# Exploring Gradient Descent On Image Reconstruction with EHT data

**Anthony Hsu**

**May 01, 2024**

# CONTENTS

# Exploring Gradient Descent On Image Reconstruction with EHT data

Abstract: The Event Horizon Telescope (EHT) has revolutionized our understanding of black holes by using Very Long Baseline Interferometry (VLBI) to reconstruct the highest resolution images of black holes. It combines data from a global network of radio telescopes to observe the immediate surroundings of black holes, particularly focusing on the supermassive black hole in the galaxy M87 and the center of our Milky Way. This Jupyter Book explores the intricate process of image reconstruction from EHT data, highlighting several critical aspects such as interpolation, loss calculation, and gradient computation methods.

Interpolation schemes play a pivotal role in connecting the snap averaged data in EHT's sparse observation and reconstructing an image. By examining the relationship between the image and Fourier domains, we gain insights into the factors influencing high-resolution image recreation. One approach to image reconstruction involves creating a loss function and minimizing it. The loss function quantifies how well the image matches the observational data and how same or expected the appearance of the image looks. The latter part of a loss function is called a regularizer. The optimum (the best performing image) is accepted as the image reconstruction of the data.

Additionally, this notebook assesses two distinct methods for computing gradients: finite differences and an "dirtying the image" approach. By comparing their efficiency and accuracy, we aim to offer guidance on selecting appropriate gradient computation methods for image reconstruction tasks. Integrating these components, we perform gradient descent on sample test images to demonstrate practical applications of these theories.

CHAPTER

ONE

# INTRO AND SETUP USE DIFF

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
```

## 1.1 Data Details

In order to capture the image and data of a black hole, we would need a telescope the size of the earth! Although we don't have one of those, the Event Horizon Telescope were still able to obtain some data about black holes by using a technology called Very Long Baseline Interferometry (VLBI). This involves having many telescope around the world. This in turn creates a virtual telescope and allows us to capture the image of the black hole's silouette. They are the only ones in the world to do so!

Most of the optics like cameras and cellphones, they don't hit a limit called the Defraction Limit. This limit is the point at which two Airy patterns are no longer distinguishable from each other and is often known as the cutoff frequence of a lens. Since EHT wants to image things that are extremely far away, they are constantly working at the Defraction limit and are limited by the physics of light.

Because of these limitations, EHT is only able to collect sparse data in the Fourier domain (rather than an actually picture image). Thus in order to obtain a picture, we must do a image reconstruction as an optimization problem in order to fill in the gaps. The data set comes from EHT (The Event Horizon Telescope) and from the HOPS pipeline (software from MIT).

## 1.2 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. They are given in terms of wavelengths (lambda).

Iamp is the Amplitude of the Fourier Coefficient

IPhase is the Phase of the Fourier Coefficient in degrees

ISigma is the estimated magnitude of 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

```python
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

```python
def process_data(data_df):
    """
    Processes the data in the dataframe into a coords list and data objects
    Args:
        data_df is a pandas data frame of the data
    Returns:
        a list of coordinates in u,v space
        a list of data objects
    """

    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)']))
        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):
    """
    reads the data from a file into a pandas dataframe
    Args:
        filename is a string that represents a csv file
    Returns:
        a pandas dataframe
    """
    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]
```

Below is EHT's reconstruction of the data above

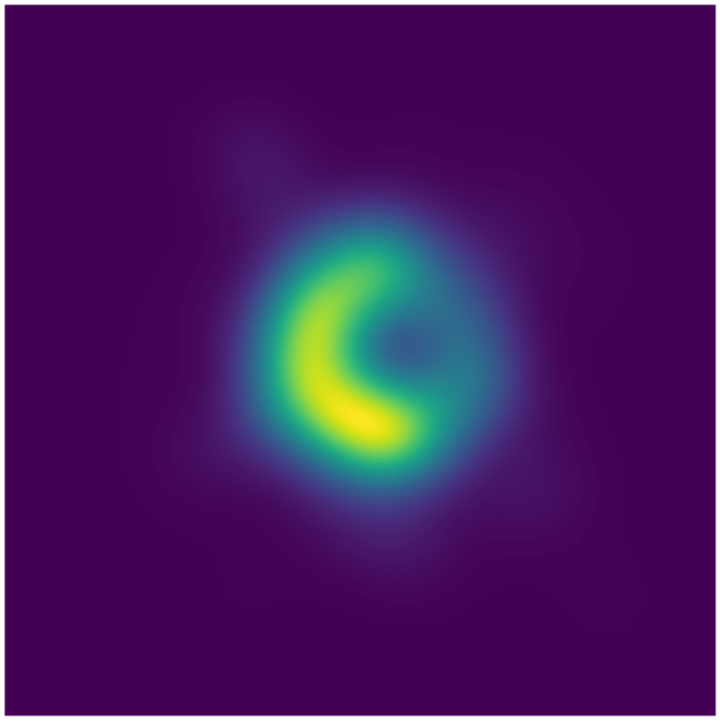
```
file = open("images/3597_blur_avg.txt","r")
lines = file.readlines()
image = np.empty([180,180])
t = []
for line in lines:
    coord = line.strip().split()
    x = int(float(coord[0]) * 1000000)
    y = int(float(coord[1]) * 1000000)
    z = int(float(coord[2]) * 10000000000)
    image[x+90][y+90] = z
```

(continues on next page)

(continued from previous page)

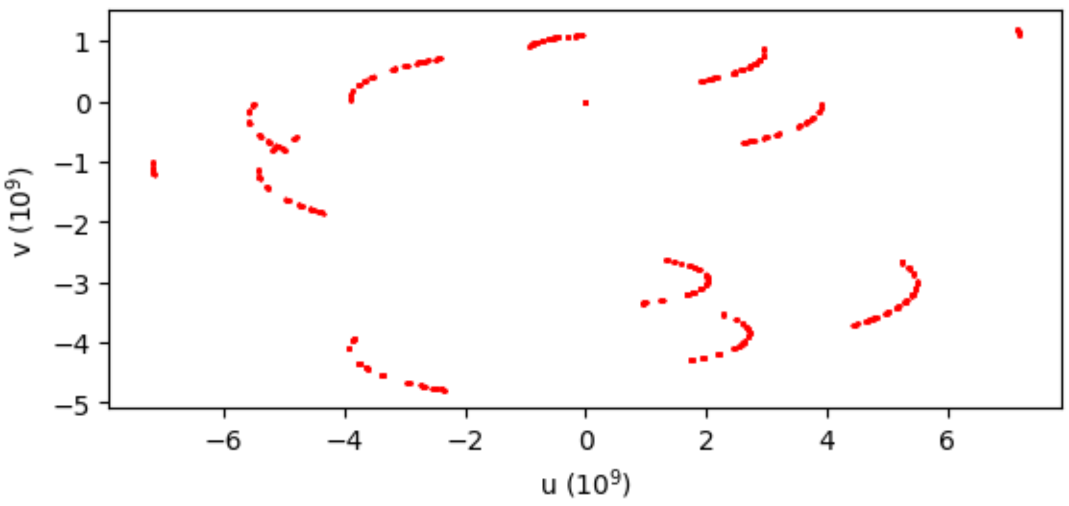
```
plt.figure()
plt.imshow(image)
plt.axis('off')
```

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



And here is what the data looks like in the Fourier Domain

```
FOV=180*u.uas.to(u.rad)
kx_1 = (1)/FOV

x,y = [], []
for data in data_list:
    x.append((data.u/kx_1))
    y.append(data.v/kx_1)

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

plt.scatter(x,y,color="red",s=0.1)
plt.xlabel("u ($10^9$)")
plt.ylabel("v ($10^9$)")
plt.gca().set_aspect("equal")
```


1
0
−1
−2
−3
−4
−5
v ($10^9$)
−6
−4
−2
0
2
4
6
u ($10^9$)

CHAPTER

TWO

# 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.

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

## 2.1 Background on FOV and k-space

FOV stands for Field Of View and it refers to the angle over which an image is acquired or displayed. These images are usually captured by some sort of signal processing like telescopes and MRIs. The number of pixels 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 k is the wavenumber which represents the number of wave cycles per unit length. Lets find the value of k in terms of FOV.

In these notebooks we are using discrete Fourier Transform in order to convert from real space to the Fourier domain. We are thinking of the FOV by FOV as a 2-dimensional torus where everything is periodic on it. The Fourier Transform then performs a function on the torus, the lowest harmonic would be related to sin or cos with the lowest possible frequency so that it is still periodically on the torus. When chooseing a k, if k is too small, then there will be less than one wavelength along the torus.

Here we will define the Fourier transform to be

$$\hat{f}(n,m) = \int_0^{FOV}\int_0^{FOV} f(x,y)e^{2\pi i\Delta knx+2\pi i\Delta kmy}dxdy$$

Ideally, we would like w to be the period of the smallest harmonic in one direction. Thus we look at sin $2\pi\Delta kx$ harmonics. Considering the slowest harmonics, then we let n = 1. As x goes from 0 to FOV, we want to end back at where we started. This means that we want $2\pi\Delta kFOV = 2\pi$. Thus $\Delta k = \frac{1}{FOV}$.

## 2.2 Interpolation

In this notebook, we have an image that is being linearly transformed using a Fourier transform. 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, due to the periodicity of the torus, extracted from the image which do not necessarily line up with the U and V coordinates from the dataset. Therefore, 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 more pixels the image will have. 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.

```python
def interpolate(image, coords, FOV):
    """
    Interpolates the values of each coordinate in coords in the Fourier domain of␣
 ↪image
    Args:
        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.
    Returns:
        The interpolated values at the coordinates based on image
    """

    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.

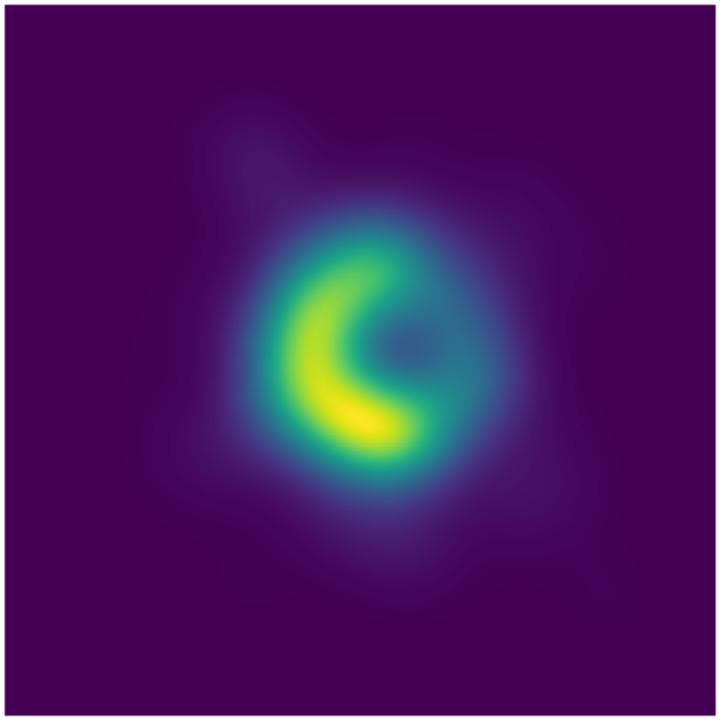
```python
x = [0,1,2,3,4]
y = [0,1,2,3,4]
z = [[1,1,1,1,1],[2,2,2,2,2],[3,3,3,3,3],[4,4,4,4,4],[5,5,5,5,5]]
interp = RegularGridInterpolator((x,y),z)
print("Expected: 1, Actual:", interp([0,3]))
print("Expected: 2.5, Actual:", interp([1.5,2.5]))
```

```
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)
```

### 2.2.1 Cubic Splines

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.

We call a piece piecewise cubic function a cubic spline if it interpolates a set of data points while also guaranteeing the smoothness at the data points observed. Splining is often used instead of Lagrange polynomial interpolations because it yields similar results and avoids the Runge's phenomenon for higher degree polynomials.

```
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
```

## 2.3 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

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

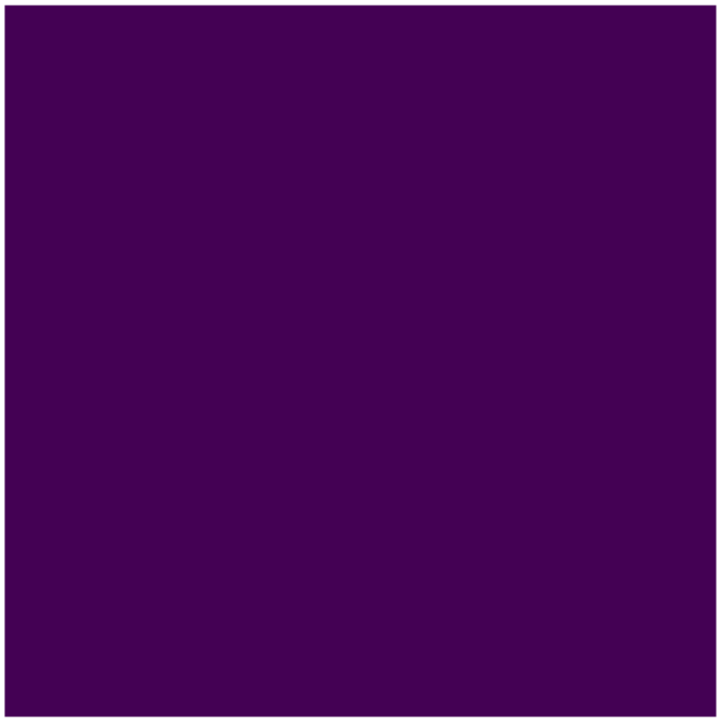
where l and m are pixel coordinates and I is the image. There are however other similar regularizers like Total variation that favors pixel-to-pixel smoothness.

```python
def calc_regularizer(image: np.array, tsv=False, p=None):
    """
    Calculates the regularizer according to total squared variation
    Args:
        image is a 80x80 pixel image that represents our reconstructed image
        p is the kind of norm to be used
        tsv is the flag for total squared variation
    Returns:
        the regularizer
    """
    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.

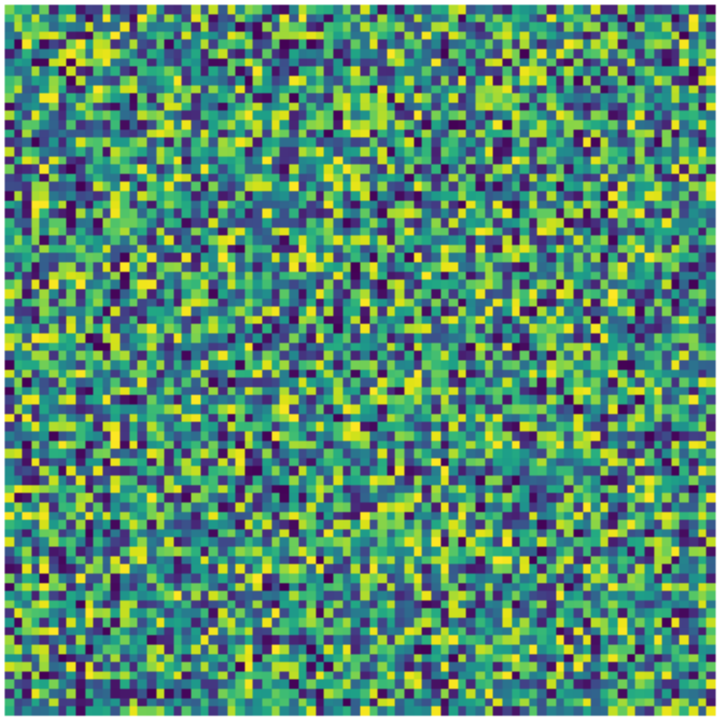
```
empty_image = np.zeros((80,80))
plt.figure()
plt.imshow(empty_image, vmin=0, vmax=1)
plt.axis('off')
```

```
(-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: -2175.392787838805
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):
    """
    Calculates the gradient of the regularizer
    Args:
        image is a 80x80 pixel image that represents our reconstructed image
    Returns:
        the gradient of the regularizer
    """
    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)
```

(continues on next page)

(continued from previous page)

```
    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:
 [[ 0.27110122 -0.26792028  0.24373484 ... -1.78212619 -0.25798071
  -1.75820855]
 [-0.05188315  0.74498931 -1.19863679 ...  2.11052592 -0.3928655
   1.5844631 ]
 [-0.83281295  1.32525577 -0.34310046 ...  1.91057963 -0.75566079
  -1.09022339]
 ...
 [-1.7355526   1.9401696  -1.88848139 ... -0.33351843 -0.75294465
  -0.99623846]
 [ 1.09733057 -0.95732659 -0.28356497 ... -1.43437622  2.56158941
  -1.81351   ]
 [ 0.35675887 -0.62055941  0.58023011 ...  1.47917241 -1.91481278
   1.58313622]]
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]]
```

CHAPTER

THREE

# 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:

$$J = \sum_i \frac{|\hat{I}(u_i, v_i) - D_i|^2}{\sigma_i^2}$$

$\sigma_i, u_i, v_i, D_i$ are all related to the $i^{th}$ data point.

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

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

## 3.1 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)):
    """
    calculates the loss of an image compared to the data given
    Args:
        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
        p is the kind of norm to be used
        reg_weight is the regularizer weight
        FOV is the Field of view from the telescopes. For the EHT data, our FOV is␣
↪100 micro ascs.
    Returns:
        a loss value
    """
    error_sum = 0
    vis_images = interpolate(image, coords, FOV)

    for i in range(len(data_list)):
```

(continues on next page)

(continued from previous page)

```
        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)
```

## 3.2 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?

### 3.2.1 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:

$$J = \sum_i \frac{|\hat{I}(u_i, v_i) - G_i D_i|^2}{\sigma_i^2}$$

$\sigma_i, u_i, v_i, D_i, G_i$ are all related to the $i^{th}$ data point.

This loss function becomes the one we are trying to minimize.

### 3.2.2 Method Two: Closure Phase

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

We construct these terms by summing over the data and using triples of telescopes:

$$\sum_{i,j,k} \frac{|\phi_{i,j} + \phi_{j,k} + \phi_{k,i} - \phi_{\hat{I}}|^2}{\sigma^2} \text{ where } i \neq j \neq k$$

$\phi_{i,j} + \phi_{j,k} + \phi_{k,i}$ is called the Closure Phase $CP$ which is independent of the telescope error and is taken from the data. $\phi_{\hat{I}}$ is a Closure Phase computed from the image. Closure Phase as defined here is independent from complex visibilities from the telescopes. This is because complex visibilities affect Amplitude (which doesn't appear here) and phase. The complex visibilites affect phase in an additive fashion and in this triple combination, the phase's complex visibilities cancel.

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

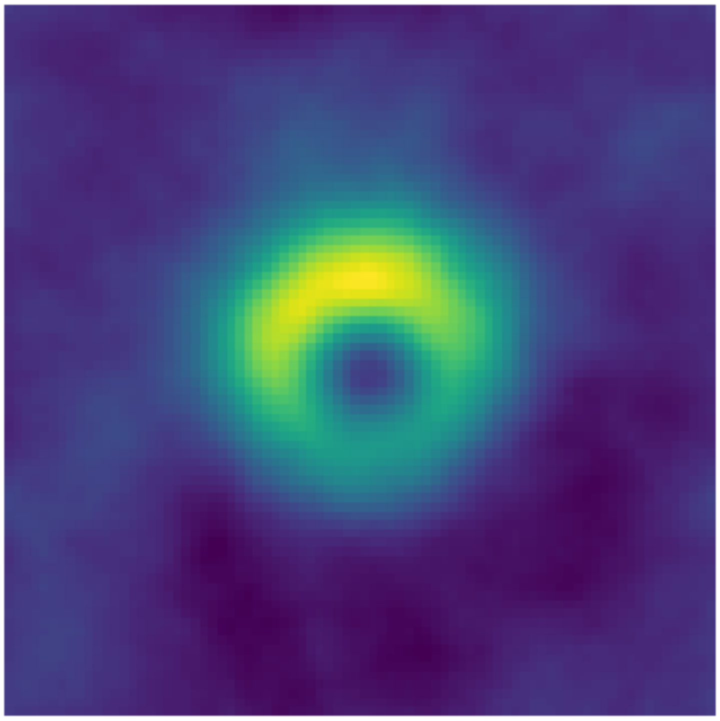
For the purposes of this notebook however we will be simplifying the problem and setting these visibilities to one.

## 3.3 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):
    """
    Collects n samples from a sample image
    Args:
        n is the number of samples as an integer
    Returns:
        a list of coordinates in u,v space
        a list of data objects
    """
    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)))
```

(continues on next page)

(continued from previous page)

```
        data_list.append(data(coords[i][0], coords[i][1], 0, ft_
↪image[coords[i][0]+40][coords[i][1]+40], 1))
    return coords, data_list
```

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

```
[[u: -2, v: -3],
 [u: 1, v: 0],
 [u: 4, v: -2],
 [u: 4, v: -2],
 [u: -4, v: 2],
 [u: 4, v: 3],
 [u: 0, v: 1],
 [u: 4, v: -3],
 [u: 4, v: 0],
 [u: -4, v: 4],
 [u: 1, v: -3],
 [u: -3, v: -2],
 [u: 3, v: -4],
 [u: 1, v: -3],
 [u: -4, v: -1],
 [u: 1, v: -2],
 [u: 2, v: 0],
 [u: 4, v: 0],
 [u: 2, v: -1],
 [u: -3, v: -2],
 [u: 2, v: 1],
 [u: 2, v: -4],
 [u: 3, v: 1],
 [u: -2, v: 1],
 [u: 4, v: 4]]
```

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.

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.

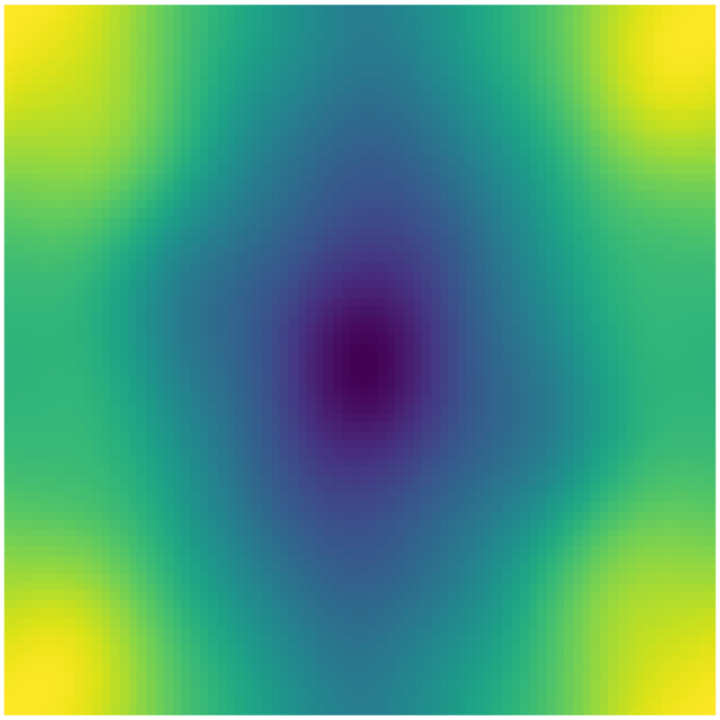
```
def calculate_losses():
    """
    Calculates the losses of the image by shifting it around
    Args:
        None
    Returns:
        an array of losses where the index is how much the image is shifted starting␣
↪with -40 to 40
    """
    loss_arr = np.zeros((len(sample),len(sample[0])))
    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
```

(continues on next page)

(continued from previous page)

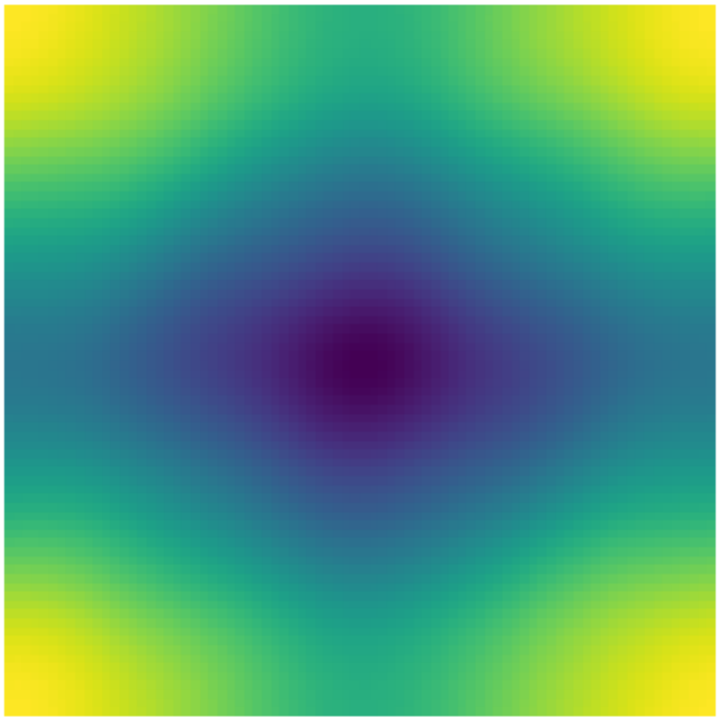
```
                loss_arr[i][j] = loss(image_2, data_list, coords, reg_weight=0, FOV=1)

    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()
```

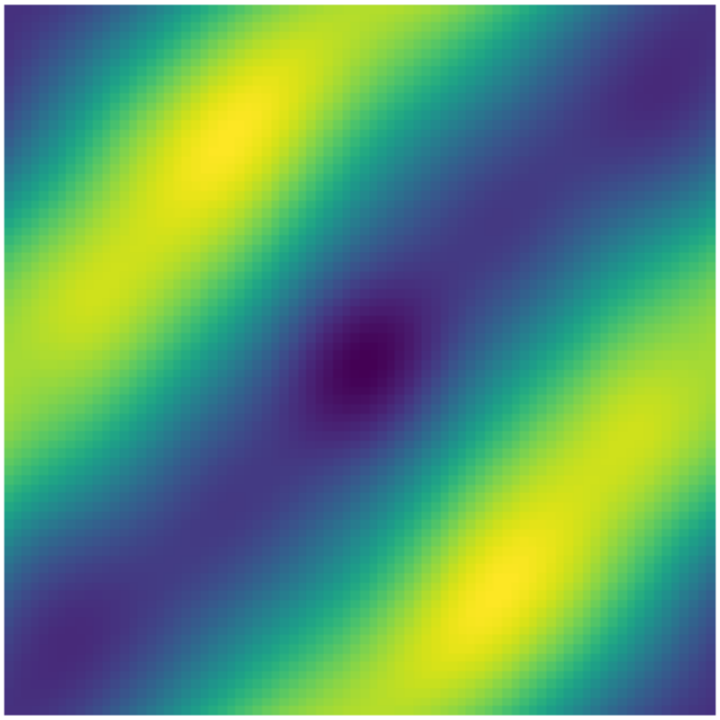


The dark spots in this picture are low points while the yellow spots are high points. Here we see that there are a few spots where the image produces low loss. If ran again, we will get differing behaviors

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



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

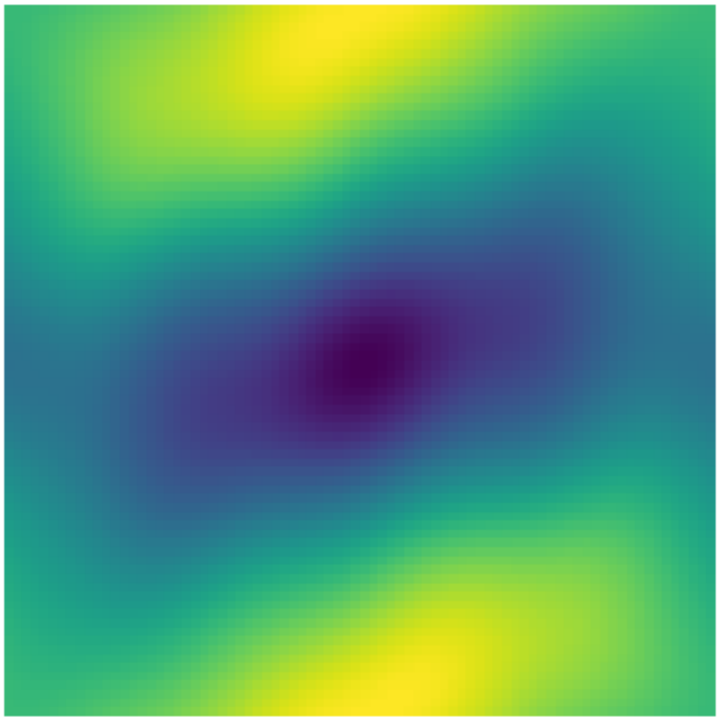


```
coords, data_list = do_sample(25)
```

(continues on next page)

(continued from previous page)

```
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

CHAPTER

FOUR

# GRADIENT CALCULATIONS: FINITE DIFFERENCES

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

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

## 4.1 Gradient's Background

A gradient is vector where the direction is towards greatest rate of change when looking at a scalar function and the magnitude is the greatest 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. The gradient is vector where each pixel has its own component in the 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.

If the function is smooth then 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!

## 4.2 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 $\nabla f$.

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 $\nabla f = \frac{f(x+\frac{h}{2})-f(x-\frac{h}{2})}{h}$

### 4.2.1 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.

```python
def gradient_finite_differences(data_list: list[data], coords, image, mode = 1, FOV =␣
↪100*u.uas.to(u.rad)):
    """
    Calculates a gradient based on finite differences
    Args:
        data_list is a list of data objects
        coords is a list of u,v coordinates that we obtained from our data
        image is a 80x80 pixel image that represents our reconstructed image
        mode is the type of difference used: 0 For central, -1 for backward, 1 for␣
↪forward
        FOV is the Field of view from the telescopes. For the EHT data, our FOV is␣
↪100 micro ascs.
    Returns:
        the gradient of the loss function
    """
    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 = loss(image_copy, data_list, coords, FOV=FOV)
                image_copy[row,col] -= 1e-6
                lower_diff = loss(image_copy, data_list, coords, FOV=FOV)
                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 = loss(image, data_list, coords, FOV=FOV)
        for row in range(len(image)):
            for col in range(len(image[row])):
                image_copy[row,col] -= 1e-8
                lower_diff = loss(image_copy, data_list, coords, FOV=FOV)
                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 = loss(image, data_list, coords, FOV=FOV)
        for row in range(len(image)):
            for col in range(len(image[row])):
                image_copy[row,col] += 1e-8
                upper_diff = loss(image_copy, data_list, coords, FOV=FOV)
                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
```

Final Note: Why do we use 1e-6 and 1e-8.

When we want to estimate our loss function using the method below.

Consider the loss function f and we want to estimate it at point x as a finite difference. Additionally, let g be the numerical computation of f. Then we consider the analysis below

$$f'_h = max_{\Delta x}\frac{|\frac{g(x+\delta x+h)-g(x+\delta x)}{h} - f'(x)|}{|f'(x)|}|\frac{x}{\Delta x}|$$

$$\sim \frac{|g(x+h)-g(x)-f'(x)h|}{\epsilon_{machine}h|f'(x)|}$$

$$\sim \frac{\epsilon_{machine}f + (\frac{fh^2}{L^2})}{\frac{\epsilon_{machine}hf}{L}}$$

$$= \frac{L}{h} + \frac{h}{\epsilon_{machine}L}$$

Here, L is a characteristic scale of x and $\epsilon_{machine}$ is the error of the machine. By minimizing $f'_h$ with respect to h, we find that $h \sim \sqrt{\epsilon_{machine}}L$. With $L \sim 1$, $h \sim \sqrt{\epsilon_{machine}}$.

Why don't we use other values of h?

We have to balance between two types of errors: The first being the error due to numerical errors within g. As h becomes two small, the error becomes bigger. The other error is from the higher order derivatives of the function f. As h grows larger, the higher order derivatives start to dominate the approximation.

How does this connect to 1e-6 and 1e-8?

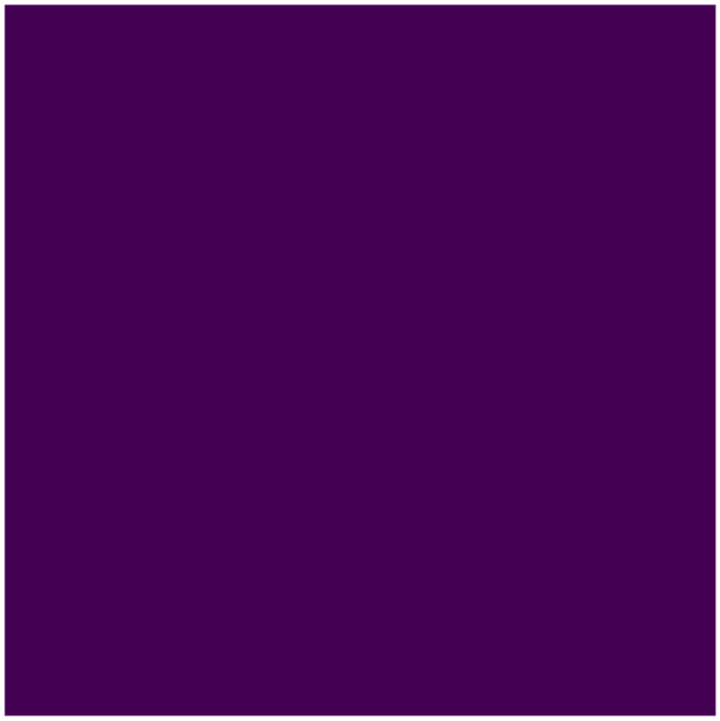
Floats in Python use IEEE Double Precision format which gives allows for it to compare up to $10^{-16}$ precision between two values. In forward and backward differences, the $h$ value uses the square root of $10^{-16}$ which is $10^{-8}$. In central differences, the $h$ value uses a cubic root which means it uses roughly $10^{-6}$

### 4.2.2 Demo Walkthrough

Start with an empty or blank image

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

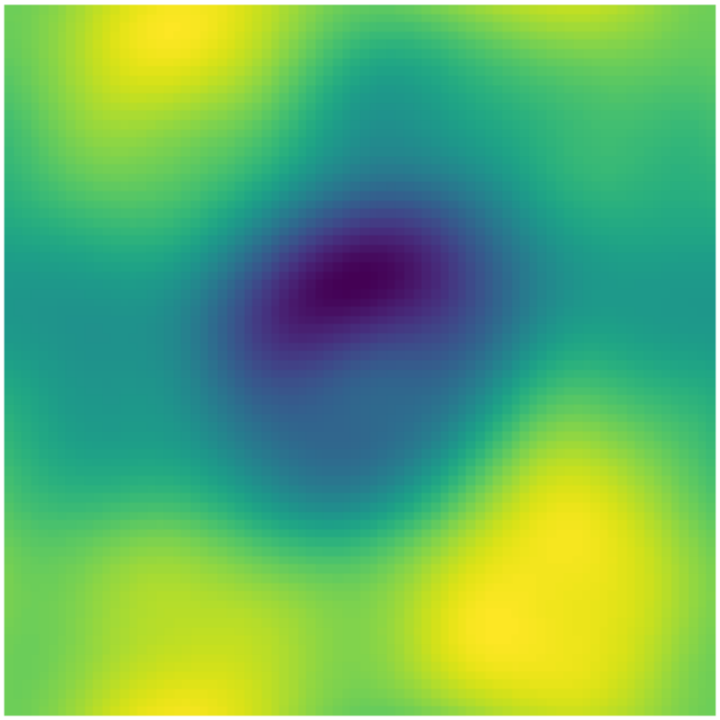
```
(-0.5, 79.5, 79.5, -0.5)
```



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, FOV = 1)
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).

## 4.3 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)) <= 0.0000001`).

The choosing of step size (the scaling of the gradient) is important because if not chosen properly, it will lead to this pattern called zig-zagging. This would then cause the program to reach the minimum slower than it would have by going on a straight path.

```
def gradient_descent(image, data_list, coords, coeffs = None, FOV = 100*u.uas.to(u.
↪rad), stopper = None, dirty = False):
    """
    Performs gradient descent to reconstruct the image
    Args:
```

(continues on next page)

(continued from previous page)

```
        image is a 80x80 pixel image that represents our reconstructed image
        data_list is a list of data objects
        coords is a list of u,v coordinates that we obtained from our data
        coeffs is the precomputed coefficients
        FOV is the Field of view from the telescopes. For the EHT data, our FOV is␣
↪100 micro ascs.
        stopper allows the descent to stop at 20 iterations
        dirty changes the gradient mode to dirty kernel
    Returns:
        the reconstructed image
    """
    image_copy = np.copy(image, subok=True) # Uses copy of the image due to lists␣
↪being mutable in python
    i = 0
    grad = None
    losses_arr = []
    # Can also use max here, min just makes it finish quicker
    while grad is None or np.min(np.abs(grad)) > 0.00001:
        t = 10000000 # Initial Step size which resets each iteration
        prev_loss = loss(image_copy, data_list, coords, FOV=FOV)

        if dirty:
            grad = dirty_gradient(data_list, coords, coeffs, image_copy)
        else:
            grad = gradient_finite_differences(data_list, coords, image_copy, FOV=FOV)

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

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

        image_copy -= t * 2 * grad.real # Multiply by 2 to undo last divide in the␣
↪while loop
        i += 1
        if stopper != None:
            if i == stopper: # Hard stop here for notebook purposes
                break
        losses_arr.append(np.log(new_loss))

    time_step = [i for i in range(len(losses_arr))]
    plt.figure()
    plt.plot(time_step, losses_arr)
    plt.title("ln(loss) vs time")
    plt.ylabel("loss")
    plt.xlabel("time t")

    plt.show()
    return image_copy
```

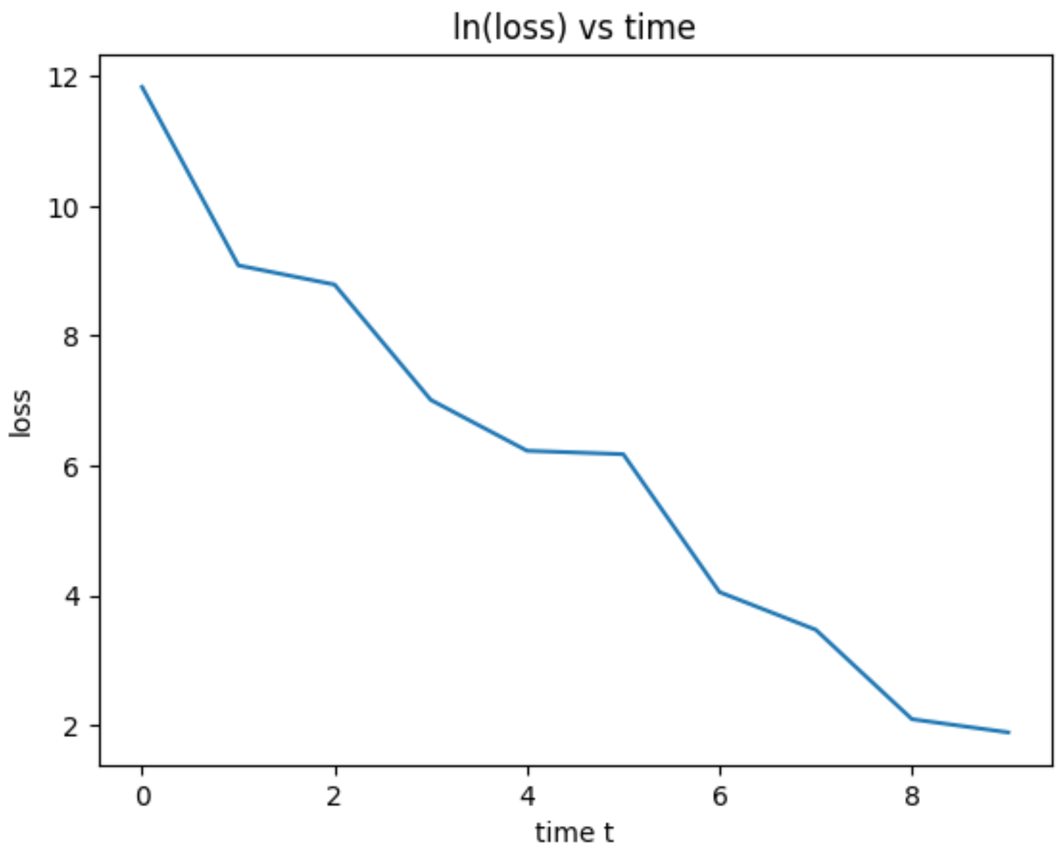
Here is what the gradient descent's output looks like after just 15 iterations. The code should be ran using

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

You will notices that the loss decreases very rapidly towards zero

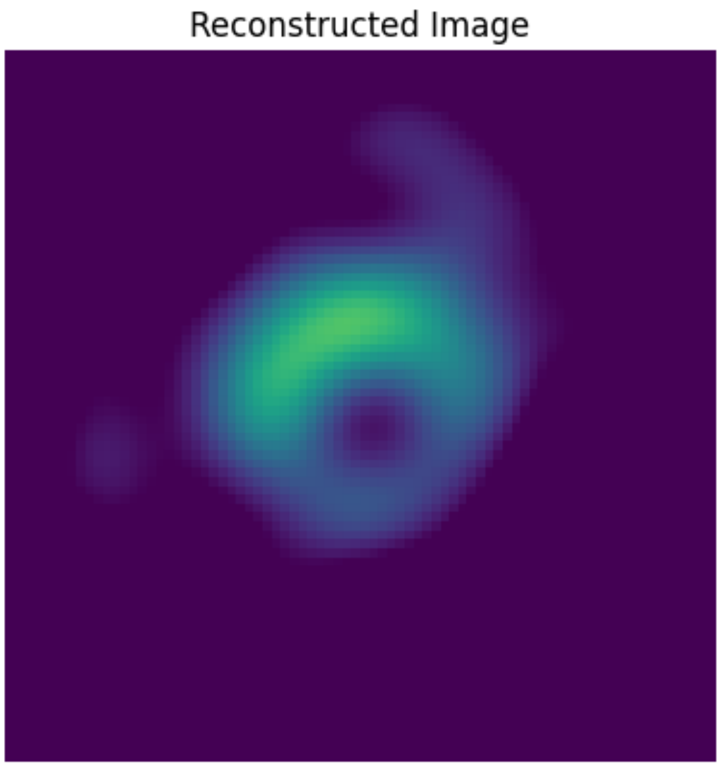
```
coords, data_list = do_sample(50)
reconstructed_img = gradient_descent(emp, data_list, coords, FOV = 1, stopper = 12)
```

```
/var/folders/_b/trlmhkgj5xq968yccj4vtg1c0000gn/T/ipykernel_83776/4118681095.py:42:
↪RuntimeWarning: invalid value encountered in log
  losses_arr.append(np.log(new_loss))
```



```
plt.figure()
plt.imshow(reconstructed_img, vmin=0, vmax=np.max(sample))
plt.axis('off')
plt.title("Reconstructed Image")
```

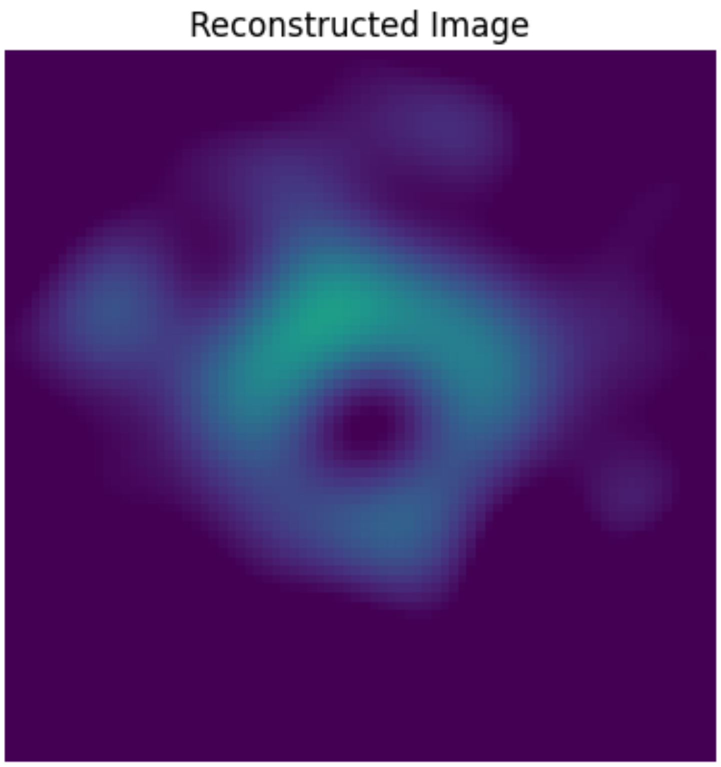
```
Text(0.5, 1.0, 'Reconstructed Image')
```



After another 20 iterations, you should see this:

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

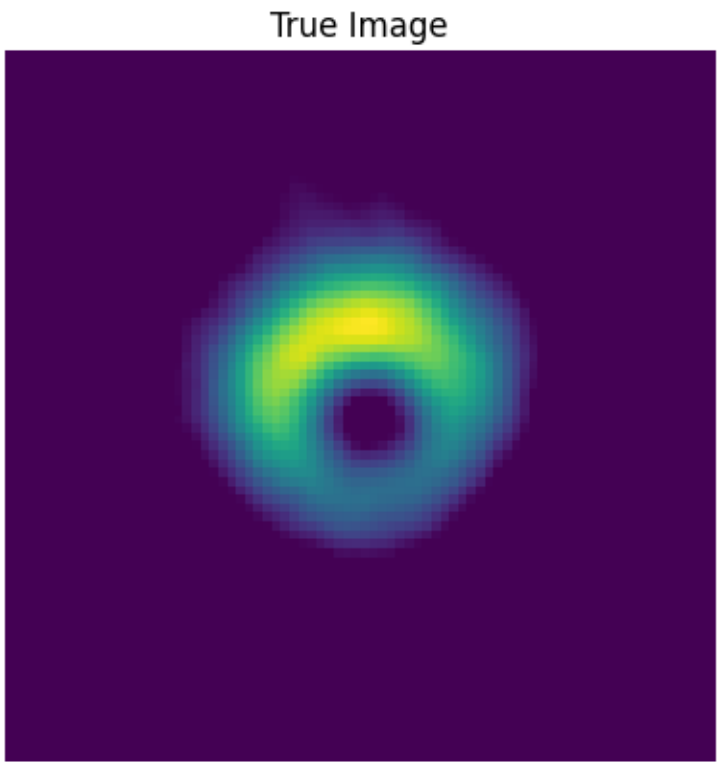
```
Text(0.5, 1.0, 'Reconstructed Image')
```



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')
plt.title("True Image")
```

```
Text(0.5, 1.0, 'True Image')
```



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

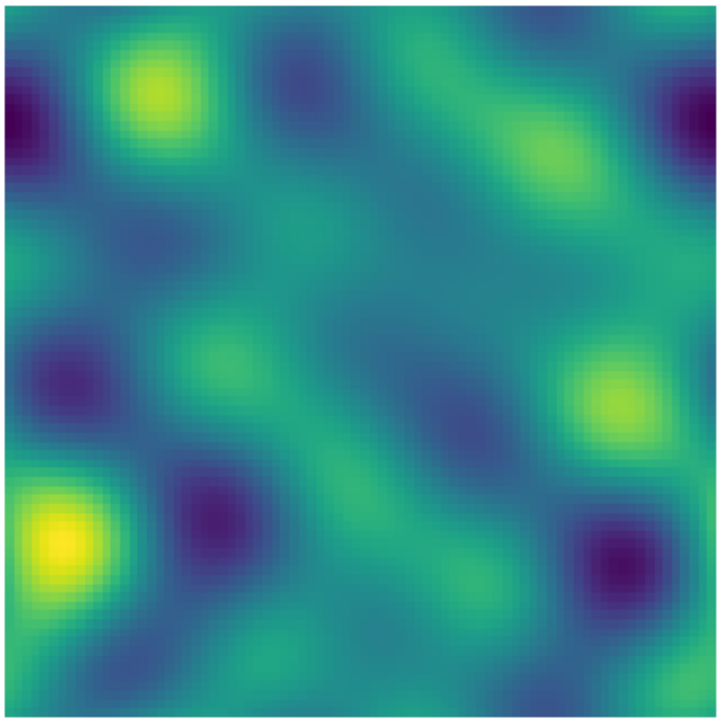
### 4.3.1 Running gradient on Real EHT Data

Here is code to run this:

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

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

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



CHAPTER

FIVE

# GRADIENT CALCULATIONS: "DIRTYING" THE IMAGE

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

## 5.1 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.

First we start with the Gradient calculation. Consider the following loss term

$$L_{data}(I) = \sum_a \frac{|I(u_a, v_a) - O_a|^2}{\sigma_a^2}$$

Here a represents which data point, O_a represents the data point itself, i and j are pixel coordinates, $I_{ij}$ is the image. By doing this, we get the gradient to be:

$$G_{ij} = \frac{\partial L_{data}}{\partial I_{ij}} = \sum_a \frac{1}{\sigma_a^2}(\frac{\partial I(u_a, v_a)}{\partial I_{ij}}(I^*(u_a, v_a) - O_a^*) + (I(u_a, v_a) - O_a)\frac{\partial I^*(u_a, v_a)}{\partial I_{ij}})$$

Ideally we can compute the partial derivates as

$$\frac{\partial I(u_a, v_a)}{\partial I_{ij}} = exp(\frac{2\pi i}{N}(iu_a + jv_a)), \frac{\partial I^*(u_a, v_a)}{\partial I_{ij}} = exp(-\frac{2\pi i}{N}(iu_a + jv_a))$$

N is the size of the image in pixels

We can further define things like so:

We introduce the so-called "dirty" image from observations as

$$DO_{ij} = \sum_a \frac{1}{\sigma_a^2}\frac{\partial I^*(u_a, v_a)}{\partial I_{ij}}O_a$$

Here $I^*(u_a, v_a)$ is the fourier domain of the image at an observed data point.

We also define the "dirtying" of the image I as

$$DI_{ij} = \sum_a \frac{1}{\sigma_a^2}\frac{\partial I^*(u_a, v_a)}{\partial I_{ij}}I(u_a, v_a)$$

This then gives us that

$$G_{ij} = 2Re(DI_{ij} - DO_{ij})$$

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

Here we preprocess and compute $\frac{\partial I(u_a,v_a)}{\partial I_{ij}}, \frac{\partial I^*(u_a,v_a)}{\partial I_{ij}}$

```
def preprocess_gradient(data_list, coords, image):
    """
    Precomputed the coefficients decribed in dirty gradient
    Args:
        data_list is a list of data objects
        coords is a list of u,v coordinates that we obtained from our data
        image is a 80x80 pixel image that represents our reconstructed image
    Returns:
        a 4d list of coefficients
    """
    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)
```

Now, we can use the formula above to compute the gradient of the image directly

```
def dirty_gradient(data_list: list[data], coords, coeffs, image, FOV = 100*u.uas.to(u.
↪rad)):
    """
    Calculates a gradient based on dirting the image
    Args:
        data_list is a list of data objects
        coords is a list of u,v coordinates that we obtained from our data
        coeffs is the precomputed coefficients
        image is a 80x80 pixel image that represents our reconstructed image
        FOV is the Field of view from the telescopes. For the EHT data, our FOV is␣
↪100 micro ascs.
    Returns:
        the gradient of the loss function
    """
    gradient_arr = np.empty(np.shape(image)) # Because we are in real space
    vis_images = interpolate(image, coords, FOV)
    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].vis_data
                vis_image = vis_images[i] # Ask about this on Wednesday
```

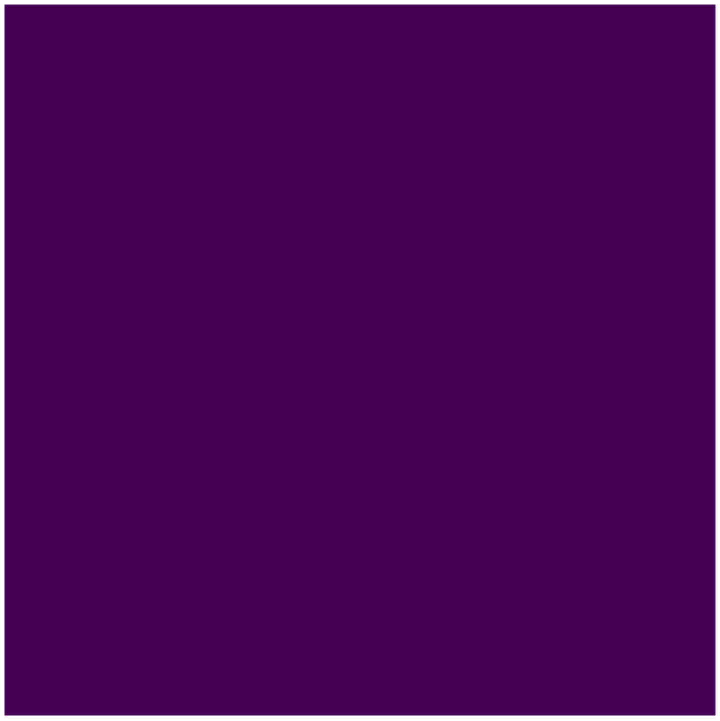
(continues on next page)

(continued from previous page)

```
                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)/(data_list[i].sigma ** 2)
            gradient_arr[row,col] = gradient_sum.real
    return gradient_arr


# Why is it made of fourier harmonics
# Setting Visibilities of 1 for the purposes of this work
```

```
x = dirty_gradient(data_list, coords, coeffs, sample, FOV = 1)
plt.figure()
plt.imshow(x)
plt.axis('off')
```

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



Below is the gradient descent routine done in the fine gradient code

```
def gradient_descent(image, data_list, coords, coeffs = None, FOV = 100*u.uas.to(u.
↪rad), stopper = None, dirty = False):
    """
    Performs gradient descent to reconstruct the image
    Args:
        image is a 80x80 pixel image that represents our reconstructed image
        data_list is a list of data objects
        coords is a list of u,v coordinates that we obtained from our data
```

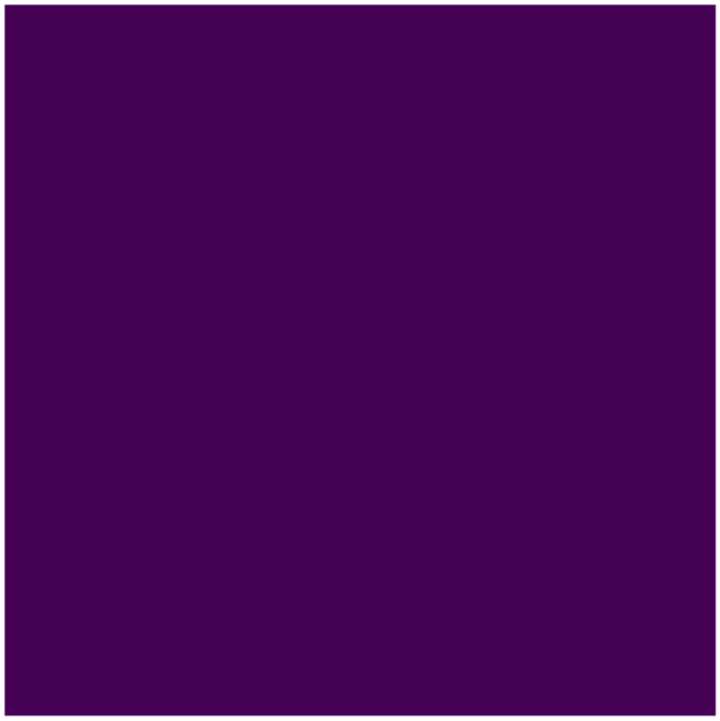
(continues on next page)

(continued from previous page)

```
        coeffs is the precomputed coefficients
        FOV is the Field of view from the telescopes. For the EHT data, our FOV is␣
↪100 micro ascs.
        stopper allows the descent to stop at 20 iterations
        dirty changes the gradient mode to dirty kernel
    Returns:
        the reconstructed image
    """
    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)) > 0.00001:
        t = 10000000 # Initial Step size which resets each iteration
        prev_loss = loss(image_copy, data_list, coords, FOV=FOV)

        if dirty:
            grad = dirty_gradient(data_list, coords, coeffs, image_copy)
        else:
            grad = gradient_finite_differences(data_list, coords, image_copy, FOV=FOV)

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

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

        image_copy -= t * 2 * grad.real # Multiply by 2 to undo last divide in the␣
↪while loop
        i += 1
        if stopper != None:
            if i == stopper: # Hard stop here for notebook purposes
                return image_copy
        print("loss:",new_loss)
    return image_copy
```

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

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



### 5.1.1 Continuing Further

Now lets say that DFT (Discrete fourier transform) is defined as

$$I(u,v) = \frac{1}{N}\sum_{i,j=0}^{N-1} exp(-\frac{2\pi i}{N}(iu+jv))I_{ij}$$

$$I_{ij} = \frac{1}{N}\sum_{i,j=0}^{N-1} exp(\frac{2\pi i}{N}(iu+jv))I(u,v)$$

The interpolation scheme computes $I(u_a, v_a)$ as $I(u_a, v_a) = \sum_{u,v=0}^{N-1} W(u_a, v_a|u,v)I(u,v)$ where $W(u_a, v_a|u,v)$ stands for the weights of $u_a, v_a$ at u and v. The interpolation scheme will compute with only a few weights being non-zero for each a (assuming the interpolation is done in a local way). Putting this expression into $I(u,v)$ we get

$$\frac{\partial I(u_a, v_a)}{\partial I_{ij}} = \frac{1}{N}\sum_{u,v=0}^{N-1} W(u_a, v_a|u,v)exp(-\frac{2\pi i}{N}(iu+jv))$$

We this, we see that the partial derivative $\frac{\partial I(u_a,v_a)}{\partial I_{ij}}$ is the interpolation of the Fourier transform kernel.

Now, we can rewrite our formula for the Dirty Image.

$$DI(u,v) = \frac{1}{N}\sum_{i,j} exp(-\frac{2\pi i}{N}(iu+jv))DI_{ij}$$

$$= \frac{1}{N}\sum_{i,j} exp(-\frac{2\pi i}{N}(iu+jv))\sum_{a}\frac{1}{\sigma_a^2}\frac{\partial I^*(u_a, v_a)}{\partial I_{ij}}I(u_a, v_a)$$

$$= \frac{1}{N}\sum_{i,j} exp(-\frac{2\pi i}{N}(iu+jv))\sum_{a}\frac{1}{\sigma_a^2}\frac{1}{N}\sum_{u',v'=0}^{N-1} W(u_a, v_a|u',v')exp(-\frac{2\pi i}{N}(iu'+jv'))I(u_a, v_a)$$

$$= \sum_a \frac{1}{\sigma_a^2} W(u_a, v_a | u, v) I(u_a, v_a)$$

$$= \sum_a \frac{1}{\sigma_a^2} W(u_a, v_a | u, v) \sum_{u', v'=0}^{N-1} W(u_a, v_a | u', v') I(u', v')$$

Now lets finally introduce the "dirty kernel" DK.

$$DK(u, v | u', v') = \sum_a \frac{1}{\sigma_a^2} W(u_a, v_a | u, v) W(u_a, v_a | u', v')$$

This transforms the dirty image equation to be

$$DI(u, v) = \sum_{u', v'=0}^{N-1} DK(u, v | u', v') I(u', v')$$

Notice here that for each $(u, v)$ there are only a few points $(u', v')$ close by where $DK(u, v|u', v') \neq 0$. Thus for each point $(u, v)$ in the Fourier domain, we can create a small list of $(u', v')$ points with tabulated $DK$ values.

Another important note is that we can dirty the image quickly since the dirty kernel is able to be precomputed. We can do so since it only depends on observational data. The time complexity of going from the image to the fourier domain using DFT is about $O(N^2 ln N)$ whereas the time complexity of going from image to dirty image by using the dirty kernel method is about $O(N^2)$.

The code below was written by Misha Stepanov in order to quickly calculate DI, DO, and DK.

```
import numpy as np
N = 80;  coeff = 4.84813681109536e-10;  Breg = 10000.;  Nreg = 100.

# reading data
list_of_strings = []
with open('./data/SR1_M87_2017_095_hi_hops_netcal_StokesI.csv', 'r') as f:
  for line in f:
    list_of_strings.append(line)
list_of_strings.pop(0)
Na = len(list_of_strings)
print(len(list_of_strings))
UVa, sigma = np.zeros((2*Na, 2)), np.zeros(2*Na)
Oa = np.zeros(2*Na).astype(complex)
for a in range(Na):
  current_string = (list_of_strings[-1]).split(",") # Split on commas
  list_of_strings.pop()
  UVa[a, 0], UVa[a, 1] = float(current_string[3]), float(current_string[4])
  Oa[a] = float(current_string[5])*np.exp(1j*float(current_string[6])*np.pi/180.)
  sigma[a] = float(current_string[7])
  UVa[a + Na], Oa[a + Na], sigma[a + Na] = -UVa[a], np.conj(Oa[a]), sigma[a]
del list_of_strings, current_string

# local cubic interpolation
def cubf1(x, y):                                    #      8   11
  cf1 = (1. + x - x*x)*(1. + 2.*y)*(1. - y)     #   8  2   3  10
  cf2 = (1. + y - y*y)*(1. + 2.*x)*(1. - x)     #   4  0   1   6
  return 0.5*(cf1 + cf2)*(1. - x)*(1. - y)      #      5   7
def cubf2(x, y):  return -0.5*x*(1. - x)*(1. - x)*(1. + 2.*y)*(1. - y)*(1. - y)
IND = np.array([[-2, -2], [-1, -2], [-2, -1], [-1, -1], [-3, -2], [-2, -3], \
  [0, -2], [-1, -3], [-3, -1], [-2, 0], [0, -1], [-1, 0]]).astype(int)
def cubfun12(x, y):  return np.array([cubf1(x, y), cubf1(1. - x, y), \
    cubf1(x, 1. - y), cubf1(1. - x, 1. - y), cubf2(x, y), cubf2(y, x), \
```

(continues on next page)

(continued from previous page)

```
    cubf2(1. - x, y), cubf2(y, 1. - x), cubf2(x, 1. - y), cubf2(1. - y, x), \
    cubf2(1. - x, 1. - y), cubf2(1. - y, 1. - x)])

"""
Calculate DO
Compare G = -2DO
"""

# computing DO, the dirty image from observations
DOuv = np.zeros((N, N)).astype(complex)
for a in range(2*Na):
  m = np.mod(coeff*UVa[a], N);  x, y = m - np.floor(m)
  m1, m2 = np.floor(np.mod(m + np.array([2., 2.]), N)).astype(int)
  C, CUBFUN = Oa[a] / sigma[a]**2, cubfun12(x, y)
  for c in range(12):  DOuv[m1 + IND[c, 0], m2 + IND[c, 1]] += C*CUBFUN[c]
DO = N*np.real(np.fft.ifft2(DOuv))
del DOuv

# computing DK, the dirty kernel
DK49, DK = np.zeros((N, N, 7, 7)), []
for a in range(2*Na):
  m = np.mod(coeff*UVa[a], N);  x, y = m - np.floor(m)
  m1, m2 = np.floor(np.mod(m + np.array([2., 2.]), N)).astype(int)
  C, CUBFUN = 1. / sigma[a]**2, cubfun12(x, y)
  for c in range(12):
    for cp in range(12):
      DK49[m1 + IND[c, 0], m2 + IND[c, 1], 3 + IND[cp, 0] - IND[c, 0], \
          3 + IND[cp, 1] - IND[c, 1]] += CUBFUN[c]*CUBFUN[cp]*C
for u in range(N):
  for v in range(N):
    for du in range(7):
      for dv in range(7):
        if (DK49[u, v, du, dv] != 0.):
          DK.append([u, v, ((u + du) % N) - 3, ((v + dv) % N) - 3, \
            DK49[u, v, du, dv]])
del DK49, IND, CUBFUN

# computing DI, the dirtying of the image I
I12, Iuv = np.zeros((2, N, N)), np.zeros((N, N)).astype(complex)
DI, DIuv = np.zeros((N, N)), np.zeros((N, N)).astype(complex)
def calc_DI():
  global Iuv, DI, DIuv
  Iuv, DIuv = np.fft.fft2(I), np.zeros((N, N)).astype(complex)
  for q in DK:  DIuv[q[0], q[1]] += q[4]*Iuv[q[2], q[3]]
  DI = np.real(np.fft.ifft2(DIuv))
  return

# computing G, the gradient of the loss function
def neighbor(Iat, Inear):  return np.sign(Iat - Inear)
def calc_G():
  global G
  calc_DI()
  G = 2.*(DI - DO)
  for i in range(N):
    for j in range(N):
      if (I[i, j] < 0.):  G[i, j] -= Breg
```

(continues on next page)

(continued from previous page)

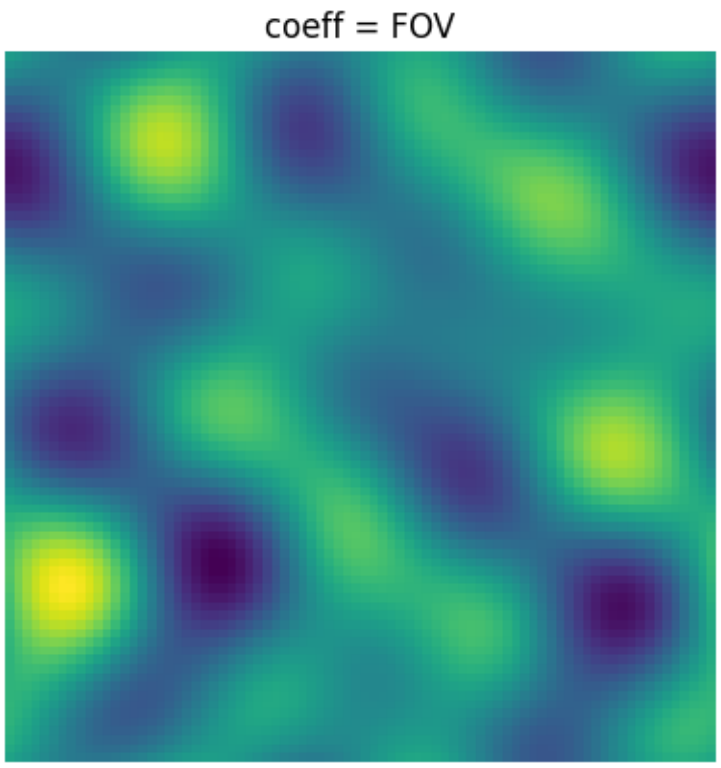
```
        if (I[i, j] > 0.):  G[i, j] += Nreg
    return

I, dt = (0.*DO), 0.00001

calc_DI()
```

```
6458
```

Below is what the gradient looks like after running the dirty kernel code. What is remarkable is its similarity to the gradient calculated in the finite differences notebook in a fraction of the time. I am using a 2021 MacBook Pro with an M1 chip and the dirty kernel code took 2.5 seconds while the finite differences code took 34.4 seconds.

```
plt.figure()
plt.imshow(-2*DO)
plt.axis('off')
plt.title("coeff = FOV")
```

```
Text(0.5, 1.0, 'coeff = FOV')
```



CHAPTER

SIX

# UTILITY APPENDIX

This is the Utility File with all of the functions put in so that all the Juypter Notebooks can use any of these functions

```
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
```

```
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}]"
```

```
def process_data(data_df):
    """
    Processes the data in the dataframe into a coords list and data objects
    Args:
        data_df is a pandas data frame of the data
    Returns:
        a list of coordinates in u,v space
        a list of data objects
```

(continues on next page)

(continued from previous page)

```
    """

    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)']))
        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):
    """
    reads the data from a file into a pandas dataframe
    Args:
        filename is a string that represents a csv file
    Returns:
        a pandas dataframe
    """
    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)
```

```
def loss(image, data_list: list[data], coords, p = 2, reg_weight = 1, FOV = 100*u.uas.
↪to(u.rad)):
    """
    calculates the loss of an image compared to the data given
    Args:
        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
        p is the kind of norm to be used
        reg_weight is the regularizer weight
        FOV is the Field of view from the telescopes. For the EHT data, our FOV is␣
↪100 micro ascs.
    Returns:
        a loss value
    """
    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)
```

```
def do_sample(n):
    """
```

(continues on next page)

(continued from previous page)

```
    Collects n samples from a sample image
    Args:
        n is the number of samples as an integer
    Returns:
        a list of coordinates in u,v space
        a list of data objects
    """
    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(data(coords[i][0], coords[i][1], 0, ft_
↪image[coords[i][0]+40][coords[i][1]+40], 1))
    return coords, data_list
```

```
def calculate_losses():
    """
    Calculates the losses of the image by shifting it around
    Args:
        None
    Returns:
        an array of losses where the index is how much the image is shifted starting␣
↪with -40 to 40
    """
    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] = loss(image_2, data_list, coords, reg_weight=0, FOV=1)

    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
```

```
def interpolate(image, coords, FOV):
    """
    Interpolates the values of each coordinate in coords in the fourier domain of␣
↪image
    Args:
        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.
    Returns:
        The interpolated values at the coordinates based on image
    """

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

(continues on next page)

(continued from previous page)

```
    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
```

```
def calc_regularizer(image: np.array, tsv=False, p=None):
    """
    Calculates the regularizer according to total squared variation
    Args:
        image is a 80x80 pixel image that represents our reconstructed image
        p is the kind of norm to be used
        tsv is the flag for total squared variation
    Returns:
        the regularizer
    """
    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
```

```
def gradient_regularizer(image: np.array):
    """
    Calculates the gradient of the regularizer
    Args:
        image is a 80x80 pixel image that represents our reconstructed image
    Returns:
        the gradient of the regularizer
    """
    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)
```

(continues on next page)

(continued from previous page)

```
    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
```

```
def gradient_finite_differences(data_list: list[data], coords, image, mode = 1, FOV =␣
↪100*u.uas.to(u.rad)):
    """
    Calculates a gradient based on finite differences
    Args:
        data_list is a list of data objects
        coords is a list of u,v coordinates that we obtained from our data
        image is a 80x80 pixel image that represents our reconstructed image
        mode is the type of difference used: 0 For central, -1 for backward, 1 for␣
↪forward
        FOV is the Field of view from the telescopes. For the EHT data, our FOV is␣
↪100 micro ascs.
    Returns:
        the gradient of the loss function
    """
    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 = loss(image_copy, data_list, coords, FOV=FOV)
                image_copy[row,col] -= 1e-6
                lower_diff = loss(image_copy, data_list, coords, FOV=FOV)
                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 = loss(image, data_list, coords, FOV=FOV)
        for row in range(len(image)):
            for col in range(len(image[row])):
                image_copy[row,col] -= 1e-8
                lower_diff = loss(image_copy, data_list, coords, FOV=FOV)
                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 = loss(image, data_list, coords, FOV=FOV)
        for row in range(len(image)):
            for col in range(len(image[row])):
                image_copy[row,col] += 1e-8
                upper_diff = loss(image_copy, data_list, coords, FOV=FOV)
```

(continues on next page)

(continued from previous page)

```
                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
```

```
def gradient_descent(image, data_list, coords, coeffs = None, FOV = 100*u.uas.to(u.
↪rad), stopper = None, dirty = False):
    """
    Performs gradient descent to reconstruct the image
    Args:
        image is a 80x80 pixel image that represents our reconstructed image
        data_list is a list of data objects
        coords is a list of u,v coordinates that we obtained from our data
        coeffs is the precomputed coefficients
        FOV is the Field of view from the telescopes. For the EHT data, our FOV is␣
↪100 micro ascs.
        stopper allows the descent to stop at 20 iterations
        dirty changes the gradient mode to dirty kernel
    Returns:
        the reconstructed image
    """
    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)) > 0.00001:
        t = 10000000 # Initial Step size which resets each iteration
        prev_loss = loss(image_copy, data_list, coords, FOV=FOV)

        if dirty:
            grad = dirty_gradient(data_list, coords, coeffs, image_copy)
        else:
            grad = gradient_finite_differences(data_list, coords, image_copy, FOV=FOV)

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

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

        image_copy -= t * 2 * grad.real # Multiply by 2 to undo last divide in the␣
↪while loop
        i += 1
        if stopper != None:
            if i == stopper: # Hard stop here for notebook purposes
                return image_copy
        print("loss:",new_loss)
    return image_copy
```

```
def preprocess_gradient(data_list, coords, image):
    """
```

(continues on next page)

(continued from previous page)

```
    Precomputed the coefficients decribed in dirty gradient
    Args:
        data_list is a list of data objects
        coords is a list of u,v coordinates that we obtained from our data
        image is a 80x80 pixel image that represents our reconstructed image
    Returns:
        a 4d list of coefficients
    """
    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
```

```
def dirty_gradient(data_list: list[data], coords, coeffs, image, FOV = 100*u.uas.to(u.
↪rad)):
    """
    Calculates a gradient based on dirting the image
    Args:
        data_list is a list of data objects
        coords is a list of u,v coordinates that we obtained from our data
        coeffs is the precomputed coefficients
        image is a 80x80 pixel image that represents our reconstructed image
        FOV is the Field of view from the telescopes. For the EHT data, our FOV is␣
↪100 micro ascs.
    Returns:
        the gradient of the loss function
    """
    gradient_arr = np.empty(np.shape(image)) # Because we are in real space
    vis_images = interpolate(image, coords, FOV)
    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].vis_data
                vis_image = vis_images[i] # Ask about this on Wednesday
                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)/(data_list[i].sigma ** 2)
            gradient_arr[row,col] = gradient_sum.real
    return gradient_arr
```