An Investigation of the Optical Properties of a Water Droplet

Ethan Hu

May 30th, 2024

Honors Research in Physics – A2

Abstract

When a droplet of water sits on a glass surface, it acts as an imaging system. Here we created a two-dimensional raytracing program that allowed us to determine focal range of droplets, along with analyze possible aberrations. In addition, we were able to observe and quantify different forms of aberrations in the system.

Introduction

When small amounts of water are placed onto a glass surface, they tend to create a curved surface called a droplet. When light enters the droplet from below, the light converges to create an image. Therefore, this curved surface can act as a planoconvex lens, a lens that is curved on one side, and flat on the other. This lens has been used as a tool to investigate concepts such as focal length and the Lens-maker equation.[1][2] However, experiments are not able to accurately explain the aberrations we see in the images formed from this system.

In this paper, we primarily look at the focal range, magnification, and aberrations of the lens. Using a raytracing program, we were able to accurately predict the focal range of a water droplet. We then described the aberrations we saw in the raytrace graph using a wavefront polynomial and hope to show the existence of spherical aberration through this polynomial.

Theoretical Model

When light passes through different mediums, it refracts. If we imagine light to be rays of light, we know that light refracts according to Snell's Law, Eq. 1.[3]

$$n_1 \sin(\theta_1) = n_2 \sin(\theta_2)$$

Eq. 1

Where the index of refraction of the initial medium, n_1, multiplied by the sin of it's incident angle, θ_1, is equal to the index of refraction of the exiting medium, n_2, multiplied by the sin of the resulting angle, θ_2.

As the focal length is defined as the point at which light from infinitely far away converges, if we look at the intersection of rays of light from infinitely far away, we can determine the focal length of the system.

To calculate this focal length, our first model was the Lens-maker's Equation for plano convex lenses in air, Eq. 2.[3]

$$\frac{1}{f} = (n-1)\left(\frac{1}{R}\right)$$

Eq. 2

Where focal length, f, is a function of the index of refraction of the lens, n, and the radius of curvature, R.

We also decided to create a raytracing program that would calculate the intersection of rays of light from infinitely far away, calculating the resulting slope of each ray according to Snell's Law, Eq. 1.

To get the magnification of the image, our first model used the paraxial relationship of magnification to focal length, Eq. 5. We derived this relation using the mirror equation, Eq. 3, and the definition of magnification, Eq. 4.[4]

$$\frac{1}{\mathrm{f}} = \frac{1}{\mathrm{d_i}} + \frac{1}{\mathrm{d_o}}$$

Eq. 3

Where the focal length, f, is described by the image distance, $\mathrm{d_i}$, and the object distance, $\mathrm{d_o}$.[3]

$$M = \frac{h_i}{h_o}$$

Eq. 4

Where the magnification, M, is defined as the ration between the image height, $\mathrm{h_i}$, and the object height, $\mathrm{h_o}$.

$$M = \frac{f}{f - d_i}$$

Eq. 5

However, this model did not account for the effect of aberrations as the image deviates from the center. To account for aberrations, we decided to use the wavefront polynomial, Eq. 6.[4][5]

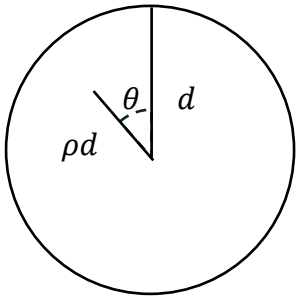

Figure 1: Diagram of Lens with Position of Ray Entering

$$W_{\rho,\theta} = s(\rho d)^4 + c\alpha(\rho d)^3\cos(\theta) + a\alpha^2(\rho d)^2\cos^2(\theta) + u\alpha^2(\rho d)^2 + g\alpha^3(\rho d)\cos(\theta)$$

Eq. 6

Where W is the wavefront error, which measure using the magnification at each pixel. s, c, a, u, and g are coefficients that stand for different types of aberrations(spherical, coma, astigmatism, field curvature, distortion). ρ is the relative distance from the center of focus. d is the pupil radius. Therefore, ρd is the sagittal displacement. α is the angle of the field. θ is the pupil angle.

The wavefront polynomial, Eq. 6, describes the displacement of an image, which allows us to describe the magnification at different points under the lens.

Data Collection

We created a variety of droplets by dropping small volumes of water onto glass slides that we cleaned using paper towels.

We found the focal lengths of varying droplets by focusing the light of various light sources through the droplet. For each experiment, we attached the water droplet onto a jack, or some device that could vary its elevation, and varied either the image distance or the object distance until the image created was focused. We also recorded a top and side view of each droplet to get its shape.

Initially, we used the ceiling light, due to its distinct light bulbs allowing us to gauge the convergence of the light. We would adjust the jack, as seen in figure 2, till the image was as small as possible without the lines from the ceiling light converging, as seen in figure 3. We recorded the image distance as the distance from the resulting image to the bottom of the droplet, and the object distance as the distance from the ceiling lights to the bottom of the droplet.

Figure 2: Experimental Setup 1

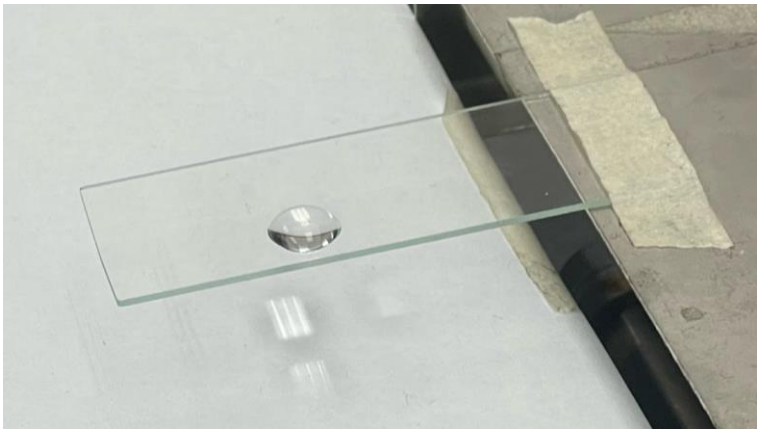

Figure 3: Image formed under ceiling lights

However, it was difficult to measure the image size due to the blurriness of the resulting image. We then tried using a resolution test target, due to its ability to give us both magnification and resolution, along with a laser, which was small enough to function as a point source. To generate an image from the laser, we looked at the pattern formed by the laser at different heights above the droplet. However, both experiments resulted in data that was imprecise and difficult to use.

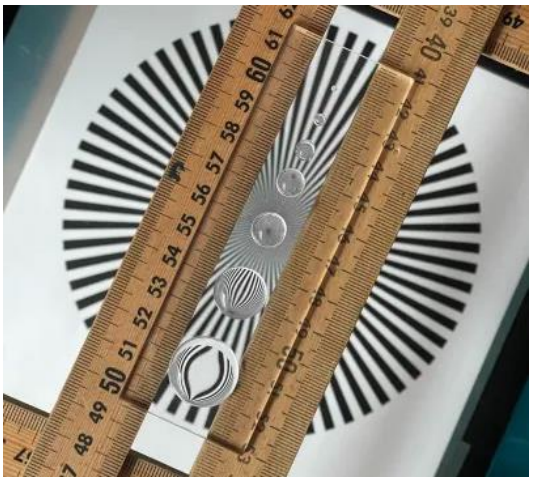

Figure 4: Image formed by Siemen's Star resolution test

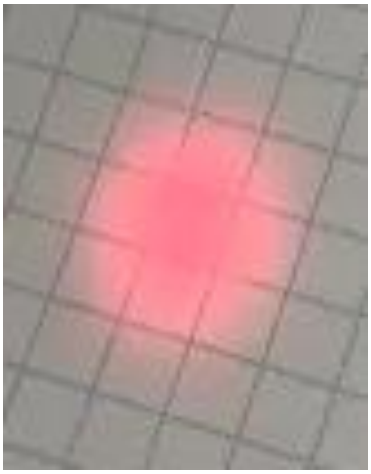

Figure 5: Image formed by laser

The data gained from the resolution test target was difficult to use, as different sections of the target would be resolved at different depths, resulting in difficulties in measuring magnification. The laser pointer was difficult to measure due to the difficulty in varying the height of the image, along with having issues with blurriness like the ceiling light.

For our next image source, we decided to use the pixel display of a Lenovo X380 Yoga. Our setup was similar to the one in figure 2. However, we used the jack to vary the position of the laptop instead of the droplet.

Figure 6: Experimental Setup 2

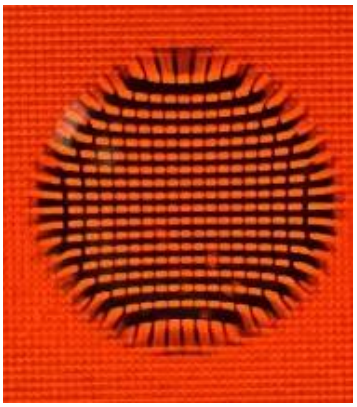

Figure 7: Image formed by pixel grid

We used Experimental Setup 2 to get our magnification data, as the pixels were clearly distinguishable at a variety of depths.

We also tested the effect of varying thickness of glass in comparison to the resulting magnification. To do this, we varied the number of glass slides underneath the droplet, and measured the resulting image distance and object distance.

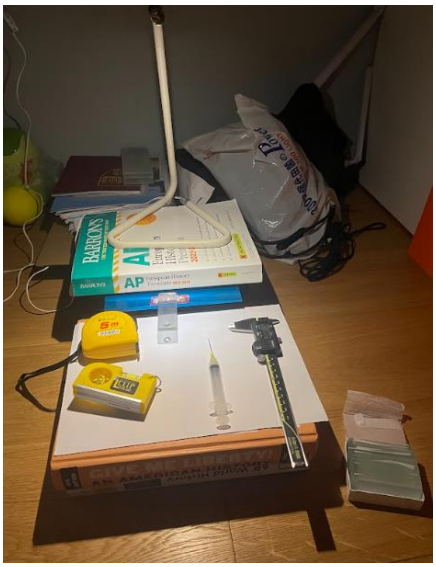

Figure 8: Experimental Setup 3

Experimental Setup 3 used a lamp light that was similar in function to the ceiling lights in Experimental Setup 1. As a result, we ran into similar issues when measuring magnification. However, we were able to get focal lengths for different glass thicknesses by varying the number of glass slides under the droplet.

Data Analysis

In order to find the focal lengths of different droplets as a function of shape, we fitted an elliptical equation to a water droplet. We took the horizontal derivative of each image, then isolated the points along the edge of the droplet using two parabolas as our boundaries. We then used these points to fit an elliptical equation. When using Lensmaker Equation, equation 2, as our model, we used a spherical fit instead of an elliptical one.

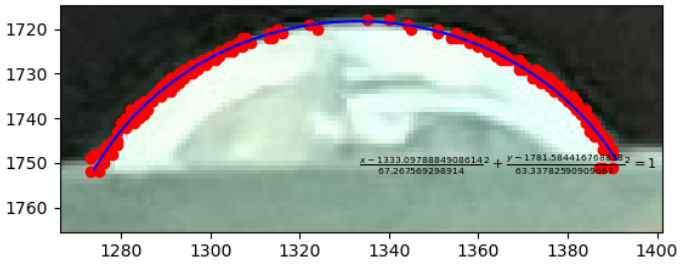

Figure 9: Side View of droplet with selected points and elliptical fit

For Experimental Setup 1, shown in figure 1, we compared the predictions of the Lensmaker Equation, equation 2, to the experimentally measured focal lengths in figure 9.

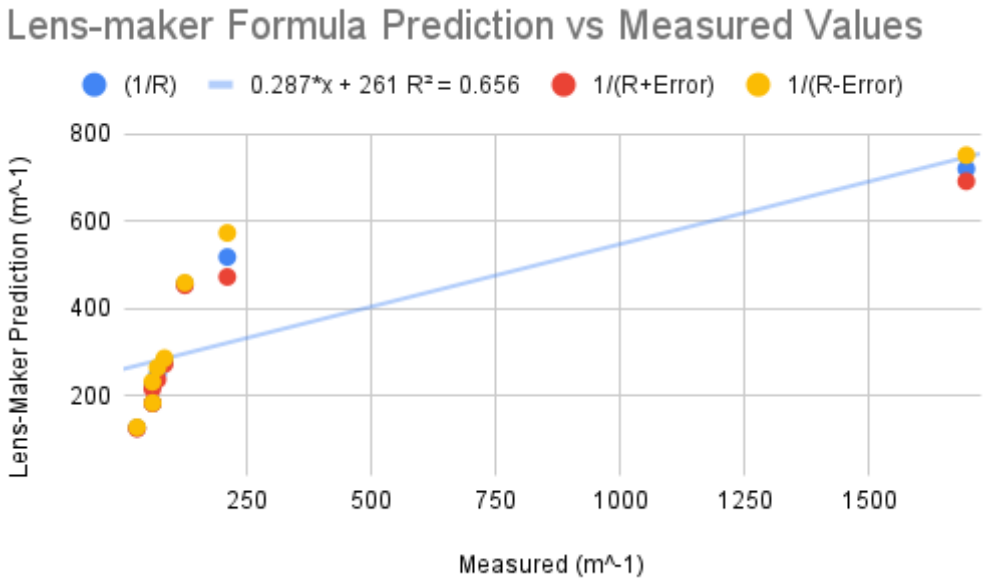

Figure 10: Lens-maker Formula Prediction vs Measured Values

As shown, the data clearly does not follow a linear trend at smaller focal lengths or greater inverse focal length predictions. This can be explained by the greater deviation from a perfect sphere at small volumes, shown in figure 10, violating the Lensmaker Equation assumption of a spherical plano convex lens.

Figure 11: Close up Image of Outlying Droplet

As such, we tried to compare the slope of the focal lengths only in the predicted inverse focal lengths.

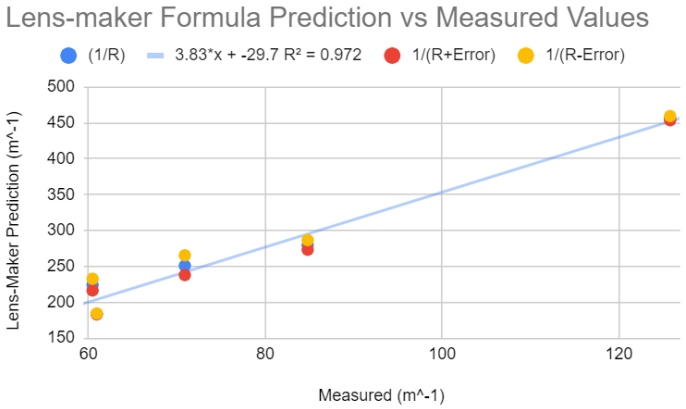

Figure 12: Inverse Radius vs Inverse Focal Length restricted to linear trend

However, the slope was still wrong. Rewriting the Lensmaker equation, equation 2, we see that slope should be $\frac{1}{n-1}$.

$$\frac{1}{R} = \frac{1}{n-1} * \frac{1}{f}$$

Eq. 7

Given our current slope, this gives a index of refraction of 1.26. The ideal index of refraction of water is 1.33, so we have a percent error of 5%.

However, this single focal length doesn't explain the aberrations that are clearly visible in the images. As shown in figure 13, magnification is not constant across the lens, it varies at different distances from the center. Different aspects of the image are also focused at different depths.

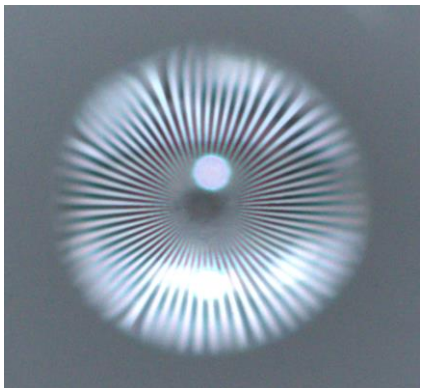

Figure 13: A Siemen Star imaged under a droplet

We decided to analyze the droplet using a raytracing program.

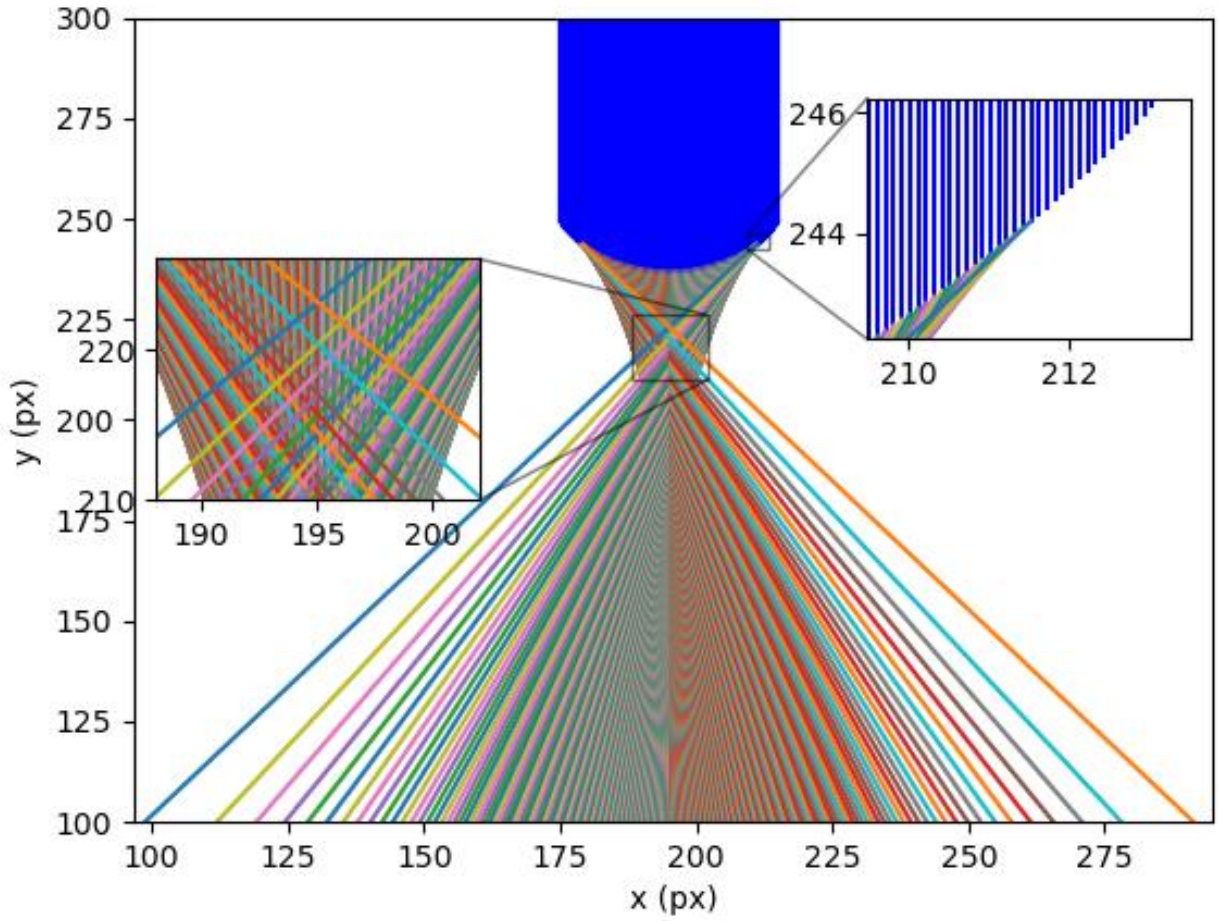

Figure 14: Raytracing plot; Closeup of intersection of lines; Closeup of point where total internal reflection occurs

Using the raytracing program, we calculated the range of focal lengths based on the intersections of the rays through the middle, depicted in the popup of figure 14. We then compared this to experimental focal lengths of the water droplet in figure 15.

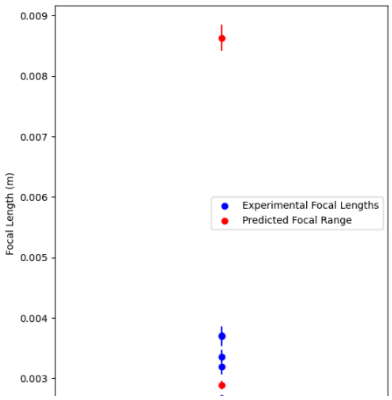

Figure 15: Comparing experimental and predicted focal lengths

As shown, most focal lengths lie within the range. The outlying focal length could be due to inaccuracies in measurements of the droplet, as it was the point at which the image started to get blurry.

To describe the aberrations more quantitatively, we decided to plot the magnification vs distance from center.

To get this data, we used ImageJ, a biological microscopy software, to find the positions and sizes of each pixel in figure 7. We then filtered it to get magnification of each point.

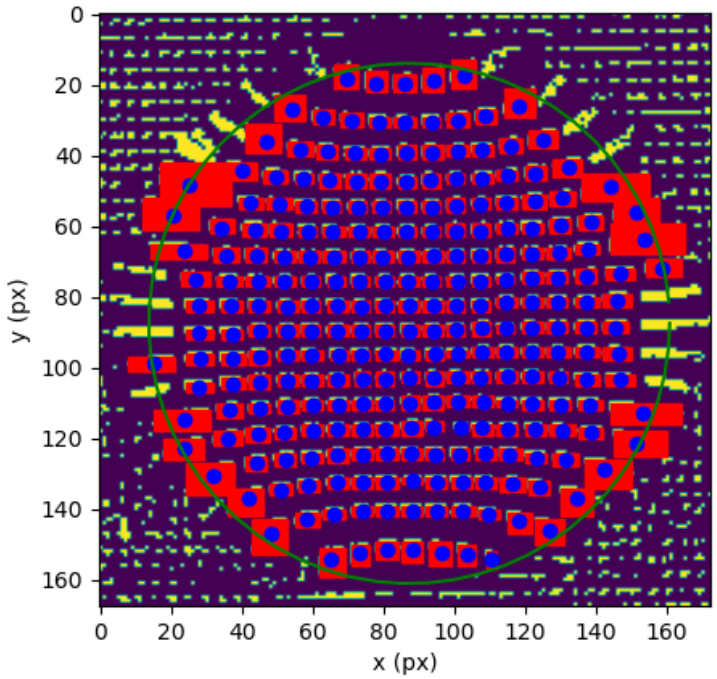

Figure 16: Figure 7 after being processed by ImageJ and filtered using python

Using the data, we plotted the magnification vs the distance the pixel was from the center. To find the center, we plotted the magnification vs coordinate position, shown in figure 16, and found where the minimum magnifications were, as our phone caused the optical center to be different from the geometric center.

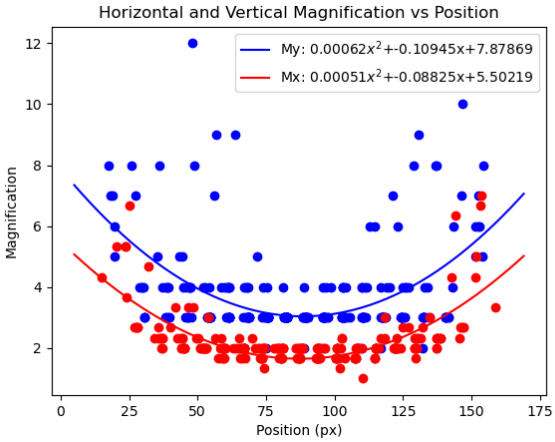

Figure 17: Magnification vs Position plots

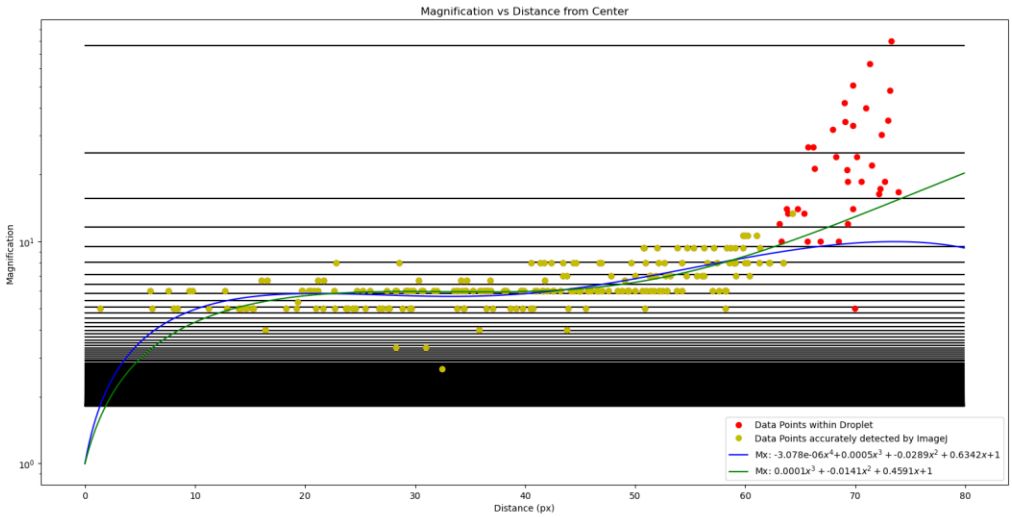

Figure 18: Magnification vs Distance from Optical Center Plot with lines representing magnification of focal lengths from raytracing

As seen in figure 18, there was not a single focal length that corresponded to the magnifications of the pixels in figure 7. Instead, using equation 6, we showed that there was spherical aberration in the lens, as a fourth degree term corresponds to spherical aberration.

Conclusion

In conclusion, our experimental data agrees with the lens maker equation within center boundaries of droplet size, and agrees with our raytracing diagram. Additionally, we are able to describe the aberrations in the droplet lens using the wavefront error equation, equation 6.

In future works, we would like to investigate the resolution of the lens. Using our raytracing program, we can determine the aperture of the lens, which lets us know the diffraction limit. Additionally, we can modify our program to generate a point spread function to determine other

resolution limiting factors. However, this would require smaller and more precise resolution test targets.

References

[1] Freeland, J., Krishnamurthi, V. R., & Wang, Y. (2020). Learning the lens equation using water and smartphones/tablets. *The Physics Teacher*, *58*(5), 360–361. https://doi.org/10.1119/1.5145539

[2] Myint, H. H., Marpaung, A. M., Kurniawan, H., Hattori, H., & Kagawa, K. (2001). Water droplet lens microscope and microphotographs. *Physics Education*, *36*(2), 97–101. https://doi.org/10.1088/0031-9120/36/2/301

[3] Smith, W.J. (2008) *Modern Optical Engineering: The design of Optical Systems*. New York i pozostałe: McGraw-Hill.

[4] Born, M. and Wolf, E. (1970) *Principles of optics; electromagnetic theory of propagation, interference, and diffraction of light, by Max born and Emil Wolf, with contributions by A.B. Bhatia et al.*. Oxford, NY,: Pergamon Press.

[5] Sacek, V. (2006) *3.5.1. Seidel aberrations, wavefront aberration function*, *telescopeOptics.net*. Available at: https://www.telescope-optics.net/Seidel_aberrations.htm (Accessed: 12 May 2024).

Acknowledgements

Thanks to Dr. Rafael Garcia for advice throughout the research process.

Thanks to Andrew Lin for allowing me to use parts of his analysis on the magnification throughout the water droplet.

Appendix

Appendix 1: Python Code for Curve Selection

```python
import numpy as np
import matplotlib.pyplot as plt
import scipy.optimize as op
#path = 'SV.JPEG'
path = 'Side2.WEBP'
#path = 'Nov15.bmp'
import PIL.Image as image

plt.close('all')

im =image.open(path)
drop=np.array(im) #Converts data into array
plt.imshow(drop) #plt is Python library that does matlab stuff
#drop = drop[:, 2600:3500]

red = drop[:,:,0] #As drop is a bit map, where [red, green, blue], can filter to only red by only having first index
#plt.figure()
#plt.imshow(red, cmap='gray')

green = drop[:,:,1]
#plt.figure()
#plt.imshow(green, cmap='gray')

blue = drop[:,:,2]
#plt.figure()
#plt.imshow(blue, cmap='gray')


grey = (drop[:,:,0] + drop[:,:,1] + drop[:,:,2])/3.0
#plt.figure()
#plt.imshow(grey, cmap='gray')
"""
grey[:1800,:] = 0
grey[2150:,:] = 0
"""

def dx(array):
    h,w = array.shape
    return array[:,1:w-1] - array[:,0:w-2]
dx_image = dx(grey)
dx_image[np.abs(dx_image)<20]=0
#meanBlurred =(grey[2:h-1,2:w-1]+grey[1:h-2,2:w-1]+grey[0:h-3,2:w-1]+grey[2:h-1,1:w-2]+grey[1:h-2,1:w-2]+grey[0:h-3,1:w-2]+grey[2:h-1,0:w-3]+grey[1:h-2,0:w-3]+grey[0:h-3,0:w-3])/9
```

```python
class PointSelector:
    def __init__(self, image):
        self.firstDone = False
        self.secondDone = False
        self.fig, self.ax = plt.subplots()
        self.points = []
        self.points2 = []
        self.point_plots = []
        self.point_plots2 = []

        self.endpoints = []
        self.point_plots0 = []

        self.cid = self.fig.canvas.mpl_connect('button_press_event', self.onclick)
        self.cid_key = self.fig.canvas.mpl_connect('key_press_event', self.onkeypress)
        self.ax.imshow(image, cmap='gray')  # Display the input image
        self.image = image
        plt.title('Click to select points. Press "Enter" to finish or "m" to remove last point.')

    def onclick(self, event):
        if event.button == 1 and event.xdata and event.ydata:  # Left mouse button
            if self.secondDone:
                self.points2.append((event.xdata, event.ydata))
                self.point_plots2.append(self.ax.plot(event.xdata, event.ydata, 'bo')[0])  # Plot the selected point
                self.fig.canvas.draw()
            elif self.firstDone:
                self.points.append((event.xdata, event.ydata))
                self.point_plots.append(self.ax.plot(event.xdata, event.ydata, 'ro')[0])  # Plot the selected point
                self.fig.canvas.draw()
            else:
                self.endpoints.append(event.xdata)
                self.point_plots0.append(self.ax.plot(event.xdata, event.ydata, 'go')[0])  # Plot the selected point
                self.fig.canvas.draw()
```

```
def onkeypress(self, event):
    if event.key == 'enter':
        if self.secondDone:
            self.fig.canvas.mpl_disconnect(self.cid)  # Disconnect mouse click event
            self.fig.canvas.mpl_disconnect(self.cid_key)  # Disconnect key press event
            #plt.close()
            #Print selected points
            print("Top:")
            for point in self.points:
                print(point)
            print("Bot:")
            for point in self.points2:
                print(point)
            self.curveFit()
        elif self.firstDone:
            self.secondDone = True
        else:
            self.firstDone = True
    elif event.key == 'm':
        if self.secondDone and self.points2:
            last_point_plot = self.point_plots2.pop()
            last_point_plot.remove()  # Remove the plot of the last selected point

            self.points2.pop()  # Remove the last selected point from the list
            self.fig.canvas.draw()
        elif self.firstDone and self.points and not self.secondDone:
            last_point_plot = self.point_plots.pop()
            last_point_plot.remove()  # Remove the plot of the last selected point

            self.points.pop()  # Remove the last selected point from the list
            self.fig.canvas.draw()
        elif self.endpoints and not self.firstDone:
            last_point_plot = self.point_plots0.pop()
            last_point_plot.remove()  # Remove the plot of the last selected point

            self.endpoints.pop()  # Remove the last selected point from the list
            self.fig.canvas.draw()
    def curveFit(self):
        #Curve fit two lines
        x = [t[0] for t in self.points]
        y = [t[1] for t in self.points]
        x2 = [t[0] for t in self.points2]
        y2 = [t[1] for t in self.points2]

        popt, pcov = np.polyfit(x,y,deg=2,cov=True)
        popt2, pcov2 = np.polyfit(x2,y2,deg=2,cov=True)
        px = np.arange(min(x),max(x), 10)
        quad = np.poly1d(popt)
        py = quad(px)
        px2 = np.arange(min(x2),max(x2), 10)
        quad2 = np.poly1d(popt2)
        py2 = quad2(px2)


        self.point_plots.append(self.ax.plot(px, py, 'r-')[0])
        self.point_plots2.append(self.ax.plot(px2, py2, 'b-')[0])
        self.fig.canvas.draw()
        #Isolate values between the curves
        apointx = []
        apointy = []

        for i in range(int(min(self.endpoints)), int(max(self.endpoints))):
            for j in range(int(quad(i)),int(quad2(i))):
                #print(str(i)+', '+str(j))
                if np.abs(self.image[j][i]) > 0:
                    apointx.append(i)
                    apointy.append(j)
        print('Apointx')
        print(apointx)
        print('Apointy')
        print(apointy)
        print(self.endpoints)
        """
        def func(x, a, b, c, d):
            return -a/c*np.sqrt(c**2-((x-b))**2)+d
        poptf, pcovf = op.curve_fit(func, apointx, apointy, p0 = ((max(apointy)-min(apointy)),((max
        self.error = np.sqrt(np.diag(pcovf))
        pxf = np.arange(min(self.endpoints), max(self.endpoints),3)
        a,b,c,d = poptf
        pyf = func(pxf, a,b,c,d)
        self.point_plots0.append(self.ax.plot(pxf,pyf,'y-'))
        self.fig.canvas.draw()
        print(str(a)+', '+str(b)+', '+str(c)+', '+str(d))
        """


# Load an example image
image = dx_image  # Replace 'example_image.jpg' with your image path

# Create a PointSelector object with the image
point_selector = PointSelector(image)
```

Appendix 2: Python Code for Curve Fitting

```
import numpy as np
import scipy.optimize as op
import matplotlib.pyplot as plt
#path = 'SV.JPEG'
#path = 'IMG_0303.CR2'
path = 'Side2.webp'
#plt.close('all')
plt.figure()
import PIL.Image as image
im =image.open(path)
drop=np.array(im) #Converts data into array
plt.imshow(drop)

#Use the points outputted by the curve selection program
apointx = [192, 199, 195, 175, 176, 176, 176, 176, 176, 176, 176, 177, 178, 179, 179, 180, 181, 181, 182, 182, 182, 183, 183, 184, 185, 186, 18
apointy = [236, 236, 236, 252, 246, 247, 248, 249, 250, 251, 252, 253, 246, 245, 244, 245, 243, 242, 243, 241, 242, 243, 240, 241, 240, 239, 238, 2


XY = list(zip(apointx, apointy))
endpoints = [174.96580789907276, 215.06403770408016]

XY = [point for point in XY if point[1] < 248]
#XY = [point for point in XY if point[1] > 110]
#XY = [point for point in XY if point[1] < -(point[0]-320)/12+110]
res = list(zip(*XY))
apointx = list(res[0])
apointy = list(res[1])

plt.plot(apointx,apointy,'ro')

def func(x, b, c, d):
    return -np.sqrt(c**2-((x-b))**2)+d
"""
pxf = np.arange(min(endpoints), max(endpoints),1)
pyf = func(pxf, 195,  18.42933005, 250)

plt.plot(pxf,pyf,'b-')
"""

poptf, pcovf = op.curve_fit(func, apointx, apointy, p0 = (195,21,258), sigma = None, absolute_sigma = False, method ='lm', nan_policy = 'omit')
pxf = np.arange(min(endpoints), max(endpoints),1)
error = np.sqrt(np.diag(pcovf))
b,c,d = poptf
pyf = func(pxf, b,c,d)

plt.plot(pxf,pyf,'b-')
RadiusDiameterRatio = c/(max(endpoints)-min(endpoints))
ErrorDiameterRatio = error[1]/(max(endpoints)-min(endpoints))
print(str(RadiusDiameterRatio)+'+-'+str(ErrorDiameterRatio))
plt.xlabel('x (px)')
plt.ylabel('y (px)')
#plt.text(b,func(min(endpoints),a,b,c,d), '$\\frac{x-%s}{%s}^2+\\frac{y-%s}{%s}^2=1$'%(str(b),str(c),str(d)),fontsize=8)
```

Appendix 3: Python Code for Raytracing

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.optimize as op
M = 1
a=21.98186321
b=195.02454635
c=21.98186321
d=258.73743641
a=a*M
b=b*M
c=c*M
d=d*M


Endpoints = np.array([174.96580789907276, 215.06403770408016])*M

pDiameter = max(Endpoints)-min(Endpoints)
if Endpoints[0] < b-c:
    Endpoints[0] = b-c
if Endpoints[1] > b+c:
    Endpoints[1] = b+c
def ellipse(x, a, b, c, d):
    return -a/c*np.sqrt(c**2-((x-b))**2)+d

def dellipse(x,a,b,c,d):
    y=ellipse(x,a,b,c,d)
    return (-a**2)/(c**2)*(x-b)/(y-d)

def line(x,m,p,q):
    return m*(x-p)+q
```

```
def refract(iM, sM, n1, n2):
    snellRatio = n1/n2
    if iM < sM:
        rayA = [-1, -iM] #Vector representation of incident ray. Make sure slope is positive, if negative, manipulate -1
    else:
        rayA = [1, iM]
    rayB = [-sM, 1] #Vector representation of line perpendicular to slope
    sinth = -np.cross(rayA,rayB)/(np.linalg.norm(rayA)*np.linalg.norm(rayB)) #sin of angle between these two lines
    if np.abs(sinth*snellRatio) > 1:
        #print('Total Internal Reflection')
        return 0
    reverseRayB = np.array(rayB)*-1 #Reversed rayB
    theta = np.arcsin(snellRatio*sinth)
    R = np.array([[np.cos(theta),-np.sin(theta)],[np.sin(theta),np.cos(theta)]])
    rayC = R @ reverseRayB
    rM = rayC[1]/rayC[0]
    return rM
tIR = []
def irefract(sM, n1, n2, x):
    snellRatio = n1/n2
    rayA = [0,1]
    rayB = [-sM, 1] #Vector representation of line perpendicular to slope
    sinth = -np.cross(rayA,rayB)/(np.linalg.norm(rayA)*np.linalg.norm(rayB)) #sin of angle between these two lines
    if np.abs(sinth*snellRatio) > 1:
        #print('Total Internal Reflection at '+str(x))
        tIR.append(x)
        return 0
    reverseRayB = np.array(rayB)*-1 #Reversed rayB
    theta = np.arcsin(snellRatio*sinth)
    R = np.array([[np.cos(theta),-np.sin(theta)],[np.sin(theta),np.cos(theta)]])
    rayC = R @ reverseRayB
    rM = rayC[1]/rayC[0]
    return rM

h = ellipse(Endpoints[0],a,b,c,d)
t = 64*M #Thickness of glass
my = (d+h)

fig, ax = plt.subplots()
plt.xlabel('x (px)')
plt.ylabel('y (px)')
xj = b
#plt.yscale('log')
#plt.plot(xj,my,'bo')
ax.plot(Endpoints,[h,h],'y-')
ax.plot(Endpoints,[h+t,h+t],'y-')

xf = np.arange(int(min(Endpoints)+1),int(max(Endpoints)),0.005*M)

ax.plot(xf,ellipse(xf,a,b,c,d),'y-')

xf = np.arange(0, b-Endpoints[0], 0.1)

totalInternalReflection = []
focalLengths = []
```

```
"""
x1 =211.5-2
x2 =211.5+2
y1 =244.25-2
y2 =244.25+2

abracadabra = ax.inset_axes([0.68, 0.6, 0.3, 0.3], xlim=(x1, x2), ylim=(y1, y2))

xa = 195 -7
xb = 195 +7
ya = 218 -8
yb = 218 +8
alakazam = ax.inset_axes([0.02, 0.4, 0.3, 0.3], xlim=(xa, xb), ylim=(ya, yb))
"""
for x in xf:
    #Right Side
    xo = x+b
    y = ellipse(xo, a,b,c,d)
    xp = xo
    yp = y
    ax.plot([xp,xo],[my,y], 'b-')
    #abracadabra.plot([xp,xo],[my,y], 'b-')
    #alakazam.plot([xp,xo],[my,y], 'b-')
    n1 = 1.33
    n2 = 1
    if xo < max(Endpoints) and xo > min(Endpoints):
        m3 = dellipse(xo,a,b,c,d)
        #print(m3)
        m = irefract(m3, n1, n2, xo)
        #print(m)
        #print('-')
        if m != 0:
            ax.plot([xp,(h-d-yp)/m+xp],[yp,h-d])
            #abracadabra.plot([xp,(h-d-yp)/m+xp],[yp,h-d])
            #alakazam.plot([xp,(h-d-yp)/m+xp],[yp,h-d])
```

```
    else:
        m=0
        ax.plot([xp,xp],[yp,h-d])
        #abracadabra.plot([xp,xp],[yp,h-d])
        #alakazam.plot([xp,xp],[yp,h-d])

    m1 = m
    x1 = xp
    y1 = yp

    #Left Side
    xo = b-x
    y = ellipse(xo, a,b,c,d)
    xp = xo
    yp = y
    ax.plot([xp,xo],[my,y], 'b-')
    #abracadabra.plot([xp,xo],[my,y], 'b-')
    #alakazam.plot([xp,xo],[my,y], 'b-')
    n1 = 1.33
    n2 = 1
    if xo < max(Endpoints) and xo > min(Endpoints):
        m3 = dellipse(xo,a,b,c,d)
        #print(m3)
        m = irefract(m3, n1, n2, xo)
        #print(m)
        #print('-')
        if m != 0:
            ax.plot([xp,(h-d-yp)/m+xp],[yp,h-d])
            #abracadabra.plot([xp,(h-d-yp)/m+xp],[yp,h-d])
            #alakazam.plot([xp,(h-d-yp)/m+xp],[yp,h-d])
    else:
        m=0
        ax.plot([xp,xp],[yp,h-d])
        #abracadabra.plot([xp,xp],[yp,h-d])
        #alakazam.plot([xp,xp],[yp,h-d])
```

```
    m2 = m
    x2 = xp
    y2 = yp

    #Find Intersect

    #y-yo = m(x-xo)
    #(y-yo)/m + xo = x
    #(y - y1)/m1 + x1 = (y - y2)/m2 + x2
    #m2*y - m2*y1 + m1*m2*x1 = m1*y - m1*y2 + m1*m2*x2
    if m2 == 0 or m1 == 0 or x1 < Endpoints[0] or x1 > Endpoints[1] or x2 < Endpoints[0] or x2 > Endpoints[1]:
        continue
    else:
        y = (m2*y1-m1*m2*x1-m1*y2+m1*m2*x2)/(m2-m1)
    focalLengths.append(y)
#ax.set_ylim(1800, 2200)
focalLengths = focalLengths[1:]
focalLengths = focalLengths - h
focalLengths = focalLengths/(pDiameter)
print(min(focalLengths))
print(max(focalLengths))
tIR = tIR[1:]
print((max(tIR)-min(tIR))/pDiameter)
ax.set_xlim(96.725701978632106, 294.7747946786321)
ax.set_ylim(100,300)
#ax.indicate_inset_zoom(abracadabra, edgecolor="black")
#ax.indicate_inset_zoom(alakazam, edgecolor="black")
```

Appendix 4: Python Code for Graphing Magnification of Pixels vs Position

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle, Circle
import scipy.optimize as op
plt.close('all')
data = np.genfromtxt('Result1.csv', delimiter = ',')

data = data[1:]

num, area, x, y, bx, by, width, height = data.T
#1920 px * 1080 px
#29.4 cm * 16.6 cm
#Pixel Dimension
#0.015cm * 0.015cm
#In x direction, pixel should be 3 units
#In y direction, pixel should be 1 unit
#Mx = width/3
#My = height

Mx = width/3
My = height/1
xo = 87.3
yo = 87.5
r = 63.6
#Clears out non droplet data
for i in num:
    if (x[np.where(num == i)]-xo)**2+(y[np.where(num == i)]-yo)**2 > r**2:
        area = np.delete(area, np.where(num == i))
        x = np.delete(x, np.where(num == i))
        y = np.delete(y, np.where(num == i))
        bx = np.delete(bx, np.where(num == i))
        by = np.delete(by, np.where(num == i))
        width = np.delete(width, np.where(num == i))
        height = np.delete(height, np.where(num == i))
        Mx = np.delete(Mx, np.where(num==i))
        My = np.delete(My, np.where(num==i))
        num = np.delete(num, np.where(num == i))


#Displays the boxes made
fig, ax = plt.subplots()

im = plt.imread('Demo.TIF')
implot = plt.imshow(im)
plt.plot(x,y,'bo')
for j in num:
    i = np.where(num==j)
    #plt.plot([bx[i]-width/2, bx[i]-width/2,bx[i]+width/2, bx[i]+width/2],[by[i]-height/2, by[i]+height/2, by[i]-height/2, by[i]+height/2],'r-')
    rect = Rectangle((bx[i],by[i]),width[i],height[i],linewidth=1,edgecolor='r',facecolor='r')
    ax.add_patch(rect)
"""
circ = Circle((xo,yo),r, color = 'b')
ax.add_patch(circ)
"""
ang = np.arange(0,2*np.pi, 0.1)
plt.plot(xo+r*np.cos(ang), yo+r*np.sin(ang),'g-')
plt.xlabel( 'x (px)')
plt.ylabel('y (px)')
plt.show()


plt.figure()
plt.title('Vertical vs Horizontal Magnification')
plt.plot(Mx, My, 'ro')
plt.xlabel('Horizontal Magnification')
plt.ylabel('Vertical Magnification')

"""
plt.figure()
plt.title('Horizontal and Vertical Magnification vs Position')

plt.xlabel('Position (px)')
plt.ylabel('Magnification')

plt.plot(y, My, 'bo',)

plt.plot(x,Mx, 'ro')

t = np.arange(5, 170)

popt, pcov = np.polyfit(y,My,deg=2,cov=True)
err = np.sqrt(np.diag(pcov))

quad = np.poly1d(popt)
a,b,c = np.round(popt,4)
s = 'My: '+str(a)+'$x^2$+'+str(b)+'$x+$'+str(c)
plt.plot(t,quad(t),'b-', label = s)

popt, pcov = np.polyfit(x,Mx,deg=2,cov=True)
err = np.sqrt(np.diag(pcov))
```

```
quad = np.poly1d(popt)
a,b,c, = np.round(popt,4)
s = 'Mx: '+str(a)+'$x^2$+'+str(b)+'$x+$'+str(c)
plt.plot(t,quad(t),'r-', label = s)
plt.legend()
"""
plt.figure()
plt.title('Magnification vs Distance from Center')
Magnification = Mx*My
xo = 86.5
yo = 88.3
Distance = np.sqrt((x-xo)**2+(y-yo)**2)
plt.plot(Distance, Magnification, 'ro')

def func(x, a, b,c,d):
    return a*x**4+b*x**3+c*x**2+d*x+1
popt,pcov = op.curve_fit(func, Distance,Magnification, p0 = (1,1,1,1), sigma = None, absolute_sigma = False, method ='lm')
#popt, pcov = np.polyfit(Distance,Magnification,deg=4,cov=True)
err = np.sqrt(np.diag(pcov))
def func2(x, b,c,d):
    return b*x**3+c*x**2+d*x+1
popt2,pcov2 = op.curve_fit(func2, Distance,Magnification, p0 = (1,1,1), sigma = None, absolute_sigma = False, method ='lm')
err2 = np.sqrt(np.diag(pcov2))
#quad = np.poly1d(popt)
#a,b,c,d,e = np.round(popt,4)
a,b,c,d = np.round(popt,4)
a = np.round(popt[0], 9)
b2, c2, d2 = np.round(popt2, 4)
#s = 'Mx: '+str(a)+'$x^4$+'+str(b)+'$x^3+$'+str(c)+'$x^2+$'+str(d)+'$x+$'+str(e)
s = 'Mx: '+str(a)+'$x^4$+'+str(b)+'$x^3+$'+str(c)+'$x^2+$'+str(d)+'$x$+1'
s2 = 'Mx: '+str(b2)+'$x^3+$'+str(c2)+'$x^2+$'+str(d2)+'$x$+1'
t = np.arange(0, r, 0.1)
plt.plot(t,func(t, *popt),'b-', label = s)
plt.plot(t,func2(t, *popt2), 'g-', label = s2)

di = 0.007865
dierr = 0.000056
pxConversion = 82.5/(0.002384671479)
d = np.arange(0,70)
plt.plot(d,np.absolute((-0.008635372628+np.sqrt(di**2+0*(d/pxConversion)**2))/(0.008635372628)), 'k-', label = 'Max Predicted Magnification')
plt.plot(d,np.absolute((-0.002888187968+np.sqrt(di**2+0*(d/pxConversion)**2))/(0.002888187968)), 'y-', label = 'Min Predicted Magnification')


plt.legend()
plt.ylabel('Magnification')
plt.xlabel('Distance (px)')
```