture, 21 mm length, 21mm height, and 20 mm width).

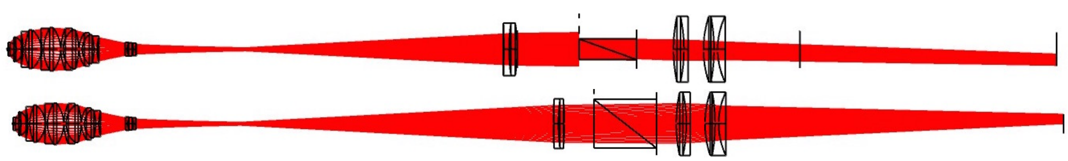

Figure 4: (top) Zemax model of the Fourier imaging setup with a Wollaston prism that is too small. The prism clearly cuts out a large portion of the light, especially at higher angles. (bottom) Corrected System.

4.3 Ray Tracing in Zemax with Matlab

Most simulations in Zemax involve relatively simple ray tracing calculations combined with comprehensive catalogs of commercial optical components and their properties (refractive index, dispersion, radii of curvature, thickness, etc.). External programmatic control of the software takes advantage of these catalogs and ray tracing capabilities while gaining more control over the initial conditions and analysis of the traces. With this incentive, I wrote some code in Matlab to automate the ray tracing and data acquisition in Zemax.

4.3.1 Ray Tracing Protocol

I used this code to develop an intuitive way to characterize the performance of the lab's Fourier imaging setups. My short program takes in a calculated radiation pattern like those in Figure 1, traces it through the optical system, and reconstructs the image that appears (theoretically) on the camera chip. The general procedure is as follows:

1. Define several points on a source plane at various distances from the optical axis. These points correspond to point emitters on the sample.

2. With each point on the source, define a series of rays uniformly distributed over all relevant emission angles (a relevant angle is one that will actually make it into the first surface in the objective).

3. Weight the intensity of each ray according to a calculated radiation pattern.

4. Trace the rays in Zemax to see where they land on the image.

5. Bin the rays on the image plane and use both their density on the image plane and original intensity weights to construct a final image.

The main subtlety in this approach is that it encodes the intensity information in the final image in two different ways. The first is the intensity weighting of each ray based on its launch angle from the sample (determined by a previously calculated radiation pattern). The second piece of intensity information comes from the density of the rays. It would be cumbersome and more computationally intensive to encode the angular intensities as densities of rays at each angle, so I don't. Instead I separate the two different sets of intensity information, run the traces, then simply add them up at the end. To obtain intensity information based on changes in the ray *density* by the optical system (i.e. distortion), I need to avoid imposing any density-related structure on the initial distribution of rays. From each source, I define rays with a uniform angular distribution. The angular intensity information is already contained in the intensity weights at the beginning, so I have to treat all angles equally when setting up the rays to trace. All of the commented code for the ray traces is in Appendix A.

4.3.2 Visualization of Distortion and Defocus

The ray tracing tool is convenient because its output is an image–the same thing we measure in the lab. This allows for an intuitive yet quantitative evaluation of optical systems, especially for geometric aberrations like distortion. An example is shown in Figure 5. The optical system clearly distorts the ideal input image, and defocusing the Bertrand lens (on the scale of 10 mm) changes the output image significantly.

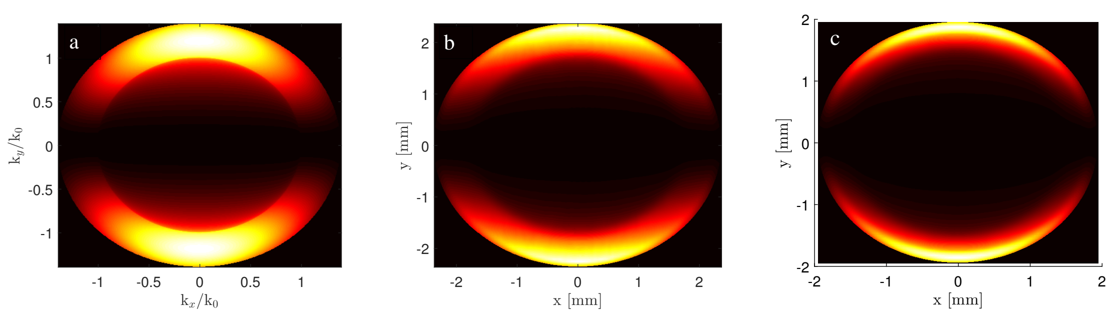

Figure 5: (a) Input radiation pattern to the ray tracing program. (b) Output image at focus. (c) Output image with the Bertrand lens 10mm out of focus.

4.3.3 Vignetting Visualization

A full simulation of the optical system also allows for an intuitive characterization of simple effects like vignetting. One scenario in which vignetting can be problematic occurs when emitters are slightly off-axis from the objective Figure 6 shows the simulated radiation patterns images by the microscope for emitters at different distances from the optical axis. After the emitters get about 200 μm off the optical axis, very little of light makes it to the image plane.

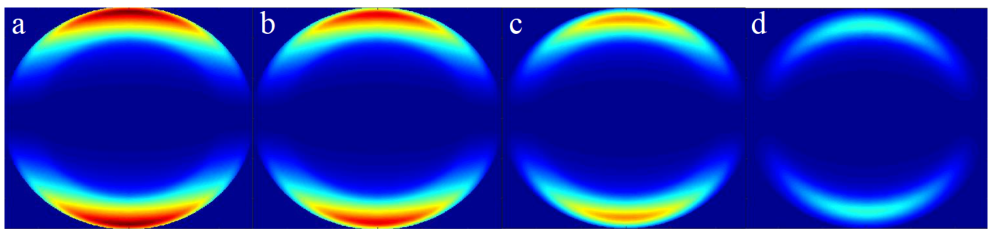

Figure 6: Simulated radiation patterns for emitters at different distances from the optical axis. (a) 10 μm (b) 50 μm (c) 100 μm (d) 180 μm.

5 Experiment and Results

5.1 Polarized Raman Spectroscopy from Bulk and Layered MoS_2

Molybdenum disulfide, or MoS_2, is an interesting material for many reasons, most of which lie outside the field of Raman spectroscopy. However, we deemed its Raman spectrum a good candidate for momentum-resolved Raman scattering measurements, for two main reasons. First, it has several well-defined Raman peaks (which would allow us to study momentum differences between transitions in the same system), but not so many peaks that it becomes difficult to isolate one from another. Second, its Raman transitions obey different polarization selection rules [4], which make it easy to isolate each transition by adjusting the polarization of the incident and scattered light. Figure 8 demonstrates these polarization selection rules–the ratio of the 386 cm^{-1} to the 412 cm^{-1} peak clearly changes with polarization.

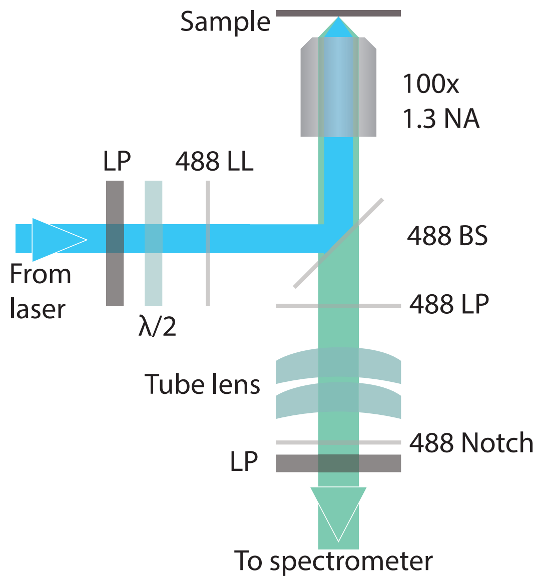

Figure 7: (left) Optical setup for polarized Raman measurements. (right) Setup for angle-resolved measurements

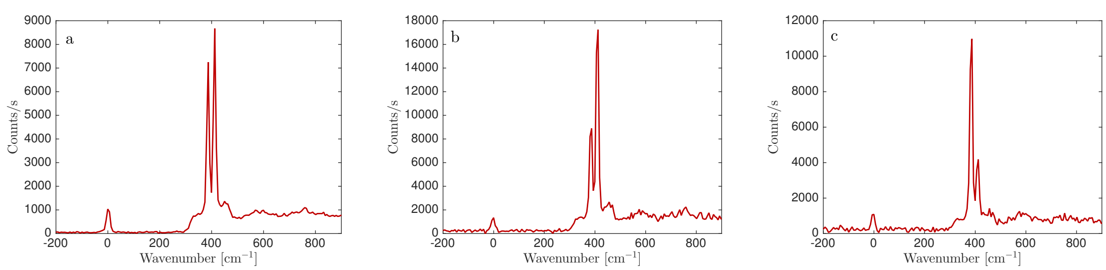

Figure 8: Raman spectra of bulk MoS_2 for different polarization configurations using 488 nm excitation. (a) unpolarized (b) incident polarization parallel to scattered polarization (c) incident polarization perpendicular to scattered polarization.

We also performed a more complete polarization study of the Raman spectrum as a function of input and output polarization. With an eye toward angle-resolved measurements (which require thin samples), we repeated this measurement with a few-layered thin sample of MoS_2 which gave similar results. Figure 9 shows the data, which shows a clear polarization dependence of the peak ratio. The high-energy peak is stronger when the input and output polarizations are parallel, and the low-energy peak is stronger when they are perpendicular.

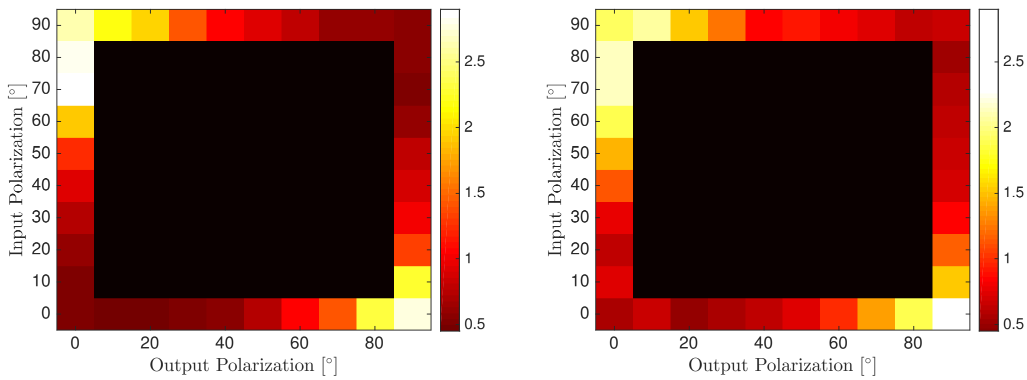

Figure 9: Ratio of the 386 cm^{-1} to the 412 cm^{-1} Raman peak as a function of input and output polarization. (left) bulk sample (right) few-layered sample.

6 Prospects of Raman Spectroscopy in a Fourier Microscope

The ultimate goal of this work was to use polarization effects in the Raman spectrum of MoS_2 to isolate each Raman peak and observe its angle-resolved radiation pattern. While angle-resolved measurements using Fourier optics have been done several times in the case of photoluminescence, the nature of Raman scattering complicates the measurement for two main reasons. The first is that Raman scattering is an inherently weak process, and this can make it difficult to get enough signal to angle resolve any peaks. Peaks that were once spread across only a few pixels in the spectrum get spread across many more pixels to form the radiation pattern. Second, since Raman scattering has a clear momentum conservation requirement, the radiation pattern should depend strongly on the **k** of the incoming light. This suggests that a traditional illumination scheme that focuses incident light on the sample (and hence excites the sample with a broad set $\mathbf{k}$ values) of will not work, because it imposes additional structure on the problem that is not related to the momentum transfer in the Raman process. An ideal experiment would excite the sample with a single $\mathbf{k}$, and that requires focusing the incident laser inside the objective, producing a roughly collimated excitation.

While this works in theory, this causes two problems in my particular setup that have been hard to overcome so far. Since the incident light is no longer focused on the sample (which boosts the intensity that drive Raman transitions inside the crystal), the Raman signal gets significantly smaller. Focusing the high energy 488 nm laser inside the objective also causes the objective to emit light, which resulted in a lovely background signal that completely dominates the Raman lines of MoS_2. These two effects of back focal plane illumination build on each other, and make it difficult to perform angle resolved measurements. Future efforts will likely include using the stronger circular polarization selection rule of MoS_2 and more intelligent background subtraction to search for an elusive angle-resolved Raman signal.

References

[1] D.L. Rousseau, R.P. Baumann, S.P.S. Porto, *J. of Raman Spectrosc.* **10**, 253 (1981).

[2] W. Hayes, R. Loudon: *Scattering of Light by Crystals*, (Wiley, New York 1978).

[3] W.H. Weber, R. Merlin: *Raman Scattering in Materials Science*, (Springer, New York 2000).

[4] Shao-Yu Chen, Changxi Zheng, Michael S. Fuhrer, Jun Yan, *Nano Letters* **15**, 4 (2015).

A Matlab Code

```
function [N_BFP,N_angles,Nr,numBins,clims,aperSize,stopDist,objSize,rotationAngles,
    offsets] = ...
    initializeTraceParameters(defocus,sourceRadii,N_sources,octMaskOption)

% ALL LENGTHS ARE IN LENS UNITS--whatever units you're using in Zemax.
% These will usually be mm.

moveBertrandLens(defocus);

N_BFP = 400;        % Grid Size for the caluclated radiation pattern (~200 is good)
N_angles = 6;       % Number of points on the ring with the smallest r (~6-8 is good
    )
Nr = 100;            % Number of radius values on pupil to sample (~90 is good)
numBins = 100;      % binnedIntensityMap has numBins X numBins bins (~100 is good)
clims = 0;          % colormap values for imagesc (clims=0 will scale automatically)

aper = zGetSystemAper;
aperSize = aper(3);                    % Aperture semi-diemeter [lens units]
stopSurface = aper(2);
objSize = zGetSurfaceData(0,5);        % Semi-diameter of object [lens units]

stopDist = 0;                          % Distance from object to stop surface [lens
    units]
for i = 0:stopSurface-1
    stopDist = stopDist + zGetSurfaceData(i,3);
end

% No rotational symmetry for the octagon, so just do 4 points per source
% radius
if octMaskOption == 1  &&  N_sources >= 3
    N_sources = 4;
    warning('only 4 sources per radius traced when using octmask')
```

```
end
rotationAngles = transpose(linspace(0,2*pi,N_sources+1));

% Source locations on the object surface
offsets = zeros(N_sources,2,length(sourceRadii));
for i = 1:length(sourceRadii)
    offsets(:,:,i) = sourceRadii(i)*[cos(rotationAngles(1:end-1)), sin(
        rotationAngles(1:end-1))];
end

end
%-----------------------------------------------------------------------
function moveBertrandLens(displacement)

% x = 179.2;
x = 217.3;
zSetSurfaceData(24,3,x + displacement);
% zSetSurfaceData(44,3,47.172 - displacement);
zPushLens(5);

end
```

```
function traceRadiationPattern_ver8(defocus,sourceRadii,N_sources,plotLimit,plotCode
    ,octMaskOption)
% MUST HAVE AN OPEN DDE CHANNEL TO ZEMAX BEFORE RUNNING!!! (use zDDEInit)
% Aperture in Zemax should be set to 'Float by Stop Size'!!


% This function starts with a radiation pattern generated by the parameters
% in 'bfpTestScript'. It then defines a set of rays that would be produced by
% point sources on the object surface emitting uniformily in angles. The
% locations of the point sources on the object surface are determined by
% sourceRadii and N_sources (you will have N_sources points at each
% sourceRadii value.

% After defining the rays, the function assigns an intensity value to each
% ray and traces the rays in Zemax. It then bins the rays at the image
% surface, and adds up the intensities in each bin to generate a final image.

% INPUTS:
%   -defocus: displacement of the bertrand lens. you'll have to set the
%       parameters in 'moveBertrandLens' to make sure you're defocusing at
%       the correct spot.
%   -sourceRadii: the radius values on the object surface where you'd like
%       to place the emitters
%   -N_sources: number of sources at each sourceRadii value
%   -plotLimit: sets the bound for the final plot. The xlim and ylim will
%       be -plotLimit to plotLimit. If plotLimit = 0, then it will autoscale
%   -plotCode: 0 gives a radiation pattern, 1 gives a cross section, 2
%       gives both
%   -octMaskOption: 0 does nothing, 1 puts an octagonal mask in k space.
%       Note that only 4 sources per radius will be traced for the octMask
%
% OUTPUT:
%    a figure showing the traced radiation pattern on the image plane


%% STEP 1: Initialization
```

```
[N_BFP, N_angles, Nr, numBins, clims, aperSize, stopDist, objSize, rotationAngles, offsets] = ...
    initializeTraceParameters(defocus, sourceRadii, N_sources, octMaskOption);

%% STEP 2: Define and throw the rays. Associate a (ux,uy) value with each ray.
[ux,uy,x,y] = traceAllRays(sourceRadii, rotationAngles, N_angles, Nr, ...
    aperSize, objSize, stopDist, offsets, octMaskOption);

%% STEP 3: Use the calculated radiation pattern values to associate intensities with each (ux,uy)
Ivals = matchIntensities(N_BFP, ux, uy, octMaskOption);
uyMax = max(uy);
uyMin = min(uy);
clear ux uy

%% STEP 4: Bin the (x,y) points and add intensities within each bin.
heat = binnedIntensityMap(x, y, Ivals, numBins);

%% STEP 5: Process the image
[finalImage, axis] = processBinnedImage(x, y, heat, octMaskOption);

%% STEP 6: Plot
if plotLimit == 0
    plotLimit = max(abs([x;y]));
end
plotBFPImage(axis, finalImage, plotLimit, defocus, sourceRadii, [uyMin, uyMax], clims,
    plotCode);

end
```

```
function [urange, radPattern] = bfpTestScript(N) %

% Just sets up the 'Multipole_BFP...' function to get a radiation pattern
% so I don't mix up all of the inputs. This assumes a square grid of kx,ky,
% and N is the number of points along one axis of the grid.

xpolOutput = 1;
urange = linspace(-1.39, 1.39, N);
n0 = 1;
n1 = 1;
n20 = 1.7;
n2e = 1.7;
n3 = 1.5;
l = 0;
s = 10;
d = 10;
lambdarange = 500;

field = Multipole_BFP_3D_Fields_v0p12('MD', xpolOutput, urange, urange, n0, n1, n20, n2e, n3
    , l, s, d, lambdarange, 0);

edx = 0;
edy = 0;
edz = 0;
mdx = 0;
mdy = 0;
mdz = 1;

```

```
pol = 0;

if pol == 0
    radPattern = edx*abs(field.xpol.EDx).^2 + edy*abs(field.xpol.EDy).^2 + edz*abs(
        field.xpol.EDz).^2 + ...
        mdx*abs(field.xpol.MDx).^2 + mdy*abs(field.xpol.MDy).^2 + mdz*abs(field.xpol
            .MDz).^2;
elseif pol ==1
    radPattern = edx*abs(field.ypol.EDx).^2 + edy*abs(field.ypol.EDy).^2 + edz*abs(
        field.ypol.EDz).^2 + ...
        mdx*abs(field.ypol.MDx).^2 + mdy*abs(field.ypol.MDy).^2 + mdz*abs(field.ypol
            .MDz).^2;
end
figure;
imagesc(urange, urange, radPattern);
set(gca,'YDir','normal');
xlabel('k$_{x}$/k$_{0}$','Interpreter','latex');
ylabel('k$_{y}$/k$_{0}$','Interpreter','latex');
title('Raw Radiation Pattern','Interpreter','latex');
colormap('hot')
%
% figure;
% xSection = radPattern(:, round(N/2));
% % p = cos(asin(urange/1.5));
% plot(urange, xSection/max(xSection));
% xlabel('ky/k0')

end
```

```
function [ux,uy,x,y] = traceAllRays(sourceRadii,rotationAngles,N_angles,Nr,aperSize,
    objSize,stopDist,offsets,octMaskOption)

% Function to trace the rays from all of the sources defined in
% 'initializeTraceParameters'.
%
% INPUTS:
%     -sourceRadii: the radius values on the object surface where you'd like
%         to place the emitters
%     -rotationAngles: array of angles by which you rotate each source.
%     -N_angles: NOT the length of rotationAngles--this has to do with the
%         rays defined on the pupil. N_angles is the number of pupil points
%         at the first nonzero radius value. This value gets scaled by the
%         radius on the pupil as the radius increases.
%     -Nr: number of radius values to sample on the pupil.
%     -aperSize: aperture semi-diemeter [lens units]
%     -objSize: semi-diameter of the object surface [lens units]
%     -stopDist: distance from object to stop surface [lens units]
%     -offsets: N_sources x 2 x length(sourceRadii) array that defines the
%         coordinates of all of the sources.
%     -octMaskOption: 0 does nothing, 1 puts an octagonal mask in k space.
%         Note that only 4 sources per radius will be traced for the octMask
%
% OUTPUT:
%     -ux,uy: coordinates in normalized k space
%     -x,y: coordinates on the image surface

N_sources = length(rotationAngles) - 1;     % don't include redundant 2pi

```

```matlab
UXcell = cell(1,length(sourceRadii));
UYcell = cell(1,length(sourceRadii));
Xcell = cell(1,length(sourceRadii));
Ycell = cell(1,length(sourceRadii));

% Throw a ring of rays for each source radius
for i = 1:length(sourceRadii)
    [px,py,~,~,hx,hy] = generateRayTraceInputs(N_angles,Nr,aperSize,stopDist,objSize
        ,offsets(1,:,i),octMaskOption);

    % Throw rays and delete vignetted ones
    [x,y,vignetted,imageCenter] = shootRays(px,py,hx,hy);
    x  = x(~vignetted);
    y  = y(~vignetted);
    px = px(~vignetted);
    py = py(~vignetted);
    imageCenter

    % Each column has coodinates for each source on the ring
    [PX,PY] = generateRotatedCoordinates(px,py,N_sources,rotationAngles,[0,0]);
    [UX,UY] = pupil2u_matrix(PX,PY,aperSize,stopDist,offsets);
    [XX,YY] = generateRotatedCoordinates(x,y,N_sources,rotationAngles,imageCenter);

    % The number of rays thrown changes with the source radius, so I store
    % each matrix in a cell.
    UXcell{i} = UX;
    UYcell{i} = UY;
    Xcell{i}  = XX;
    Ycell{i}  = YY;
end

ux = [];
uy = [];
x  = [];
y  = [];
for i = 1:length(sourceRadii)
    ux = [ux; UXcell{i}(:)];
    uy = [uy; UYcell{i}(:)];
    x  = [x; Xcell{i}(:)];
    y  = [y; Ycell{i}(:)];
end

end
%------------------------------------------------------------------------
function [px,py,ux,uy,hx,hy] = generateRayTraceInputs(N_angles,Nr,aperSize,stopDist,
    objSize,offset,octMaskOption)

% Function to generate all of the inputs needed for the ray traces in Zemax.
% INPUTS:
%     -N_angles: number of angles sampled at the first nonzero radius value on
%      the pupil
%     -Nr: number of radii sampled
%     -aperSize: semi-diameter of the aperture stop surface
%     -stopDist: distance from the object to the stop surface
%     -objSize: the semi-diameter of the object surface
%     -octMaskOption: 0 does nothing, 1 puts an octagonal mask in k space
%
% OUTPUTS:
```

```
%     -px,py: normalized pupil corrdinates that determine where the rays hit
%       the stop surface
%     -ux,uy: normalized k vector components u = n*k/k0
%     -hx,hy: normalized object coordinates (determine the xy shift in the
%       object plane)


s = size(offset);
px = [];
py = [];
ux = [];
uy = [];
hx = [];
hy = [];

for i = 1:s(1)
    xOffset = offset(i,1);
    yOffset = offset(i,2);

    % Find image pupil that results from uniform angular emission
    [ppxx, ppyy] = generatePupilImage(N_angles, Nr, aperSize, stopDist, offset(i,:));
    px = [px; ppxx'];
    py = [py; ppyy'];

    % Convert determined pupil coordinates to u coordinates for later use
    [uuxx, uuyy] = pupil2u(ppxx,ppyy,aperSize,stopDist,xOffset,yOffset);
    ux = [ux; uuxx'];
    uy = [uy; uuyy'];

    % Store the shifts for later use
    hx = [hx; ones(length(ppxx),1)*xOffset/objSize];
    hy = [hy; ones(length(ppyy),1)*yOffset/objSize];
end

% Optional octagonal mask
if octMaskOption == 1
    idx = kSpace_octMask(51,0.5,ux,uy);
    ux = ux(idx);
    uy = uy(idx);
    px = px(idx);
    py = py(idx);
    hx = hx(idx);
    hy = hy(idx);
end

end
%-----------------------------------------------------------------------
function  idx = kSpace_octMask(gridSize,normalizedRadius,ux,uy)

% Function to put an octagonal mask over the k space radiation pattern
% INPUTS:
%     -gridSize: number of points inside the octagon (must be multiple of 3)
%     -normalizedRadius: the "radius" of the octagon in k space, as a fraction
%       of 1.5
%     -ux,uy: normalized k vector values u = n*k/k0
%
% OUTPUT:
```

```
%     -idx: indicies for ux,uy at which the points fall inside the mask

Noct = gridSize;
uRadius = normalizedRadius;

% Make an octagon
oct = strel('octagon',Noct);
oct = oct.getnhood;

% Put that octagon in the center of a bigger array
octAxis = linspace(-1.5*uRadius,1.5*uRadius,length(oct));
newsize = round(1/uRadius*length(octAxis));
n = round(0.5*(newsize - length(octAxis)));
mask = zeros(newsize,newsize);
mask(n:n+length(oct)-1, n:n+length(oct)-1) = oct;
uAxis = linspace(-1.5,1.5,length(mask));

% Get the mask in vector form (to match the actual (ux,uy) coordinates)
mask2 = interp2(uAxis,uAxis,mask,ux,uy);
idx = find(mask2==1);

end
%-----------------------------------------------------------------------
function [px,py] = generatePupilImage(N_angles,N_radii,aperSize,stopDist,offset)

% Function to generate all the pupil positions in Zemax
% INPUTS:
%     -N_angles: number of angles sampled at the first nonzero radius value on
%      the pupil
%     -N_radii: number of radii sampled
%     -aperSize: semi-diameter of the aperture stop surface
%     -stopDist: distance from the object to the stop surface
%     -objSize: the semi-diameter of the object surface
%
% OUTPUTS:
%     -px,py: normalized pupil corrdinates that determine where the rays hit
%      the stop surface


xOffset = offset(1);
yOffset = offset(2);

% Account for an off-axis emitter
alpha = sqrt(xOffset^2 + yOffset^2) + aperSize;

% Space radii values on pupil according to uniform angular emission
thetaMax = atan(alpha/stopDist);
qr = stopDist*tan(linspace(0,thetaMax,N_radii));

% Do qr = 0 separately
qx = 0;
qy = 0;

% Then do the rest
for i = 2:length(qr)
    nTheta = round(N_angles*qr(i)/qr(2));
    angles = linspace(0, 2*pi, nTheta);
    qx = [qx, qr(i)*cos(angles)];
```

```
    qy = [qy, qr(i)*sin(angles)];
end

% Account for an off-axis emitter
px = (qx - xOffset)/aperSize;
py = (qy - yOffset)/aperSize;

% Cut values outside a unit circle (otherwise you'd miss the stop surface)
circleCut = px.^2 + py.^2 <= 1;
px = px(circleCut);
py = py(circleCut);

end
%--------------------------------------------------------------------------
function [x,y,vignetted,imageCenter] = shootRays(px,py,hx,hy)

% Function that shoots a bunch of rays in Zemax.
% INPUTS
%     -px,py: normalized pupil corrdinates that determine where the rays hit
%      the stop surface
%     -hx,hy: normalized object coordinates (determine the xy shift in the
%      object plane)
%
% OUTPUTS
%     -x,y: positions at which the rays land on the image surface


x = zeros(length(px),1);
y = zeros(length(px),1);
vignetted = zeros(length(px),1);

imageCenter = zeros(1,2);
rayTraceData = zGetTrace(1,0,-1,0,0,0,0);
imageCenter(1) = rayTraceData(3);        % These are real space coordinates, in
imageCenter(2) = rayTraceData(4);        % whatever units you're working with in Zemax.


for i = 1:length(px)
    rayTraceData = zGetTrace(1,0,-1,hx(i),hy(i),px(i),py(i));
    x(i) = rayTraceData(3);        % These are real space coordinates, in
    y(i) = rayTraceData(4);        % whatever units you're working with in Zemax.
    if rayTraceData(2) ~= 0
        vignetted(i) = 1;
    else
        vignetted(i) = 0;
    end
end

end
%--------------------------------------------------------------------------
function [XX,YY] = generateRotatedCoordinates(x,y,N_sources,rotationAngles,center)

% Takes a set of (x,y) coordinates on the image plane and duplicates them at
% different angles. The results are 2 matricies--the columns of
% the matricies give the x and y coords for the points at each rotation
% angle
%
```

```
% INPUTS:
%     -x,y: coordinates on the image plane from a single-point trace
%     -N_sources: the number of sources (each emitting from a different spot)
%     -rotationAngles: the angles that determine where the sources are
%
% OUTPUTS:
%     -XX,YY: length(x) X N_sources matricies giving the all the xy coords

XX = zeros(length(x),N_sources);
YY = zeros(length(x),N_sources);
XX(:,1) = x;
YY(:,1) = y;

T = [1,0,-center(1);0,1,-center(2);0,0,1];
Tinv = inv(T);

if isequal(center,[0,0])

    v = [x';y'];
    for i = 2:N_sources
        R = [cos(rotationAngles(i)) -sin(rotationAngles(i));...
            sin(rotationAngles(i)),cos(rotationAngles(i))];
        v2 = R*v;
        x2 = v2(1,:);
        y2 = v2(2,:);
        XX(:,i) = x2';
        YY(:,i) = y2';
    end

else

    v = [x';y';ones(1,length(x))];
    for i = 2:N_sources
        R = [cos(rotationAngles(i)),-sin(rotationAngles(i)),0;...
             sin(rotationAngles(i)),cos(rotationAngles(i)), 0;...
             0,0,1];
        v2 = Tinv*R*T*v;
        x2 = v2(1,:);
        y2 = v2(2,:);
        XX(:,i) = x2';
        YY(:,i) = y2';
    end

end
end
%------------------------------------------------------------------------
function [UX,UY] = pupil2u_matrix(PX,PY,aperSize,stopDist,offsets)

% Converts pupil coords to ux,uy coords

s = size(PX);

UX = zeros(s(1),s(2));
UY = zeros(s(1),s(2));

for i = 1:s(2)
    [uuxx, uuyy] = pupil2u(PX(:,i),PY(:,i),aperSize,stopDist,offsets(i,1),offsets(i,2));
```

```
    UX(:,i) = uuxx;
    UY(:,i) = uuyy;
end

end
%-------------------------------------------------------------------------
```

```
function [ux,uy] = pupil2u(px,py,a,d,xOffset,yOffset)

% Converts from normalized pupil coordinates to normalized k space
% coordinates. px and py are corresponding vectors for the pupil
% coordinates, a is the aperture stop semi diameter, and d is the distance
% from the object to the stop surface.

cx = (px*a - xOffset)/d;
cy = (py*a - yOffset)/d;
n = 1.5;

ux = n* cx./sqrt(1 + cx.^2 + cy.^2);
uy = n* cy./sqrt(1 + cx.^2 + cy.^2);

end
```

```
function Ivals = matchIntensities(N_BFP,ux,uy,octMaskOption)

% Function to assign intensity values to a set of ux,uy values
% INPUTS
%     -N_BFP: grid size for the caluclated radiation pattern
%     -ux,uy: k vector components that you want intensities for
%     -octMaskOption: 0 is standard, 1 makes intensities uniform
% OUTPUT:
%     -Ivals: vector of intensity values corresponding to ux,uy


if octMaskOption == 1
    Ivals = ones(length(ux),1);
else
    [urange, radPattern] = bfpTestScript(N_BFP);
    Ivals = transpose(interp2(urange,urange,radPattern,ux,uy));
end

end
```

```
function heat = binnedIntensityMap(xVals,yVals,I,gridSize)

% Takes a bunch of scattered points (x,y) and associated intensity values
% I, divides them into a grid of bins determined by N, and adds up
% the intensities in each bin.

totalMax = max(abs([xVals;yVals]));

xAxis = linspace(-totalMax,totalMax, gridSize);
yAxis = linspace(-totalMax,totalMax, gridSize);
heat = zeros(gridSize-1,gridSize-1);

for i = 1:gridSize-1
    yidx = find(yVals >= yAxis(i) & yVals <= yAxis(i+1));
    xOptions = xVals(yidx);
```

```
    IOptions = I(yidx);

    for j = 1:gridSize-1
        xidx = find(xOptions >= xAxis(j)  &  xOptions <= xAxis(j+1));
        if isempty(xidx)
            heat(i,j) = 0;
        else
            heat(i,j) = sum(IOptions(xidx));
        end
    end
end

% figure;
% imagesc(xAxis,yAxis,heat)
% set(gca,'YDir','normal')
% colormap('jet')
% xlabel('x [mm]','Interpreter','latex');
% ylabel('y [mm]','Interpreter','latex');
end
```

```
function [newImage,axis] = processBinnedImage(x,y,heat,octMaskOption)

xmin = min(x); xmax = max(x); ymin = min(y); ymax = max(y);
totalMax = max(abs([xmin,xmax,ymin,ymax]));

% Smooth the signal by averaging over every 5x5 pixel box
h = 1/25*ones(5);
heat2 = filter2(h,heat);
q2 = linspace(-totalMax,totalMax,length(heat));
[XX, YY] = meshgrid(q2,q2);

clear heat

% Interpolate over a finer grid
N3 = 500;
axis = linspace(-totalMax,totalMax,N3);
[Xq, Yq] = meshgrid(axis,axis);
heat3 = interp2(XX,YY,heat2,Xq,Yq);

% Don't mask the octagon
x0 = (xmin + xmax)/2;
y0 = (ymin + ymax)/2;
a = abs(xmax - x0);
b = abs(ymax - y0);

% Scale the intensities to the area of the image. This corrects for the
% fact that there is a constant number of bins, regardless of the image
% size.
A = a*b;
newImage = heat3/A;

if octMaskOption == 0
    mask = (Xq - x0).^2/a^2 + (Yq - y0).^2/b^2 < 1;
    newImage = heat3.*mask;
end

end
```

```
function [xSec,uAxis] = plotBFPImage(q3,heat3,plotLimit,defocus,sourceRadii,uLims,
    clims,plotCode)

if (plotCode == 0  ||  plotCode == 2)
    figure;
    if clims == 0
        imagesc(q3,q3,heat3)
    else
        imagesc(q3,q3,heat3,clims)
    end

    set(gca,'YDir','normal')
    colormap('hot')
    xlabel('x [mm]','Interpreter','latex');
    ylabel('y [mm]','Interpreter','latex');
%     colorbar;
    t = plotLimit;
    xlim([-t,t]);
    ylim([-t,t]);
    titleStr = ['defocus: ',num2str(defocus),' mm'];
    title(titleStr,'Interpreter','latex');
%     set(gca, 'Color', [0,0,0.56])
end

xSec = 0;
uAxis = 0;

if (plotCode == 1  ||  plotCode == 2)
    figure;
    uAxis = linspace(uLims(1),uLims(2),length(q3));
    xSec = heat3(:,round(length(q3)/2));
    plot(uAxis,xSec,'LineWidth',2)
    xlabel('uy','Interpreter','latex');
    ylabel('Intensity [arb. units]','Interpreter','latex');
    titleStr = ['radius: ',num2str(sourceRadii*1e3),' $\mu$m'];
    title(titleStr, 'Interpreter','latex');
    if length(clims) > 1
        ylim(clims)
    end
    xlim(uLims)
end

end
```