Homework 4

```
import astropy.units as u
import astropy.constants as c
from astropy.coordinates import SkyCoord
from astropy.time import Time
import numpy as np
import matplotlib.pyplot as plt
import scipy
import pandas as pd
import glob
%matplotlib inline

from astropy.visualization import time_support
import shapely
from numpy.fft import rfft2, rfftfreq, irfft2, rfft, irfft
from shapely.geometry import LineString, Point
```

```
plt.rcParams['figure.figsize'] = (10, 10)
plt.rc('axes', labelsize=14)
plt.rc('axes', labelweight='bold')
plt.rc('axes', titlesize=16)
plt.rc('axes', titleweight='bold')
plt.rc('font', family='sans-serif')
```

Problem 1

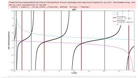
Consider a square potential well of width w, with walls of height V. Using Schrödinger's equation, it can be shown that the allowed energies E of a single quantum particle of mass m trapped in the well are solutions of [illegible] for the even numbered states, [illegible] for the odd numbered states, where the states are numbered starting from 0, with the ground state being state 0, the first excited state being state 1, and so forth. a) For an electron (mass 9.1094 × 10 − 31 kg) in a well with V = 20 eV and w = 1 nm, write a Python program to plot the three quantities [illegible], on the same graph, as a function of E from E = 0 to E = 20 eV. From your plot make approximate estimates of the energies of the first six energy levels of the particle. b) Write a second program to calculate the values of the first six energy levels in electron volts to an accuracy of 0.001 eV using binary search.

a) For an electron (mass 9.1094 × 10 − 31 kg) in a well with V = 20 eV and w = 1 nm, write a Python program to plot the three quantities [illegible], on the same graph, as a function of E from E = 0 to E = 20 eV. From your plot make approximate estimates of the energies of the first six energy levels of the particle.

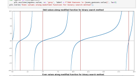
```
m = c.m_e
V = 20*u.electronvolt
w = 1*u.m*1e-9
def y1(E):
    val = ((w**2) * m * E)/(2*((c.hbar.to(u.electronvolt*u.s))**2))
    v = val.decompose()
    return np.tan(np.sqrt(v).value)
def y2(E):
    return np.sqrt((V-E)/(E))
def y3(E):
    return -1*np.sqrt((E)/(V-E))
```

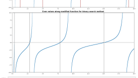
```
plt.rcParams['figure.figsize'] = (20, 10)
Es = np.arange(0,20,0.001)*u.electronvolt
plt.scatter(Es, y1(Es), label = 'y1', c= 'b', marker = 'o', s = 5)
plt.plot(Es, y2(Es), label = 'y2', c = 'r')
plt.plot(Es, y3(Es), label = 'y3', c = 'm')
plt.ylim(-10,10)
plt.xlim(0,20)
y1s = y1(Es)
y2s = y2(Es)
y3s = y3(Es)
#Getting a rough guess

even_guesses = [0.35, 2.85, 7.85, 15.07]*u.electronvolt
odd_guesses = [1.27, 5.0, 11.2, 19.1]*u.electronvolt
#idx = np.argwhere(np.diff(np.sign(y1s - y2s))).flatten()
#plt.plot(Es[idx], y1s[idx], 'bo', label = 'Even Intersections', markersize = 10)
#evens = Es[idx]
#idx = np.argwhere(np.diff(np.sign(y1s - y3s))).flatten()
#plt.plot(Es[idx][0::2], y3s[idx][0::2], 'ro', label = 'Odd Intersections', markersize = 10)
#odds = Es[idx][0::2]
for eguess in even_guesses:
    plt.axvline(eguess.value, c= 'brown', label = f'Even Guesses \n {odd_guesses.value}', lw = 5)
for eguess in odd_guesses:
    plt.axvline(eguess.value, c= 'grey', label = f'Odd Guesses \n {even_guesses.value}', lw=5)
plt.title('y(E)')
plt.xlabel('E [eV]')
plt.ylabel('y(E) [Dimensionless]')
handles, labels = plt.gca().get_legend_handles_labels()
by_label = dict(zip(labels, handles))
```

[illegible]

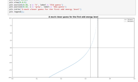


[illegible]

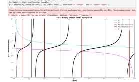




[illegible]



[illegible]



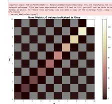
[illegible]

Problem 2

[illegible]

Plotting H

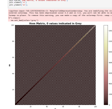
[illegible]



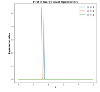
[illegible]

Finding Eigenvalues

[illegible]



[illegible]



Problem 3

Consider the equation $x = 1 - e^{-cx}$, where c is a known parameter and x is unknown. This equation arises in a variety of situations, including the physics of contact processes, mathematical models of epidemics, and the theory of random graphs. a) Write a program to solve this equation for x using the relaxation method for the case $c = 2$. Calculate your solution to an accuracy of at least 10^{-6}. b) Modify your program to calculate the solution for values of c from 0 to 3 in steps of 0.01 and make a plot of x as a function of c. You should see a clear transition from a regime in which $x = 0$ to a regime of nonzero x. This is another example of a phase transition. 3 In physics this transition is known as the percolation transition; in epidemiology it is the epidemic threshold. The ordinary relaxation method involves iterating the equation $x' = f(x)$, starting from an initial guess, until it converges. As we have seen, this is often a fast and easy way to find solutions to nonlinear equations. However, it is possible in some cases to make the method work even faster using the technique of overrelaxation. Suppose our initial guess at the solution of a particular equation is, say, $x = 1$, and the final, true solution is $x = 5$. After the first step of the iterative process, we might then see a value of, say, $x = 3$. In the overrelaxation method, we observe this value and note that x is increasing, then we deliberately overshoot the calculated value, in the hope that this will get us closer to the final solution—in this case we might pass over $x = 3$ and go straight to a value of $x = 4$ perhaps, which is closer to the final solution of $x = 5$ and hence should get us to that solution quicker. The overrelaxation method provides a formula for performing this kind of overshooting in a controlled fashion and often, though not always, it does get us to our solution faster. In detail, it works as follows. We can rewrite the equation $x' = f(x)$ in the form $x' = x + \Delta x$, where $\Delta x = x' - x = f(x) - x$. The overrelaxation method involves iteration of the modified equation $x' = x + (1 + \omega)\Delta x$, (keeping the definition of Δx the same). If the parameter ω is zero, then this is the same as the ordinary relaxation method, but for $\omega > 0$ the method takes the amount Δx by which the value of x would have been changed and changes by a little more. Using $\Delta x = f(x) - x$, we can also write x' as $x' = x + (1 + \omega)[f(x) - x] = (1 + \omega)f(x) - \omega x$, which is the form in which it is usually written. For the method to work the value of ω must be chosen correctly, although there is some wiggle room—there is an optimal value, but other values close to it will typically also give good results. Unfortunately, there is no general theory that tells us what the optimal value is. Usually it is found by trial and error). For the overrelaxation method, it can be shown that the error on x' is given by $\epsilon' = x - x' \, / \, 1 - 1/[(1 + \omega)f'(x) - \omega]$. c) Consider again the equation $x = 1 - e^{-cx}$. Take the program you wrote for part (a), which solved the equation for the case $c = 2$, and modify it to print out the number of iterations it takes to converge to a solution accurate to 10^{-6}. d) Now write a new program (or modify the previous one) to solve the same equation $x = 1 - e^{-cx}$ for $c = 2$, again to an accuracy of 10^{-6}, but this time using overrelaxation. Have your program print out the answers it finds along with the number of iterations it took to find them. Experiment with different values of ω to see how fast you can get the method to converge. A value of $\omega = 0.5$ is a reasonable starting point. With some trial and error you should be able to get the calculation to converge about twice as fast as the simple relaxation method, i.e., in about half as many iterations. 4 e) Are there any circumstances under which using a value $\omega < 0$ would help us find a solution faster than we can with the ordinary relaxation method? (Hint: The answer is yes, but why?)

a) Write a program to solve this equation for x using the relaxation method for the case $c = 2$. Calculate your solution to an accuracy of at least 10^{-6}.

In [26]:
```
def eqn(x,C):
    return 1-np.exp(-1*C*x)
```

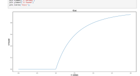
In [27]:
```
#This is just an example from the book: mostly following its format
x = 0.8
accur = 1e-6
k = 0
error = 1
while error >accur:
    xp = x
    x=eqn(x,2)
    k+=1
    error = abs(x-xp)
    print(x, f"error = {error}")
```

```
0.7981634826051646 error = 0.0018905179060554
0.7973362270653534 error = 0.0008767256491818262
0.7970249887566617 error = 0.0003112382676723079
0.7968988162987747 error = 0.00012638246768765538
0.7968472720060883 error = 5.134336265461461e-05
0.7968264387344767 error = 2.086217168457426e-05
0.7968179332738492 error = 8.477461426516843e-06
0.7968144883875782 error = 3.444865471866638e-06
0.7968130883686114 error = 1.3999389667587245e-06
0.7968125184692648 error = 5.688993466312966e-07
```

b) Modify your program to calculate the solution for values of c from 0 to 3 in steps of 0.01 and make a plot of x as a function of c. You should see a clear transition from a regime in which $x = 0$ to a regime of nonzero x. This is another example of a phase transition. 3 In physics this transition is known as the percolation transition; in epidemiology it is the epidemic threshold. The ordinary relaxation method involves iterating the equation $x' = f(x)$, starting from an initial guess, until it converges. As we have seen, this is often a fast and easy way to find solutions to nonlinear equations. However, it is possible in some cases to make the method work even faster using the technique of overrelaxation. Suppose our initial guess at the solution of a particular equation is, say, $x = 1$, and the final, true solution is $x = 5$. After the first step of the iterative process, we might then see a value of, say, $x = 3$. In the overrelaxation method, we observe this value and note that x is increasing, then we deliberately overshoot the calculated value, in the hope that this will get us closer to the final solution—in this case we might pass over $x = 3$ and go straight to a value of $x = 4$ perhaps, which is closer to the final solution of $x = 5$ and hence should get us to that solution quicker. The overrelaxation method provides a formula for performing this kind of overshooting in a controlled fashion and often, though not always, it does get us to our solution faster. In detail, it works as follows. We can rewrite the equation $x' = f(x)$ in the form $x' = x + \Delta x$, where $\Delta x = x' - x = f(x) - x$. The overrelaxation method involves iteration of the modified equation $x' = x + (1 + \omega)\Delta x$, (keeping the definition of Δx the same). If the parameter ω is zero, then this is the same as the ordinary relaxation method, but for $\omega > 0$ the method takes the amount Δx by which the value of x would have been changed and changes by a little more. Using $\Delta x = f(x) - x$, we can also write x' as $x' = x + (1 + \omega)[f(x) - x] = (1 + \omega)f(x) - \omega x$, which is the form in which it is usually written. For the method to work the value of ω must be chosen correctly, although there is some wiggle room—there is an optimal value, but other values close to it will typically also give good results. Unfortunately, there is no general theory that tells us what the optimal value is. Usually it is found by trial and error). For the overrelaxation method, it can be shown that the error on x' is given by $\epsilon' = x - x' \, / \, 1 - 1/[(1 + \omega)f'(x) - \omega]$.

In [28]:
```
def find_x(C):
    x = 0.8
    accur = 1e-6
    k = 0
    error = 1
    while error >accur:
        xp = x
        x=eqn(x,C)
        k+=1
        error = abs(x-xp)
    return(x)
```

In [29]:
```
cs = np.arange(0,3.01,.01)
xvals = []
for C in cs:
    xvals.append(find_x(C))
plt.plot(cs, xvals)
plt.xlabel('C values')
plt.ylabel('X found')
plt.title('C(x)');
```



c) Consider again the equation $x = 1 - e^{-cx}$. Take the program you wrote for part (a), which solved the equation for the case $c = 2$, and modify it to print out the number of iterations it takes to converge to a solution accurate to 10^{-6}.

In [30]:
```
def find_x(C):
    x = 0.8
    accur = 1e-6
    k = 0
    error = 1
    while error >accur:
        xp = x
        x=eqn(x,C)
        k+=1
        error = abs(x-xp)
    return(k)
```

In [31]:
```
print(f'It took {find_x(2)} iterations to converge to a solution.')
```

```
It took 10 iterations to converge to a solution.
```

d) Now write a new program (or modify the previous one) to solve the same equation $x = 1 - e^{-cx}$ for $c = 2$, again to an accuracy of 10^{-6}, but this time using overrelaxation. Have your program print out the answers it finds along with the number of iterations it took to find them. Experiment with different values of ω to see how fast you can get the method to converge. A value of $\omega = 0.5$ is a reasonable starting point. With some trial and error you should be able to get the calculation to converge about twice as fast as the simple relaxation method, i.e., in about half as many iterations. 4

In [32]:
```
def find_x(C, w):
    x = 0.8
    accur = 1e-6
    k = 0
    error = 1
    while error >accur:
        xp = x
        delta = eqn(x,C)-x
        x+=(1+w)*delta
        k+=1
        error = abs(x-xp)
        print( f'{k} iterations:', x)
```

In [33]:
```
find_x(2,1.7)
```

```
1 iterations: 0.7967709366666659
2 iterations: 0.7968124088483369
3 iterations: 0.7968121288891462
```

For an ω value of 1.7, it only takes 3 iterations for the code to converge. This is a little over 3 times faster than the simpler relaxation method.

e) Are there any circumstances under which using a value $\omega < 0$ would help us find a solution faster than we can with the ordinary relaxation method? (Hint: The answer is yes, but why?)

In [51]:
```
find_x(2,-1)
```

```
[illegible]
```

```
<ipython-input-32-d222d626813f>:9: RuntimeWarning: overflow encountered in double_scalars
  x=(1+w)*delta
<ipython-input-32-d222d626813f>:11: RuntimeWarning: invalid value encountered in double_scalars
  error = abs(x-xp)
```

```
[illegible]
35 iterations: inf
36 iterations: inf
```

```
<ipython-input-32-d222d626813f>:9: RuntimeWarning: overflow encountered in double_scalars
  x=(1*w)*delta
<ipython-input-32-d222d626813f>:11: RuntimeWarning: invalid value encountered in double_scalars
  error = abs(x-xp)
```

A negative value is counterintuitive but it could be helpful. We see that in both cases the intervals converge to infinity. However, for larger values, it converges to infinity faster. In the case that we were taking a root of a very large number, it may be helpful to consider using negative values for omega.

Problem 4

Problem 4: Detecting periodicity Let's analyze again the data in sunspots.txt (from Homework #2), which contains the ob- served number of sunspots on the Sun for each month since January 1749. The file contains two columns of numbers, the first representing the month and the second being the sunspot number. a) Write a program that reads the data in the file and makes a graph of sunspots as a function of time. You should see that the number of sunspots has fluctuated on a regular cycle for as long as observations have been recorded. Make an estimate of the length of the cycle in months. b) Modify your program to calculate the Fourier transform of the sunspot data and then make a graph of the magnitude squared | c k | 2 of the Fourier coefficients as a function of k—also called the power spectrum of the sunspot signal. You should see that there is a noticeable peak in the power spectrum at a nonzero value of k. The appearance of this peak tells us that there is one frequency in the Fourier series that has a higher amplitude than the others around it—meaning that there is a large sine-wave term with this frequency, which corresponds to the periodic wave you can see in the original data. c) Find the approximate value of k to which the peak corresponds. What is the period of the sine wave with this value of k? You should find that the period corresponds roughly to the length of the cycle that you estimated in part (a). This kind of Fourier analysis is a sensitive method for detecting periodicity in signals. Even in cases where it is not clear to the eye that there is a periodic component to a signal, it may still be possible to find one using a Fourier transform.

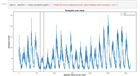
a) Write a program that reads the data in the file and makes a graph of sunspots as a function of time. You should see that the number of sunspots has fluctuated on a regular cycle for as long as observations have been recorded. Make an estimate of the length of the cycle in months.

In [308]:

```python
def read_sunspots(path):
    """
    Reads in a file containing information about sunspot data and produces a plot of the sunspots over time

    Parameters
    ----
    path: string, absolute path to the sunspot file

    Returns
    ----
    Produces a matplotlib plot of the sunspots over time. Sunspot count is the number of sunspots observed
    in a given month since 01-01-1749.

    """
    #Define an empty list to store the dates
    plt.rcParams['figure.figsize'] = (20, 10)
    dates = []
    #Read in the file as a pandas DataFrame, indicate that the file contains no headers
    df = pd.read_table(path, header = None)
    #Months are indicated in the first column and spot counts are indicated in the second
    months = df[0].values
    spots = np.array(df[1].values)
    plt.plot(months, spots, label = 'Data')
    plt.axvline(600, c='k', label = 'Arbitrary peak bound')
    plt.axvline(700, c ='k')
    plt.title('Sunspots over time')
    plt.xlabel('Months since 01-01-1749')
    plt.ylabel('Sunspot Count')
    plt.legend();
    return spots, months
```

In [300]:

```python
spots, months = read_sunspots(path = '/home/kelsey/computational_phys/Homework2/sunspots.txt')
```



In [42]:

```python
guess = 700-600
print(f'I will make an arbitrary guess that the cycle repeats every {guess} months.')
```

```
I will make an arbitrary guess that the cycle repeats every 100 months.
```

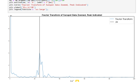
b) Modify your program to calculate the Fourier transform of the sunspot data and then make a graph of the magnitude squared | c k | 2 of the Fourier coefficients as a function of k—also called the power spectrum of the sunspot signal. You should see that there is a noticeable peak in the power spectrum at a nonzero value of k. The appearance of this peak tells us that there is one frequency in the Fourier series that has a higher amplitude than the others around it—meaning that there is a large sine-wave term with this frequency, which corresponds to the periodic wave you can see in the original data.

In [313]:

```python
plt.rcParams['figure.figsize'] = (20, 10)
plt.plot(np.abs(rfft(spots))**2)
plt.title('Fourier Transform of Sunspot Data')
plt.xlabel('$|c_k|^2$');
```

In [311]:

```python
plt.rcParams['figure.figsize'] = (20, 10)
plt.plot(np.abs(rfft(spots))**2, label = 'Fourier Transform')
plt.xlim(0,100)
plt.ylim(0,.5*1e10)
mx = np.argmax((np.abs(rfft(spots))**2)[10:])+10
plt.axvline(mx, c= 'k', label = f'{mx}')
plt.title('Fourier Transform of Sunspot Data Zoomed, Peak Indicated')
plt.xlabel('$|c_k|^2$')
plt.legend(fontsize = 'xx-large');
```



c) Find the approximate value of k to which the peak corresponds. What is the period of the sine wave with this value of k? You should find that the period corresponds roughly to the length of the cycle that you estimated in part (a). This kind of Fourier analysis is a sensitive method for detecting periodicity in signals. Even in cases where it is not clear to the eye that there is a periodic component to a signal, it may still be possible to find one using a Fourier transform.

In [317]:

```python
iffs = irfft(np.abs(rfft(spots)))
print(f'We notice a period every {iffs[24]} months.')
#plt.plot(irfft(np.abs(rfft(spots))));
```

```
We notice a period every 141.56882416613986 months.
```

Problem 5

n the web site you will find a file called blur.txt that contains a grid of values repre- senting brightness on a black-and-white photo—a badly out-of-focus one that has been deliberately blurred using a Gaussian point spread function of width σ = 25. Write a program that reads the grid of values into a two-dimensional array of real numbers and then draws the values on the screen of the computer as a density plot. You should see the photo appear. If you get something wrong it might be upside-down. Work with the details of your program until you get it appearing correctly. (Hint: The picture has the sky, which is bright, at the top and the ground, which is dark, at the bottom.) ?

In [33]:

```python
data = np.loadtxt('/home/kelsey/Downloads/blur.txt')
plt.imshow(data, cmap = 'gray')
plt.axis('off')
plt.title('Blurred Image');
```

b) Write another program that creates an array, of the same size as the photo, containing a grid of samples drawn from the Gaussian f (x, y) above with σ = 25. Make a density plot of these values on the screen too, so that you get a visualization of your point spread function. Remember that the point spread function is periodic (along both axes), which means that the values for negative x and y are repeated at the end of the interval. Since the Gaussian is centered on the origin, this means there should be bright patches in each of the four corners of your picture, something like this:

In [53]:

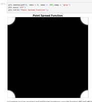
```python
def gaussian(sigma, x_0, y_0, x, y):
    #Calculating the values for a gaussian beam centered at x,y at some arbitrary pixel x_0,y_0 some distance a
    #from the gaussian center
    xs = (x-x_0)**2
    ys = (y-y_0)**2
    gauss_val = np.e** ((-(xs + ys))/(2*(sigma**2)))
    return gauss_val
```

In [67]:

```python
sigma = 25
Nx = data.shape[0]
Ny = data.shape[1]
xspan = np.linspace(-Nx/2,Nx/2,Nx)
yspan = np.linspace(-Ny/2,Ny/2,Ny)
psf = np.zeros([Nx,Ny])

xcorners = [-Nx/2, Nx/2, -Nx/2, Nx/2]
ycorners = [-Ny/2, Ny/2, Ny/2, -Ny/2]
for corner in range(len(xcorners)):
    x_vals = xcorners[corner]
    y_vals = ycorners[corner]
    for xind, xval in enumerate(xspan):
        for yind, yval in enumerate(yspan):
            psf[xind, yind] += gaussian(sigma, x_vals, y_vals, xval, yval)
```

In [67]:

```python
plt.imshow(psf/4, vmin = 0, vmax = .001,cmap = 'gray')
plt.axis('off')
plt.title('Point Spread Function');
```



c) Combine your two programs and add Fourier transforms using the functions rfft2 and irfft2 from numpy.fft, to make a program that does the following: i) Reads in the blurred photo ii) Calculates the point spread function iii) Fourier transforms both iv) Divides one by the other v) Performs an inverse transform to get the unblurred photo vi) Displays the unblurred photo on the screen When you are done, you should be able to make out the scene in the photo, although probably it will still not be perfectly sharp. Hint: One thing you'll need to deal with is what happens when the Fourier transform of the point spread function is zero, or close to zero. In that case if you divide by it you'll get an error (because you can't divide by zero) or just a very large number (because you're dividing by something small). A workable compromise is that if a value in the Fourier transform of the point spread function is smaller than a certain amount ε you don't divide by it— just leave that coefficient alone. The value of ε is not very critical but a reasonable value seems to be 10 − 3 .

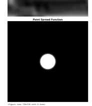
In [99]:

```python
def deconvolve(path):
    plt.rcParams['figure.figsize'] = (10, 10)
    data = np.loadtxt(path)
    plt.imshow(data, cmap = 'gray')
    plt.axis('off')
    plt.title('Blurred Image');

    sigma = 25
    Nx = data.shape[0]
    Ny = data.shape[1]
    xspan = np.linspace(-Nx/2,Nx/2,Nx)
    yspan = np.linspace(-Ny/2,Ny/2,Ny)
    psf = np.zeros([Nx,Ny])

    xcorners = [0]#, Nx/2, -Nx/2, Nx/2]
    ycorners = [0]#, Ny/2, Ny/2, -Ny/2]
    for corner in range(len(xcorners)):
        x_vals = xcorners[corner]
        y_vals = ycorners[corner]
        for xind, xval in enumerate(xspan):
            for yind, yval in enumerate(yspan):
                psf[xind, yind] += gaussian(sigma, x_vals, y_vals, xval, yval)
    psf = psf

    plt.figure()
    plt.imshow(psf, cmap = 'gray', vmin = 0, vmax = 0.001)
    plt.axis('off')
    plt.title('Point Spread Function');

    plt.figure()

    ft_data = (rfft2((data)))
    ft_ps = rfft2((psf))

    Nx = ft_data.shape[0]
    Ny = ft_data.shape[1]
    xspan = np.linspace(-Nx/2,Nx/2,Nx)
    yspan = np.linspace(-Ny/2,Ny/2,Ny)
    ft_matrix = np.zeros([Nx,Ny])

    for xi, x in enumerate(xspan):
        for yi, y in enumerate(yspan):
            eps = 1e-3
            ps = abs(ft_ps[xi,yi])
            val = ((ft_data[xi,yi])/abs(ft_ps[xi,yi]))
            if ps>eps:
                ft_matrix[xi, yi] = val
            else:
                ft_matrix[xi,yi] = ft_data[xi,yi]
    plt.figure()
    plt.imshow(np.abs(ft_matrix), cmap = 'gray',vmin = 0, vmax = 0.001)
    plt.axis('off')
    plt.title('FT(data)/FT(psf)');

    plt.figure()
    plt.imshow(((irfft2(((ft_matrix))))), cmap = 'gray', aspect='auto')
    plt.axis('off')
    plt.title('Unblurred Image')
    plt.ylim(Nx/2,0);
    return ft_matrix, psf
```

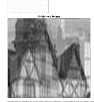
In [99]:

```python
path = '/home/kelsey/Downloads/blur.txt'
a,b = deconvolve(path)
```

```
<ipython-input-99-74c90865822f>:47: ComplexWarning: Casting complex values to real discards the imaginary part
  ft_matrix[xi, yi] = val
<ipython-input-99-74c90865822f>:49: ComplexWarning: Casting complex values to real discards the imaginary part
  ft_matrix[xi,yi] = ft_data[xi,yi]
<ipython-input-99-74c90865822f>:45: RuntimeWarning: divide by zero encountered in cdouble_scalars
  val = ((ft_data[xi,yi])/abs(ft_ps[xi,yi]))
<ipython-input-99-74c90865822f>:45: RuntimeWarning: invalid value encountered in cdouble_scalars
  val = ((ft_data[xi,yi])/abs(ft_ps[xi,yi]))
```




```
[illegible]
```



I played around with this for a long time and could not find a way to make it look any better. The best I could do was reduce the gaussian beams so that there was only one beam at the center and this seemed to do a bit better, but it does not look like what the original PSF looked like and there is obviously some strange reflecting going on. I think there needs to be some absolute value somewhere but I could not find a place to put it that didnt make the image a lot worse. I look forward to seeing the solution because I could not figure this one out and sunk too much time in to it attempting to get something to work.

d) Bearing in mind this last point about zeros in the Fourier transform, what is it that limits our ability to deblur a photo? Why can we not perfectly unblur any photo and make it completely sharp? We have seen this process in action here for a normal snapshot, but it is also used in many physics applications where one takes photos. For instance, it is used in astronomy to enhance photos taken by telescopes. It was famously used with images from the Hubble Space Tele- scope after it was realized that the telescope's main mirror had a serious manufacturing flaw and was returning blurry photos—scientists managed to partially correct the blurring using Fourier transform tech

The limits on this lie in the acuracy of the point spread function. The more acurate the point spread function, the more acurate the deconvolution. As we move away from the center of the gaussian beam in the PSF, we also move towards smaller and smaller values. These values are the limiting factor on the acuracy of our image.

In [ ]: