My sample book

The Jupyter Book Community

Apr 14, 2024

CONTENTS

This is a small sample book to give you a feel for how book content is structured. It shows off a few of the major file types, as well as some sample content. It does not go in-depth into any particular topic - check out the Jupyter Book documentation for more information.

Check out the content pages bundled with this sample book to see more.

CHAPTER

ONE

INTRO AND SETUP

These are all the libraries that are needed

```
import pandas as pd
import numpy as np
import cmath
import math
from scipy.optimize import fmin, minimize
from astropy import units as u
from scipy.interpolate import RegularGridInterpolator
import matplotlib.pyplot as plt
import copy
%matplotlib inline
```

As the earth spins, we trace out a track in the Fourier domain in order to give us one snapshot of the black hole image. On earth we create a sinusoidal wave in order to amplify the data that we recieve from our snapshot. This create two different sine bands. One being hi and the other being low. We can use these two data sets to create two different images for the same day in practice. Theoretically they should create the same image

CHAPTER

TWO

PASTE IMAGES HERE

2.1 Data Details

The data set from EHT (The Event Horizon Telescope) and from the HOPS pipeline (software from MIT), comes in multiple sets. From one data collection, we are given a hi band and low band sets.

Suppose that the sky signal is some radio wave $Sky = \epsilon \sin(\omega t)$. The signal however is extremely small which make it hard to detect. In order to pick out this signal, we use a local oscillator which generates another wave (i.e. $LO = A\sin(\omega' t)$) where A is some amplitude. Once these two signals are multiplied together, we get something like so: $Sky \times LO = A\epsilon \sin(\omega t) \times \sin(\omega' t) = A\epsilon\text{sin}[(\omega + \omega')t] + sin[(\omega - \omega')t]$.

Here we have $A\epsilon$ be an amplitude such that the signals are big enough for our instruments to detect. The $\omega + \omega'$ and $\omega - \omega'$ results in two bands which the data is taken from

CHAPTER

THREE

SOME DETAILS ABOUT THE TABLE

time is the time that it was taken. This time stamp is not used because we assume things are static.

T1 and T2 are the two telescopes.

U and V are the fourier location of the fourier domain. When we plot each point in the fourier domain, we should obtain something like below #HEREEE They are given in terms of lambda.

Iamp is the Amplitude of the Fourier Coefficient

IPhase is the Phase of the Fourier Coefficient in degrees

ISigma is the error or the noisiness of the data point

Here, we will start to preprocess the data. First we will create a class to hold the information is a more accessible manner

```
class data:
    u: float
    v: float
    phase: float
    amp: float
    sigma: float
    vis_data: complex
    def __init__(self, u, v, phase, amp, sigma):
        self.u = u
        self.v = v
        self.phase = phase
        self.amp = amp
        self.sigma = sigma
        self.vis_data = amp * np.exp(1j * math.radians(phase))

    def __repr__(self):
        return f"[u: {self.u}, v: {self.v}]"

    def __str__(self):
        return f"[u: {self.u}, v: {self.v}]"
```

The next few cells read the data from a csv file

```
def process_data(data_df):
    coords = []
    data_list = []
    for i in range(len(data_df)):
        data_list.append(data(data_df.loc[i, 'U(lambda)'], data_df.loc[i, 'V(lambda)
↪'], data_df.loc[i, 'Iphase(d)'], data_df.loc[i, 'Iamp(Jy)'], data_df.loc[i,
↪'Isigma(Jy)']))
```

(continues on next page)

(continued from previous page)

```
        coords.append([data_df.loc[i, 'U(lambda)'], data_df.loc[i, 'V(lambda)']])
    coords = np.array(coords)
    return coords, data_list
```

```
def read_data(filename: str()):
    df = pd.read_csv(filename)
    return df
```

```
df = read_data("./data/SR1_M87_2017_095_hi_hops_netcal_StokesI.csv")
coords, data_list = process_data(df)
df
```

```
      #time(UTC)  T1  T2     U(lambda)     V(lambda)  Iamp(Jy)  Iphase(d)  \
0       0.768056  AA  LM  1.081710e+09 -3.833722e+09  0.014292  -118.9454
1       0.768056  AA  PV -4.399933e+09 -4.509480e+09  0.136734     5.8638
2       0.768056  AA  AP  8.349088e+05 -1.722271e+06  1.119780    58.1095
3       0.768056  AP  LM  1.080840e+09 -3.832004e+09  0.018448  -137.6802
4       0.768056  AP  PV -4.400757e+09 -4.507747e+09  0.139619   -57.1724
...          ...  ..  ..           ...           ...       ...        ...
6453    8.165278  AZ  LM -1.078324e+09  1.029597e+09  0.315983    10.9377
6454    8.165278  AZ  JC  3.392180e+09  9.968579e+08  0.058864    46.0474
6455    8.165278  JC  LM -4.470504e+09  3.273711e+07  0.108582  -178.7050
6456    8.165278  JC  SM  1.745735e+04 -1.192282e+05  1.123722   -29.5589
6457    8.165278  LM  SM  4.470522e+09 -3.285633e+07  0.104931    96.6936

      Isigma(Jy)
0       0.005847
1       0.004968
2       0.005243
3       0.044576
4       0.032591
...          ...
6453    0.030449
6454    0.090288
6455    0.043965
6456    0.091870
6457    0.028783

[6458 rows x 8 columns]
```

```
%run ./utility.ipynb
```

CHAPTER

FOUR

INTERPOLATION AND REGULARIZERS

In this notebook, we will talk about interpolating coordinate values from an image as well as calculating the regularizer for gradient descent.

4.1 Background on FOV and k-space

FOV stands for Field of view and it refers to the distance over which an image is acquired or displayed. These images are usually captured by some sort of signal processing like telescopes and MRIs. The FOV and pixel width determine the number of units in the fourier domain that must be obtained in order to reconstruct an image.

For the sakes of this research and example, FOVx = FOVy = FOV and $\Delta x = \Delta y = \Delta w$. Where w is the pixel width.

The Fourier projection of spatial frequencies all follow a similar pattern. Instead of regular sine waves, we see complex exponentials , cos n ω t + i sin n ω t. Each pair of samples lines differ by exactly 1 cycle over the FOV. This means that $\Delta k = k_{n+1} - k_n = \frac{n+1}{FOV} - \frac{n}{FOV} = \frac{1}{FOV}$

4.2 Interpolation

In this notebook, we have an image that is being linearly transformed using fourier transforms. We do this in order to allow us to use direct comparisons between our image (in real space but being transformed into fourier space) and the data obtained by the telescopes (which are in the spectral or fourier domain.)

When doing this transformation, we obtain a grid of points extracted from the image which do not necessarily line up with the U and V coordinates from the dataset. Instead, we must interpolate or estimate the values of the data points using existing known points in our grid.

Furthermore, in order to translate correctly from image space to fourier space, we need to multiply the grids (kx and ky) by k_FOV. The calculation for this is explained above.

It is important to note that the more grid points we have, the bigger the image. Thus the size of our Fourier Domain is the Number of pixels of the image by k_FOV. If our FOV is very small, then the width of the fourier grid becomes large.

Next we want to interpolate the complex value at each coordinate. RegularGridInterpolator however cannot interpolate complex numbers so we interpolate the real and imaginary parts separately and then combined them for the final result.

Finally, we use a linear method for interpolation due to it's local interpolation scheme. The other schemes that RegularGridInterpolator offers uses a C^2-smooth split which is a non-local scheme. Since we wanted to optimize the computation time, we chose the linear method.

```
# Assumption image is 80x80 pixels
# Pass in array of u,v coords then return array of interpolated values
def interpolate(image, coords, FOV):
    """
    image is a 80x80 pixel image that represents our reconstructed image
    coords is a list of u,v coordinates that we obtained from our data
    FOV is the Field of view from the telescopes. For the EHT data, our FOV is 100␣
↪micro ascs.
    """

    ft_image = np.fft.fftshift(np.fft.fft2(image))

    k_FOV = 1/FOV

    kx = np.fft.fftshift(np.fft.fftfreq(ft_image.shape[0], d = 1/(k_FOV*ft_image.
↪shape[0])))
    ky = np.fft.fftshift(np.fft.fftfreq(ft_image.shape[1], d = 1/(k_FOV*ft_image.
↪shape[1])))

    interp_real = RegularGridInterpolator((kx, ky), ft_image.real, bounds_error=False,
↪ method="linear")
    interp_imag = RegularGridInterpolator((kx, ky), ft_image.imag, bounds_error=False,
↪ method="linear")

    real = interp_real(coords)
    imag = interp_imag(coords)

    return real + imag * 1j
```

Here we do some tests using interp_real and interp_imag to verify that they do interpolate a value using a linear scheme. First we start with a very basic 5x5 grid.

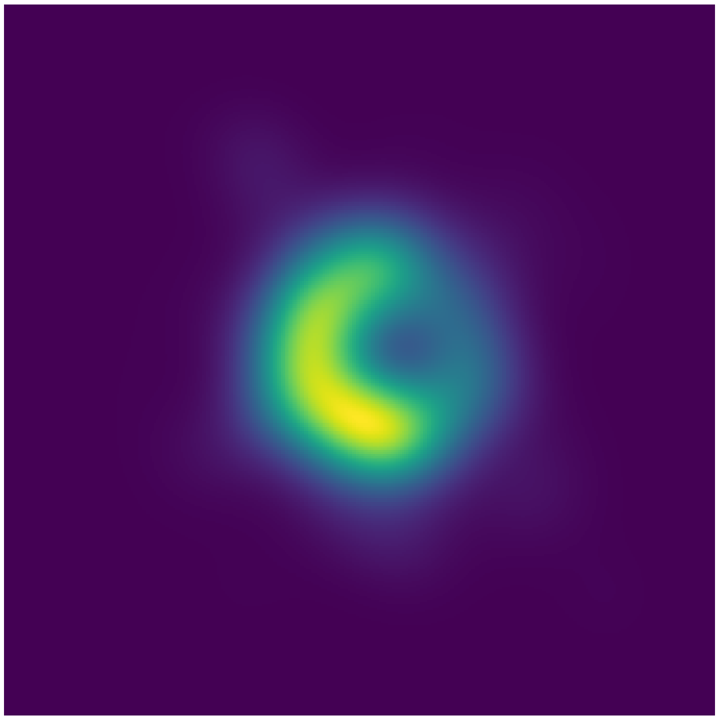
```
x = [0,1,2,3,4]
y = [0,1,2,3,4]
z = [[1,1,1,1,1],[2,2,2,2,2],[3,3,3,3,3],[4,4,4,4,4],[5,5,5,5,5]]
interp = RegularGridInterpolator((x,y),z)
print("Expected: 1, Actual:", interp([0,3]))
print("Expected: 2.5, Actual:", interp([1.5,2.5]))
```

```
Expected: 1, Actual: [1.]
Expected: 2.5, Actual: [2.5]
```

Now, here is a test on a fourier transformed image

```
sample = np.loadtxt("images/interpolate_test.csv", delimiter=",")
plt.figure()
plt.imshow(sample)
plt.axis('off')
```

```
(-0.5, 179.5, 179.5, -0.5)
```



```
ft_image = np.fft.fftshift(np.fft.fft2(sample))
interpolated_points = interpolate(sample,[[-9.0e+09,-9.0e+09], [-8.9e+09,-9.0e+09] ,[-
↪8.95e+09,-9.0e+09]],180*u.uas.to(u.rad))
print("Expected:",ft_image[0,0],"\nActual:", interpolated_points[0])
print("\nExpected:",ft_image[1,0],"\nActual:", interpolated_points[1])
print("\nExpected: ",(ft_image[1,0]+ft_image[0,0])/2,"\nActual:", interpolated_
↪points[2])
```

```
Expected: (3.0000000019953994-2.1431905139479568e-09j)
Actual: (-335133.52213257825+690198.3841637481j)

Expected: (38.20989061676637+0.6177808031820264j)
Actual: (-231640.3460875617+376131.9723567965j)

Expected:  (20.604945309380888+0.30889040051941796j)
Actual: (-283386.93411007+533165.1782602723j)
```

Below is an interpolation subroutine written by Misha Stepanov.

The code provides an insight into a local interpolation scheme using 2D cubic splines. It is local in a sense that only 12 grid points around the point at which we are interpolating is used to contribute the the estimation. The code the interpolation is done with data being wrapped into a 2D torus and the xy-coordinates are supplied assuming that the step of the grid in both the x and y directions are equal to 1

4.2.1 include stuff about cublic splines? Get permission first

```
def cubf1(x, y):
  cf1 = (1. + x - x*x)*(1. + 2.*y)*(1. - y)
  cf2 = (1. + y - y*y)*(1. + 2.*x)*(1. - x)
  return 0.5*(cf1 + cf2)*(1. - x)*(1. - y)
def cubf2(x, y):
  return -0.5*x*(1. - x)*(1. - x)*(1. + 2.*y)*(1. - y)*(1. - y)
def cubic12(A, X):
  N = len(A);  F = np.zeros(len(X))
  for i in range(len(X)):
    m = np.mod(X[i], N);  x, y = m - np.floor(m)
    m1, m2 = np.floor(np.mod(m + np.array([2., 2.]), N)).astype(int)
    F[i]  = A[m1 - 2, m2 - 2]*cubf1(x, y)
    F[i] += A[m1 - 1, m2 - 2]*cubf1(1. - x, y)
    F[i] += A[m1 - 2, m2 - 1]*cubf1(x, 1. - y)
    F[i] += A[m1 - 1, m2 - 1]*cubf1(1. - x, 1. - y)
    F[i] += A[m1 - 3, m2 - 2]*cubf2(x, y)
    F[i] += A[m1 - 2, m2 - 3]*cubf2(y, x)
    F[i] += A[m1    , m2 - 2]*cubf2(1. - x, y)
    F[i] += A[m1 - 1, m2 - 3]*cubf2(y, 1. - x)
    F[i] += A[m1 - 3, m2 - 1]*cubf2(x, 1. - y)
    F[i] += A[m1 - 2, m2    ]*cubf2(1. - y, x)
    F[i] += A[m1    , m2 - 1]*cubf2(1. - x, 1. - y)
    F[i] += A[m1 - 1, m2    ]*cubf2(1. - y, 1. - x)
  return F

def func(x, y):
  return np.sin(2*np.pi*x) + np.cos(2*np.pi*(x + y))
```

```
A = np.zeros((10, 10))
for i in range(10):
  x = i / 10.
  for j in range(10):
    y = j / 10.
    A[i, j] = func(x, y)

X = np.zeros((10000, 2))
for i in range(10000):
  X[i, 0] = i / 1000.
  X[i, 1] = i / 2000.
F = cubic12(A, X)
for i in range(10000-10,10000):
  print(X[i, 0], X[i, 1], F[i], func(X[i, 0] / 10., X[i, 1] / 10.))
```

```
9.99 4.995 -1.0058598083263428 -1.0062387310745087
9.991 4.9955 -1.0052755185370548 -1.0056188621460636
9.992 4.996 -1.004690839850659 -1.0049981027528014
9.993 4.9965 -1.0041057771382442 -1.0043764531360648
9.994 4.997 -1.0035203352965503 -1.0037539135379836
9.995 4.9975 -1.0029345192480468 -1.0031304842014765
9.996 4.998 -1.002348333941004 -1.002506165370251
9.997 4.9985 -1.001761784349569 -1.0018809572888083
9.998 4.999 -1.001174875473833 -1.0012548602024367
9.999 4.9995 -1.0005876123399073 -1.0006278743572115
```

4.3 Regularization

Here we start to calculate regularizers. We do so in order to smooth image and add some bias into the model to prevent it from overfitting the training data. Regularizers allow machine learning models to generalize to new examples that it has not seen during training.

The regularizer below implements a regularization method called "total squared variation." This method favors smooth edges and is often used for astronomical image reconstruction.

The formula for a TSV regularizer is

$-\sum_l \sum_m [(I_{l+1,m} - I_{l,m})^2 + (I_{l,m+1} - I_{l,m})^2]$

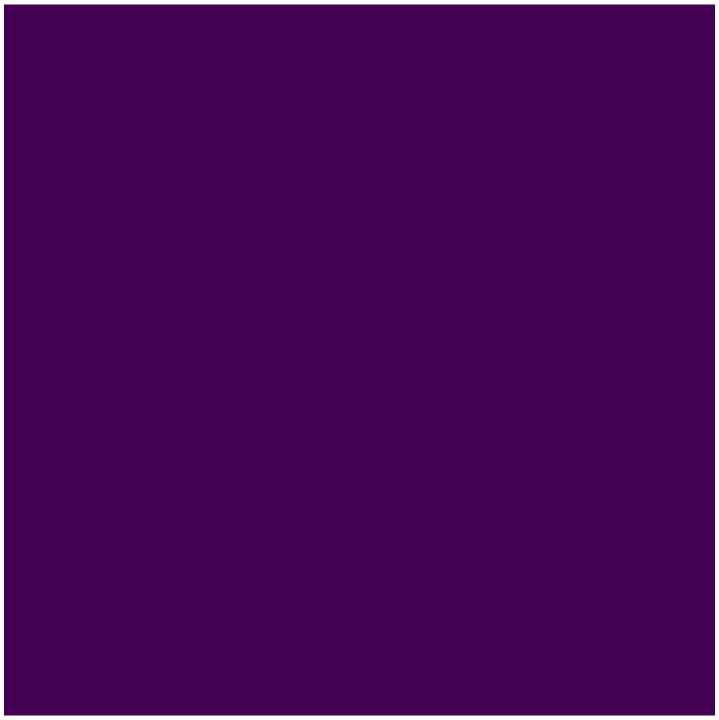
where l and m are pixel coordinates and I is the image. There are however other similar regularizers like Total variation that favors pixel-to-pixel smoothness.

```
# p is the exponent of the regularizing terms
# The smaller p the more sensitive it is to noise
# TSV = Total Squared Variation
def calc_regularizer(image: np.array, tsv=False, p=None):
    if tsv and p == None:
        raise Exception("p value not set")
    reg = 0
    if tsv:
        image_lshift = np.copy(image, subok=True)
        image_lshift = np.roll(image_lshift, -1,axis=1)
        image_lshift[:,-1] = image_lshift[:,-2]
        image_upshift = np.copy(image, subok=True)
        image_upshift = np.roll(image_upshift, -1, axis=0)
        image_upshift[-1] = image_upshift[-2]

        term_1 = np.power(np.absolute(np.subtract(image_lshift, image)),p)
        term_2 = np.power(np.absolute(np.subtract(image_upshift, image)),p)
        reg = np.sum(np.add(term_1,term_2))
    return -1 * reg
```

Here we test the calc_regulaizer function using an empty image, a monotone image, and a noisy image.

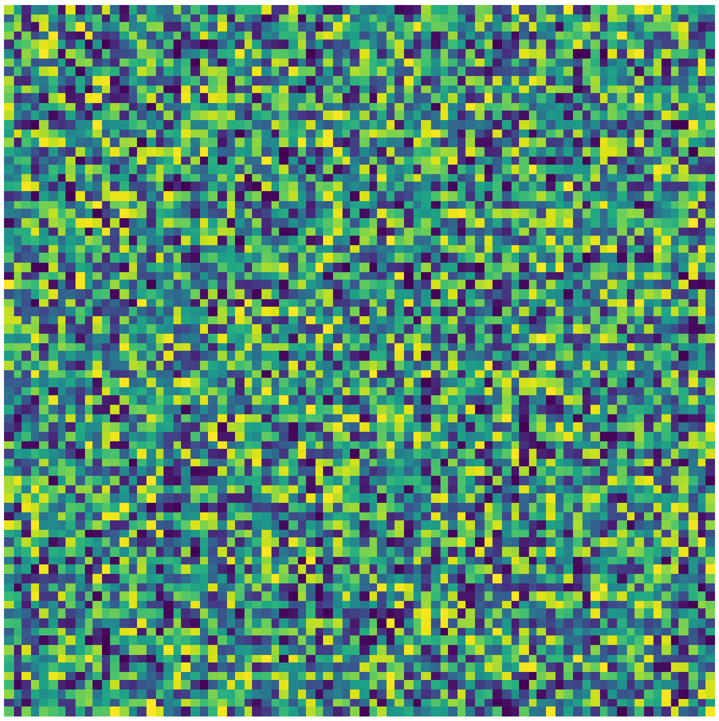
```
empty_image = np.zeros((80,80))
plt.figure()
plt.imshow(empty_image, vmin=0, vmax=1)
plt.axis('off')
```

```
(-0.5, 79.5, 79.5, -0.5)
```



```
noisy_image = np.random.rand(80,80)
plt.figure()
plt.imshow(noisy_image, vmin=0, vmax=1)
plt.axis('off')
```

```
(-0.5, 79.5, 79.5, -0.5)
```



```
monotone_image = np.full((80,80), 1)
plt.figure()
plt.imshow(monotone_image, vmin=0, vmax=1)
plt.axis('off')
```

```
(-0.5, 79.5, 79.5, -0.5)
```

```
print("Empty:",calc_regularizer(empty_image,True,2))
print("Noisy:",calc_regularizer(noisy_image,True,2))
print("Monotone:",calc_regularizer(monotone_image,True,2))
```

```
Empty: -0.0
Noisy: -2108.9772836537513
Monotone: 0
```

Here we see that the regularizer favors smooth images over noisy/rough ones. It is important to note that both the empty image and the monotone image have a regularizer of 0. This is because we weight the images towards smoother images.

This is the gradient of the regularizer. We calculate each point in the picture by doing $x_{i+1,j} + x_{i-1,j} + x_{i,j+1} + x_{i,j-1} - 4x_{i,j}$ for each i,j pixels

```
def gradient_regularizer(image: np.array):
    image_lshift = np.copy(image, subok=True)
    image_lshift = np.roll(image_lshift, -1,axis=1)
    image_lshift[:,-1] = image_lshift[:,-2]
    image_upshift = np.copy(image, subok=True)
    image_upshift = np.roll(image_upshift, -1, axis=0)
    image_upshift[-1] = image_upshift[-2]
    image_rshift = np.copy(image, subok=True)
    image_rshift = np.roll(image_rshift, 1,axis=1)
    image_rshift[:,0] = image_rshift[:,1]
    image_dshift = np.copy(image, subok=True)
    image_dshift = np.roll(image_dshift, 1, axis=0)
    image_dshift[0] = image_lshift[1]
    g_reg = 4 * image - image_lshift - image_upshift - image_rshift - image_dshift
    return g_reg
```

```
print("Empty:\n",gradient_regularizer(empty_image))
print("Noisy:\n",gradient_regularizer(noisy_image))
print("Monotone:\n",gradient_regularizer(monotone_image))
```

```
Empty:
 [[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
Noisy:
 [[ 1.04025833  0.31293167 -1.48128372 ...  0.22462667  0.2264166
   0.58445354]
 [-1.45402225  1.290966   -1.51713099 ...  1.15713575 -0.19833866
  -0.09194965]
 [ 0.1087026   1.74525518 -1.26408352 ... -0.34713719 -0.77749415
  -0.57635953]
 ...
 [ 1.18315853  0.50915739 -0.48980817 ... -1.33555771  2.27147516
   0.3352341 ]
 [-1.47661333  0.64698704 -1.56710514 ...  0.77084388 -2.04752708
  -0.40909334]
 [ 1.24388255 -1.59917979  1.76252233 ... -0.75868688  0.32312203
   0.84747094]]
Monotone:
 [[0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 ...
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]]
```

```
%run ./utility.ipynb
```

CHAPTER

FIVE

LOSS

The Loss function is trying to compare the data with the image and see if they agree with each other.

Let $I(x, y)$ be the image and we transform it into $\hat{I}(x, y)$ by using fourier transfroms described in the the interpolate section of the previous notebook.

Next we compare each interpolated point with its data term counterpart and get the equation for loss to be:

$$J = \sum(\frac{|\hat{I} - D|^2}{\sigma^2})$$

σ here is the error for each data point.

Finally, we add the regularizer to the loss using the method in the previous notebook.

5.1 Data Term calculation

We have above that $\hat{I}$ is in the Fourier Domain. In order to compute the data term, we calculate the term as $D = \text{Amp} * e^{i*\text{phase}}$.

It is important to note that the phase in Z is in radians, not degrees

```
def loss(image, data_list: list[data], coords, p = 2, reg_weight = 1, FOV = 100*u.uas.
↪to(u.rad)):
    error_sum = 0
    vis_images = interpolate(image, coords, FOV)

    for i in range(len(data_list)):
        vis_data = data_list[i].amp * np.exp(1j * math.radians(data_list[i].phase))
        vis_image = vis_images[i]
        error = (abs(vis_image-vis_data) / data_list[i].sigma) ** 2
        error_sum += error

    return error_sum + reg_weight * calc_regularizer(image=image, tsv=True, p=2)
```

5.2 Furthermore into the data

What we are doing here is a simplified version of the loss function. In reality, the loss is more complicated due to other factors.

Because of the method of combining signal data from many telescopes, we get errors that may throw off the data terms.

How would we change our loss function?

5.2.1 Method One: Gain

We add a new function called the Gain which represents the telescoping errors. It is a smooth function in time that has a phase and amplitude. We multiply it with the data term to result in:

$$J = \sum(\frac{|\hat{I} - GD|^2}{\sigma^2})$$

Now when we try to minimize the image, we are also trying to minimize G.

5.2.2 Method Two: Cosher Phase

The other way is to take into the account the Cosher Phase.

Using three telescopes we get:

$$\sum(\frac{|\phi_{1,2} + \phi_{2,3} + \phi_{3,1} - \phi_{\hat{I}}|^2}{\sigma^2})$$

$\phi_{1,2} + \phi_{2,3} + \phi_{3,1}$ is called the Cosher Phase CP which is independent of the telescope error and is taken from the data. $\phi_{\hat{I}}$ is a Cosher Phase computed from the image.

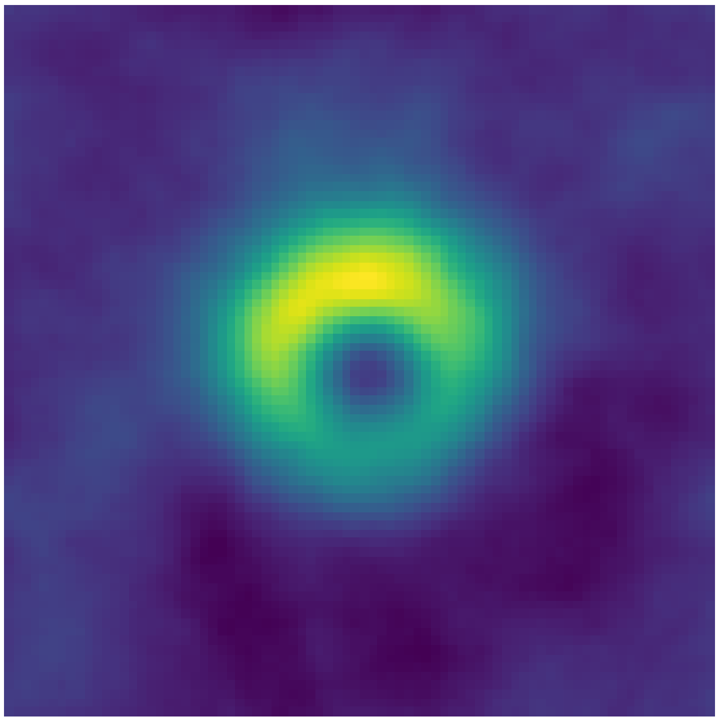
In practice, these are the two major methods in order to fit the image to the data.

5.3 Testing the function

Below we have a image that was generated for testing purposes. Sometimes we want to shift the image in someway so that the loss function can be optimized further. In EHT, they have tested with many variations and come up with the loss being the best with the ring centered in the middle. We will test this claim below.

```
sample = np.loadtxt("images/data.csv", delimiter=",")
plt.figure()
plt.imshow(sample)
plt.axis('off')
```

```
(-0.5, 79.5, 79.5, -0.5)
```



Here we generate a coords and data_list list by sampling data from the ring (not the noise around the ring).

```
def do_sample(n):
    coords = []
    data_list = []
    ft_image = np.fft.fftshift(np.fft.fft2(sample))
    for i in range(n):
        coords.append((int(np.random.rand()*10-5), int(np.random.rand()*10-5)))
        data_list.append((coords[i],ft_image[coords[i][0]+40][coords[i][1]+40]))
    return coords, data_list
```

```
coords, data_list = do_sample(25)
data_list
```

```
[((3, 0), (9.345246066663234-12.763088241472762j)),
 ((1, -4), (19.26638997942068+0.645458355184074j)),
 ((4, 1), (16.740754111448133+2.309720618467048j)),
 ((0, 0), (-130.7977532545254+0j)),
 ((3, 0), (9.345246066663234-12.763088241472762j)),
 ((4, 2), (-16.213537172215425+2.677957292860941j)),
 ((-4, -1), (16.740754111448133-2.3097206184670487j)),
 ((2, -2), (-20.259515490227393+0.5869595357657675j)),
 ((-3, 0), (9.345246066663236+12.763088241472765j)),
 ((-4, 0), (-21.092399479624063-7.428119584083979j)),
 ((3, -2), (18.046523876827386+2.5971752659760368j)),
 ((3, -2), (18.046523876827386+2.5971752659760368j)),
 ((-3, -2), (18.59914527451261+7.6023610122723255j)),
 ((0, 2), (45.25288306329196-3.200855947045237j)),
 ((2, 4), (-11.797382443780851+1.8726152619466332j)),
 ((-3, -2), (18.59914527451261+7.6023610122723255j)),
```

(continues on next page)

(continued from previous page)

```
 ((-3, 4), (10.97997772948687+0.8737280514156963j)),
 ((-4, 2), (-11.05946261320548+0.4895276507069246j)),
 ((2, -4), (-11.158564582308404+2.58553536706419j)),
 ((4, 0), (-21.092399479624063+7.428119584083978j)),
 ((2, -4), (-11.158564582308404+2.58553536706419j)),
 ((-3, 0), (9.345246066663236+12.763088241472765j)),
 ((4, 1), (16.740754111448133+2.309720618467048j)),
 ((-4, 2), (-11.05946261320548+0.4895276507069246j)),
 ((1, 3), (-11.35613468978443+3.361553558377294j))]
```

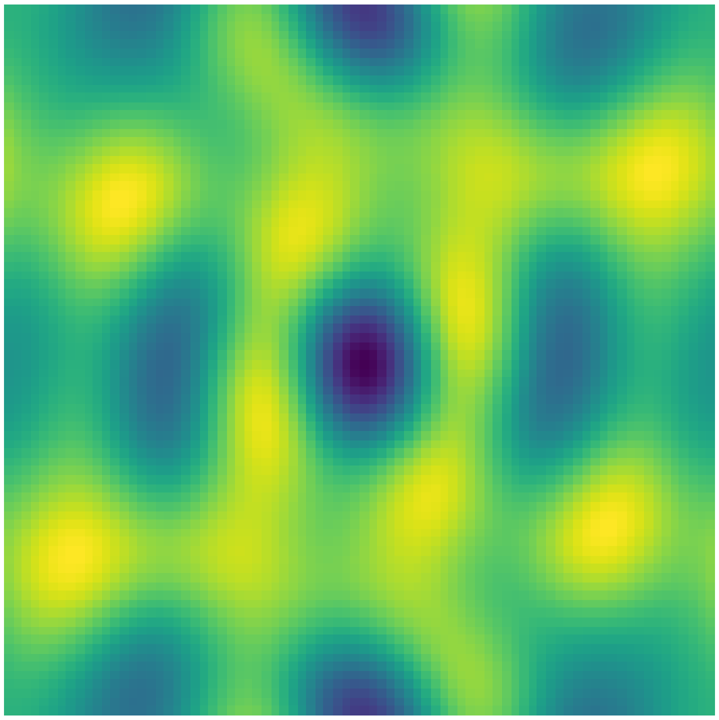
It is important to note here that the data points in data_list is complex. Since we sampled from the fourier transform, we have no need to calcuate the data term like we do in the loss function above. Below is a altered version of the loss function above.

```
# Assumption image is 80px by 80px => 6400 variables
def simple_loss(image, data_list: list[data], coords, p = 2, reg_weight = 1, FOV =␣
↪1): #Ask about u.rad
    error_sum = 0
    vis_images = interpolate(image, coords, FOV)

    for i in range(len(data_list)):
        vis_data = data_list[i][1]
        vis_image = vis_images[i]
        error = (abs(vis_image-vis_data)) ** 2
        error_sum += error

    return error_sum
```

Finally, below we calcuate the loss of the sample image we started with and then shift the image in all directions. Then we plot the array of losses that we obtained.

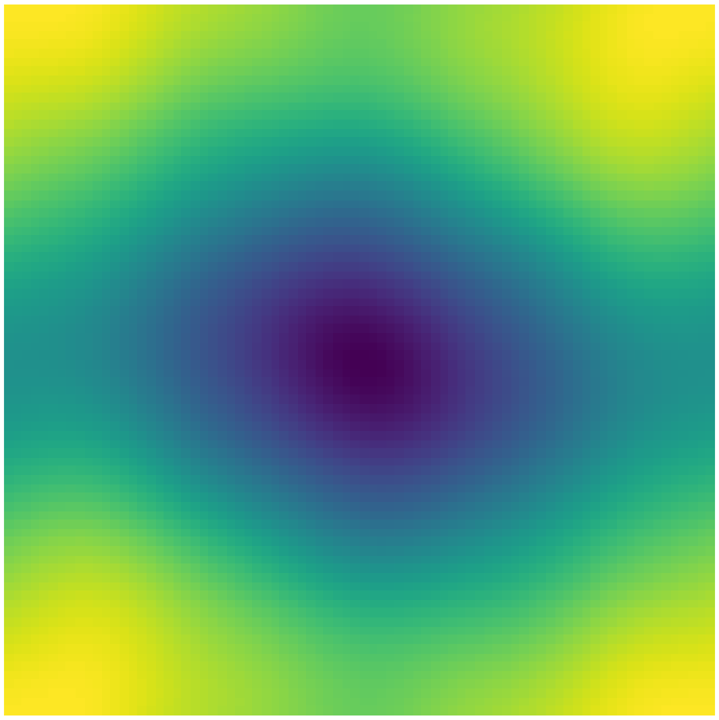
```
def calculate_losses():
    loss_arr = np.zeros((len(sample),len(sample[0])))
    for i in range(len(sample)):
        image_1 = np.roll(sample, i, axis=1) # Right shifts
        for j in range(len(sample[i])):
            image_2 = np.roll(image_1, j, axis = 0) # Up shifts
            loss_arr[i][j] = simple_loss(image_2, data_list, coords, reg_weight=0)

    # Note here for why we shift
    loss_arr1 = np.roll(loss_arr, 40, axis=1)
    loss_arr2 = np.roll(loss_arr1, 40, axis=0)

    plt.figure()
    plt.imshow(loss_arr2)
    plt.axis('off')
    return loss_arr
```

```
loss_arr = calculate_losses()
```

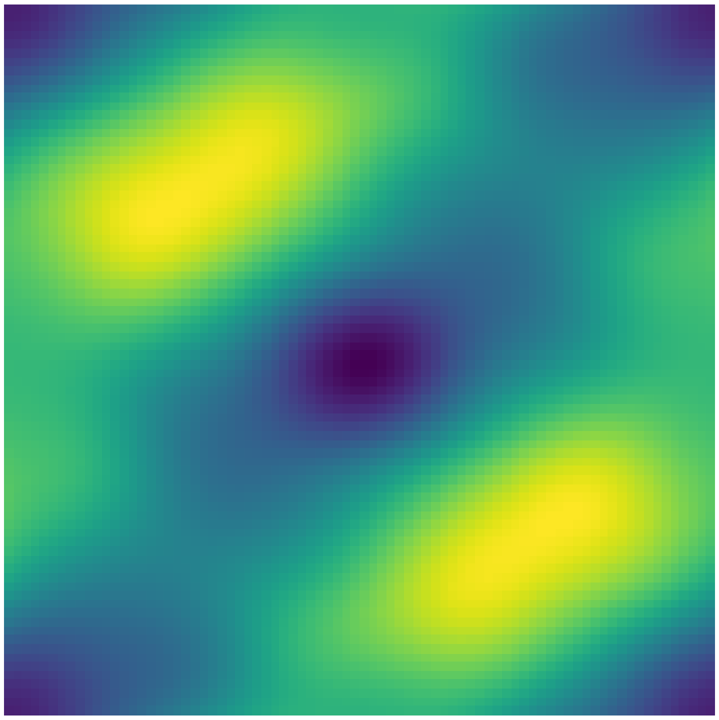


The dark spots in this picture are low points while the yellow spots are high points. Here we see that there are a few spots where the image produces low loss. If ran again, we will get differing behaviors

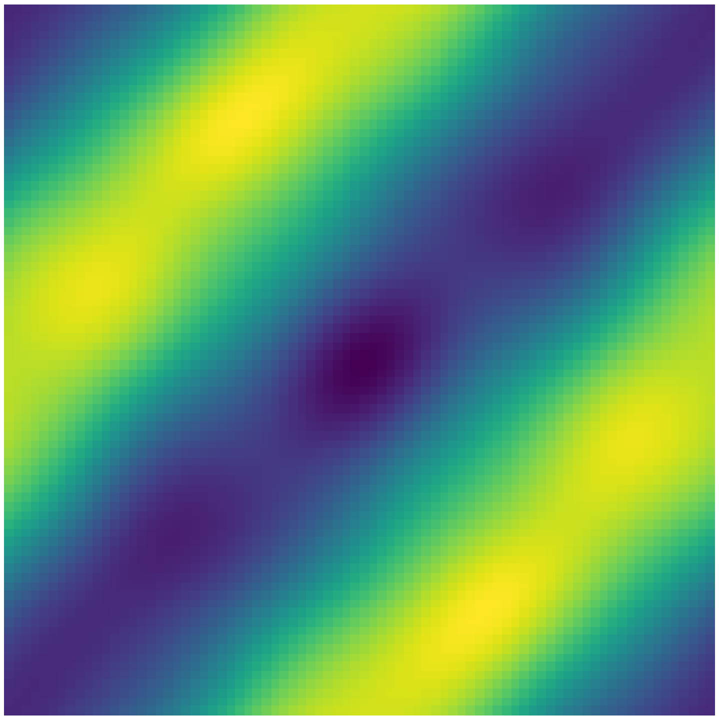
```
coords, data_list = do_sample(25)
loss_arr = calculate_losses()
```



```
coords, data_list = do_sample(25)
loss_arr = calculate_losses()
```



```
coords, data_list = do_sample(25)
loss_arr = calculate_losses()
```



In all four of these trials (and other reruns that we test), we generally see dark spots in the centers suggesting that the best area to put the ring is centered in the middle of the image

```
%run ./utility.ipynb
```

CHAPTER

SIX

GRADIENT CALCULATIONS: FINITE DIFFERENCES

We have two different methods of computing gradients and in this notebook we will explore the first one.

6.1 Gradient's Background

A gradient is the direction of greatest change when looking at a scalar function. It is usually described as a generalized derivative or the rate of change. In our model, the gradient will be positive or negative depending on if a pixel's value must in increased or decreased. Additionally, each pixel will have its own gradient.

Here's an intuitive way of looking at gradients: Image you are standing in Tucson at Alvernon and Grant and we are interested in the function f that tells you your elevation. The gradient of f will always tell you which direction you should travel in in order to rise in elevation the quickest.

Since gradients are consistent (meaning if initial conditions are constant and the function doesn't change then we always get the same result), we can use gradients to find local maximums and minimums by simply following the gradient until either the gradient is 0 or we never finish (in the event of infinity end behaviors).

Why are gradient's important? In the real world, gradients are used in many fields like physics, robotics, and optimizations. We use it all the time in order to quanify the net rate of change in multi-variable functions!

6.2 Method one: Finite Differences Methods

The Finite Differences Method are a numerical analysis technique for solving differential equations by approximating derivatves using finite differences. Using these finite differences we can approximate a gradient of a function which we will denote as ∇f.

So what is Finite Differences exactly? It is a mathematical expression of the form $f(x+b) - f(x+a)$.

There are three basic types that are commonly considered for this method: Forward, Backward, and Central.

Forward differences is calcuated by $\nabla f = \frac{f(x+h)-f(x)}{h}$

Backward differences is calcuated by $\nabla f = \frac{f(x)-f(x-h)}{h}$

Central differences is calcuated by $\nabla f = \frac{f(x+\frac{h}{2})-f(x-\frac{h}{2})}{h}$

6.2.1 The Function Implemented

Below we have all three methods implemented controlled by a mode flag.

For Gradient Descent, we consider f to be our loss function and h to be a minute difference from a pixel's value. We iterate over every pixel and calculate the loss needed for equation used ($f(x+h)$, $f(x-h)$, or $f(x+\frac{h}{2}) - f(x-\frac{h}{2})$). This gives us the gradient for each pixel.

```
def gradient_finite_differences(data_list: list[data], coords, image, mode = 1): # 0␣
↪For central, -1 for backward, 1 for forward
    image_copy = np.copy(image, subok=True)
    upper_diff: float
    lower_diff: float
    h: float
    gradient_arr = np.empty(np.shape(image),dtype=np.complex_)
    if (mode == 0): # Central difference
        for row in range(len(image)):
            for col in range(len(image[row])):
                image_copy[row,col] += 1e-6 / 2
                upper_diff = simple_loss(image_copy, data_list, coords)
                image_copy[row,col] -= 1e-6
                lower_diff = simple_loss(image_copy, data_list, coords)
                image_copy[row,col] = image[row,col] # Reset that pixel to original␣
↪value
                gradient_arr[row,col] = (upper_diff - lower_diff) / 1e-6
    elif (mode == -1): # Backward difference
        upper_diff = simple_loss(image, data_list, coords)
        for row in range(len(image)):
            for col in range(len(image[row])):
                image_copy[row,col] -= 1e-8
                lower_diff = simple_loss(image_copy, data_list, coords)
                gradient_arr[row,col] = (upper_diff - lower_diff) / 1e-8
                image_copy[row,col] = image[row,col]
    elif (mode == 1) : # Forward difference is default
        lower_diff = simple_loss(image, data_list, coords)
        for row in range(len(image)):
            for col in range(len(image[row])):
                image_copy[row,col] += 1e-8
                upper_diff = simple_loss(image_copy, data_list, coords)
                gradient_arr[row,col] = (upper_diff - lower_diff) / 1e-8
                image_copy[row,col] = image[row,col]
    else:
        raise ValueError('Incorrect mode for finite differences')
    return gradient_arr.real
```

Note: We are using "simple_loss" in this because we are using the synthetic data. For using this on real data, use the "loss" function.

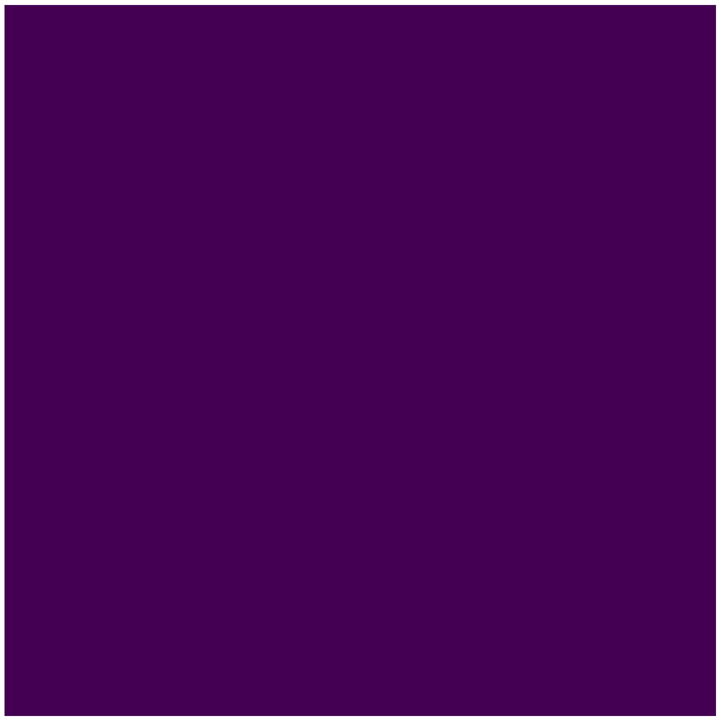
The version in the utility file uses "loss".

6.2.2 Demo Walkthrough

Start with an empty or blank image

```
emp = np.zeros((80,80))
plt.figure()
plt.imshow(emp)
plt.axis('off')
```

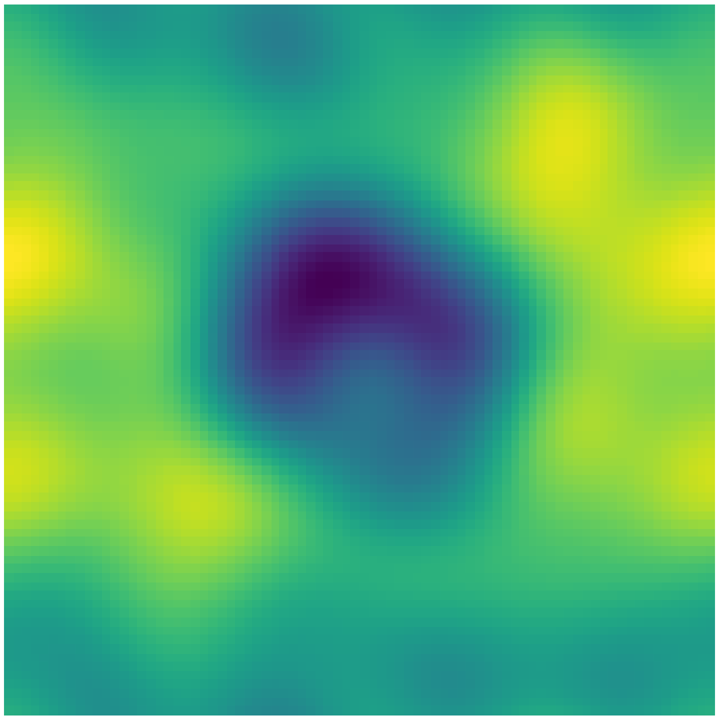
```
(-0.5, 79.5, 79.5, -0.5)
```



Next we get the sample data points from our synthetic data. Afterwards we will run our gradient code.

```
sample = np.loadtxt("images/data.csv", delimiter=",")
coords, data_list = do_sample(50)
x = gradient_finite_differences(data_list, coords, emp)
plt.figure()
plt.imshow(x)
plt.axis('off')
```

```
(-0.5, 79.5, 79.5, -0.5)
```



Above is the image representation of the gradient, darker areas represent smaller gradients and lighter areas represent bigger gradients.

It is specifically a gradient when we start our "image" as an empty or blank image. The gradient above shows which pixels are different than what we would expect given a data set (whether some should be brighter, darker or stay the same).

6.3 Gradient Descent

Now that we have computed a gradient, we can now start doing gradient descent. Gradient Descent is an optimization algorithm for finding local minimum within a differentiable function. We can use this to find values of parameters that minimize some sort of cost function. Here, the parameters as simply each pixel in our image, and the cost functions that we are trying to minimize is the loss function.

Heres how it works. We first calcualte the loss at the current position. The algorithm then iteratively calculates the next image by using the gradient at the current position. We scale the gradient so that we can obtain a loss less than the current loss and then subtract the scaled gradient from the image. This is called a single step. We subtract the gradient because we are looking to minimise the function rather than maximizing it.

We take a bunch of these steps until we hit some stopping condition (here it is np.min(np.abs(grad)) $\leq$ 0.0000001).

```
def gradient_descent(image, data_list, coords, subset_percent = 10):
    image_copy = np.copy(image, subok=True) # Uses copy of the image due to lists␣
↪being mutable in python
    i = 0
    grad = None
    # Can also use max here, min just makes it finish quicker
    while grad is None or np.min(np.abs(grad)) > 0.0000001:

        t = 10000000 # Initial Step size which resets each iteration
```

(continues on next page)

(continued from previous page)

```
        prev_loss = simple_loss(image_copy, data_list, coords)
        grad = gradient_finite_differences(data_list, coords, image_copy)

        new_image = image_copy - t * grad.real
        new_loss = simple_loss(new_image, data_list, coords)

        while new_loss > prev_loss: # Only run when new_loss > prev_loss
            new_image = image_copy - t * grad.real
            new_loss = simple_loss(new_image, data_list, coords)
            t /= 2

        image_copy -= t * 2 * grad.real # Multiply by 2 to undo last divide in the␣
↪while loop
        i += 1
        if i == 1000: # Hard stop here for notebook purposes
            return image_copy
    return image_copy
```

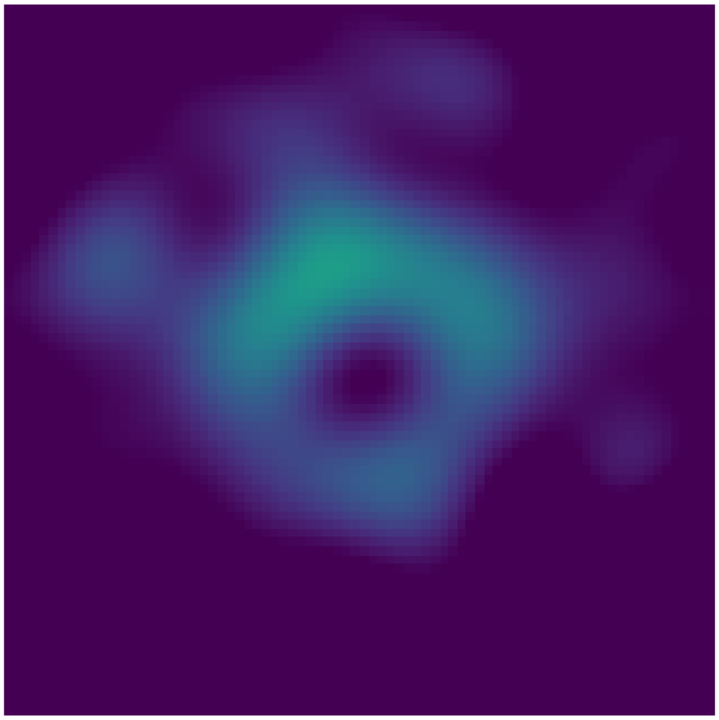
Here is what the gradient descent's output looks like after 1000 iterations. The code should be ran using

```
reconstructed_img = gradient_descent(emp, data_list, coords)
```

The code isn't ran here because the notebook doesn't allow for something to run for extended periods of time (upwards of 10 min).

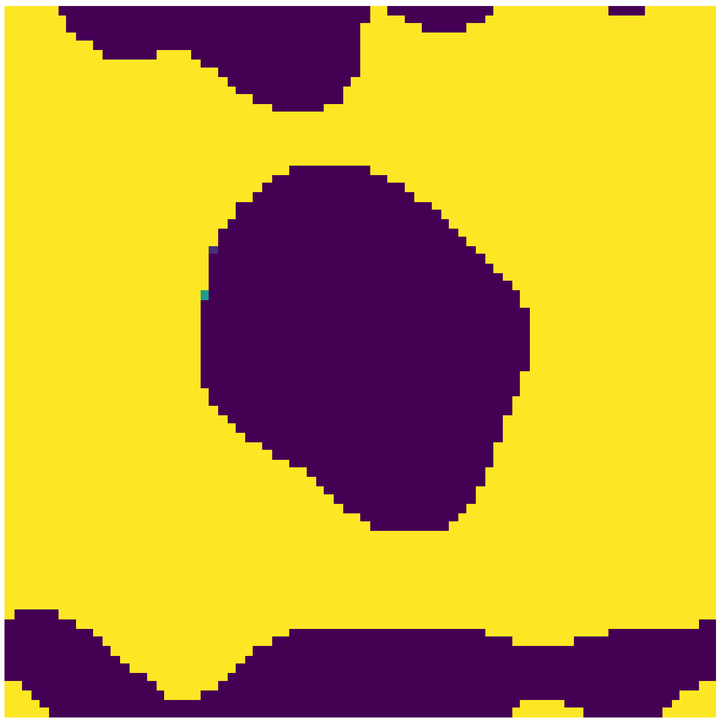
```
reconstructed_img = np.loadtxt("images/reconstructed_img.csv", delimiter=",")
plt.figure()
plt.imshow(reconstructed_img, vmin=0, vmax=np.max(sample))
plt.axis('off')
```

```
(-0.5, 79.5, 79.5, -0.5)
```



```
plt.figure()
plt.imshow(x, vmin=0, vmax=np.max(sample))
plt.axis('off')
```

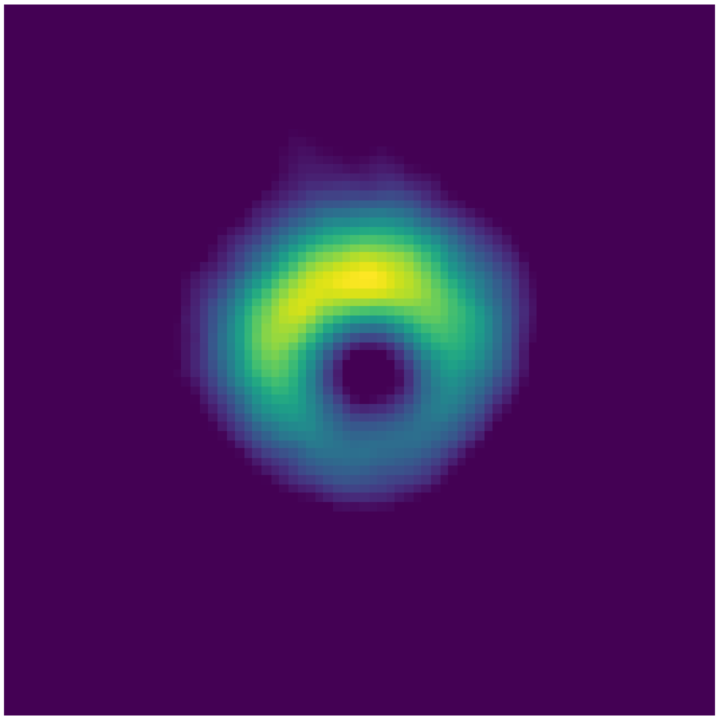
```
(-0.5, 79.5, 79.5, -0.5)
```



Here is the original image that the data was taken from

```
plt.figure()
plt.imshow(sample, vmin=0, vmax=np.max(sample))
plt.axis('off')
```

```
(-0.5, 79.5, 79.5, -0.5)
```



As you can see here, the reconstructed image doesn't replicate the original image directly but it will still reconstruct some of the big important features that EHT Analysis uses. Here we can see the general shape of the reconstructed image matches up with the original image

```
%run ./utility.ipynb
```

CHAPTER

SEVEN

GRADIENT CALCULATIONS: “DIRTYING” THE IMAGE

7.1 Method two: Dirtying the Image

Here, we use a different method of computing gradients in order to speed up the gradient calculation process. By doing so, we should also see an increase in speed for the gradient descent.

```
sample = np.loadtxt("images/data.csv", delimiter=",")
coords, data_list = do_sample(25)
emp = np.zeros((80,80))
```

Here we calculate gradient using some preprocessing and derivatives. We also reduce time by using a selection approach.

```
def preprocess_gradient(data_list, coords, image):
    r, c = np.shape(image)
    preprocessed = np.empty([r,c,len(data_list),2], dtype=np.complex_)
    for row in range(len(image)):
        for col in range(len(image[row])):
            for datum in range(len(data_list)):
                term = ((2*np.pi*1j)/image.size)*(row*coords[datum][0] +␣
↪col*coords[datum][1]) #.size for numpy array returns # of rows * # of cols
                term_1 = np.exp(term)
                term_2 = np.exp(-1*term)
                preprocessed[row,col,datum,0] = term_1
                preprocessed[row,col,datum,1] = term_2
    return preprocessed
```

```
coeffs = preprocess_gradient(data_list, coords, emp)
```

```
def dirty_gradient(data_list: list[data], coords, coeffs, image, subset_percent = 10,␣
↪FOV = 100*u.uas.to(u.rad)):
    gradient_arr = np.empty(np.shape(image)) # Because we are in real space
    vis_images = interpolate(image, coords, FOV)
    selection = np.random.choice(np.arange(len(data_list)), size=len(coords)*subset_
↪percent//100, replace=False) # selection is full of indicies
    for row in range(len(image)):
        for col in range(len(image[row])):
            gradient_sum = 0
            for i in selection:
                vis_data = data_list[i].vis_data
                vis_image = vis_images[i]
                term_1 = coeffs[row,col,i,0] * (np.conj(vis_image) - np.conj(vis_
↪data))
```

(continues on next page)

(continued from previous page)

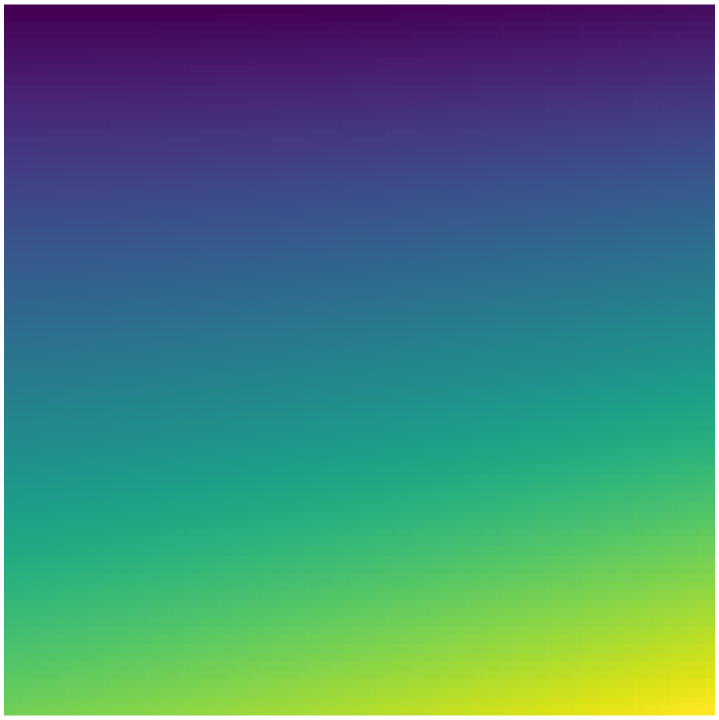
```
                term_2 = coeffs[row,col,i,1] * (vis_image - vis_data)
                gradient_sum += (term_1 + term_2)/(data_list[i].sigma ** 2)
            gradient_arr[row,col] = gradient_sum
    return gradient_arr
```

```
def dirty_gradient_simple(data_list: list[data], coords, coeffs, image, subset_
↪percent = 10, FOV = 100*u.uas.to(u.rad)):
    gradient_arr = np.empty(np.shape(image)) # Because we are in real space
    vis_images = interpolate(image, coords, FOV)
    # selection = np.random.choice(np.arange(len(data_list)), size=len(coords)*subset_
↪percent//100, replace=False) # selection is full of indicies
    for row in range(len(image)):
        for col in range(len(image[row])):
            gradient_sum = 0
            for i in range(len(data_list)):
                vis_data = data_list[i][1]
                vis_image = vis_images[i]
                term_1 = coeffs[row,col,i,0] * (np.conj(vis_image) - np.conj(vis_
↪data))
                term_2 = coeffs[row,col,i,1] * (vis_image - vis_data)
                gradient_sum += (term_1 + term_2)/(1 ** 2) # What should I set sigma␣
↪to?
            gradient_arr[row,col] = gradient_sum
    return gradient_arr
```

```
x = dirty_gradient_simple(data_list, coords, coeffs, emp)
plt.figure()
plt.imshow(x)
plt.axis('off')
```

```
/var/folders/_b/trlmhkgj5xq968yccj4vtg1c0000gn/T/ipykernel_69440/1027418742.py:14:␣
↪ComplexWarning: Casting complex values to real discards the imaginary part
  gradient_arr[row,col] = gradient_sum
```

```
(-0.5, 79.5, 79.5, -0.5)
```



```python
def gradient_descent(image, data_list, coords, coeffs):
    image_copy = np.copy(image, subok=True) # Uses copy of the image due to lists␣
↪being mutable in python
    i = 0
    grad = None
    # Can also use max here, min just makes it finish quicker
    while grad is None or np.min(np.abs(grad)) > 0.0000001:

        t = 10000000 # Initial Step size which resets each iteration
        prev_loss = simple_loss(image_copy, data_list, coords)
        grad = dirty_gradient_simple(data_list, coords, coeffs, image_copy)

        new_image = image_copy - t * grad.real
        new_loss = simple_loss(new_image, data_list, coords)

        while new_loss > prev_loss: # Only run when new_loss > prev_loss
            new_image = image_copy - t * grad.real
            new_loss = simple_loss(new_image, data_list, coords)
            t /= 2

        image_copy -= t * 2 * grad.real # Multiply by 2 to undo last divide in the␣
↪while loop
        i += 1
        if i == 1000: # Hard stop here for notebook purposes
            return image_copy
    return image_copy
```

```python
#x = gradient_descent(emp,data_list,coords,coeffs)
```

```python
plt.figure()
plt.imshow(x)
plt.axis('off')
```

```
(-0.5, 79.5, 79.5, -0.5)
```