assignment_8

December 1, 2022

[2]:
```
import numpy as np
import matplotlib.pylab as plt
import numba
import scipy
```

1 Potential at the origin

Let V(1,0) = -log(1). Because of the summation property, $V(1,0) = \frac{1}{4}(V(0,0)+V(1,1)+V(1,-1)+V(2,0))$. Inverting to get $V(0,0) = -4Log(1)+2Log(\sqrt{2})+Log(2) = 2Log(\sqrt{2})+Log(2) = 2Log(2)$

[3]:
```
v0_ref = 2*np.log(2)
v0_ref
```

[3]:
```
1.3862943611198906
```

[4]:
```
rho0_ref = v0_ref + np.log(1)
rho0_ref
```

[4]:
```
1.3862943611198906
```

Rescale by Dividing everything by 1.386... But this way we **DON'T** get correct V[5,0] = -1.05 behaviour...

[5]:
```
v5_ref = np.log(5) / v0_ref
v5_ref
```

[5]:
```
1.160964047443681
```

2 Alternatively, calculate average of neighbours to iteratively generate potential map. This is essentially solving Poisson's equation in 2D numerically.

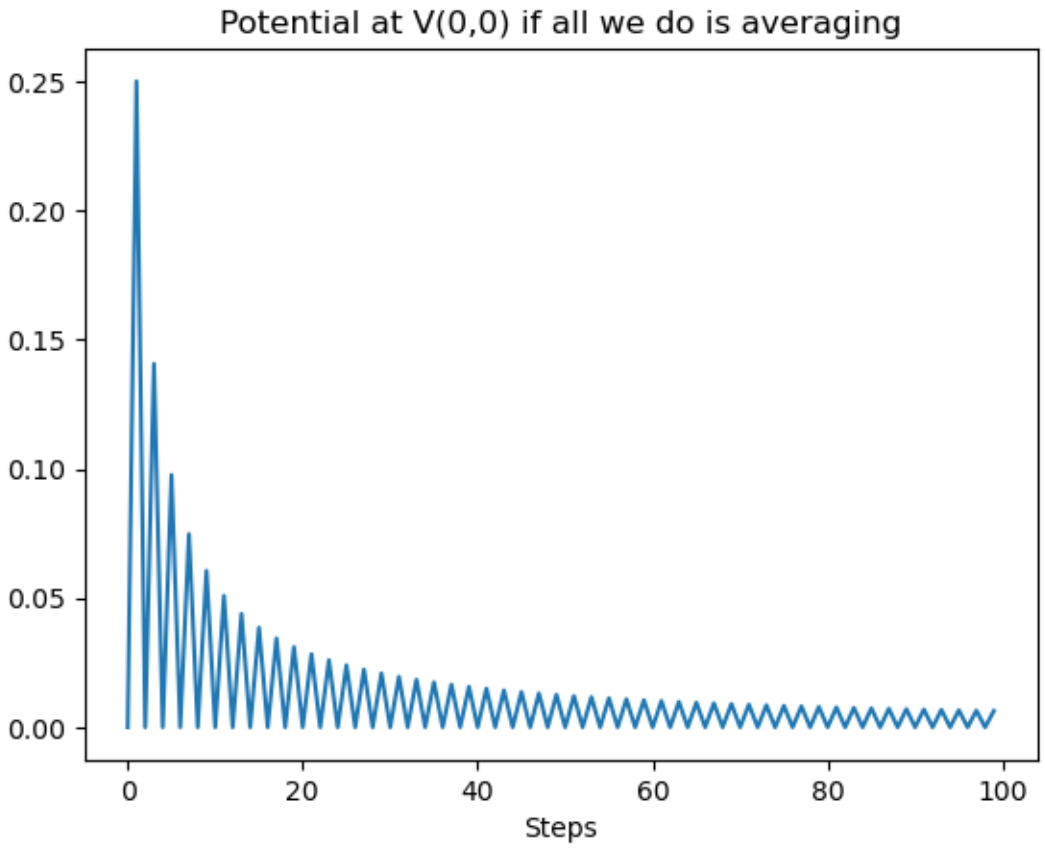
[6]:
```
def laplace(mat):
    m1 = np.roll(mat, (0, -1), axis=(1, 0))
    m2 = np.roll(mat, (0, 1), axis=(1, 0))
    m3 = np.roll(mat, (-1, 0), axis=(1, 0))
```

```
    m4 = np.roll(mat, (1, 0), axis=(1, 0))

    return 1/4*(m1+m2+m3+m4)
```

3 Generate zero matrix and set origin to 1. By symmetry, this would always be 1

[7]:
```
size = 200
m0 = np.zeros([size,size])

m0[size//2, size//2] = 1

steps = 100

v0 = np.zeros(steps)

for i in np.arange(steps):
    m0 = laplace(m0)
    v0[i] = m0[size//2, size//2]

plt.plot(v0)
plt.title("Potential at V(0,0) if all we do is averaging")
plt.xlabel('Steps')
```

[7]:
```
Text(0.5, 0, 'Steps')
```



4 Using the average neighbour method, the potential at the origion get's "Averaged out" to zero. We need to manually add one.

[8]:
```
m0 = np.zeros([size,size])

m0[size//2, size//2] = 1

steps = 12000

v0 = np.zeros(steps)

for i in np.arange(steps):
    m0 = laplace(m0)
    m0[size//2, size//2] +=1 # This forces the charge to be at the origin

offset = 1 - m0[size//2, size//2]
m0 = m0 + offset
```

[9]:
```
green = m0
```

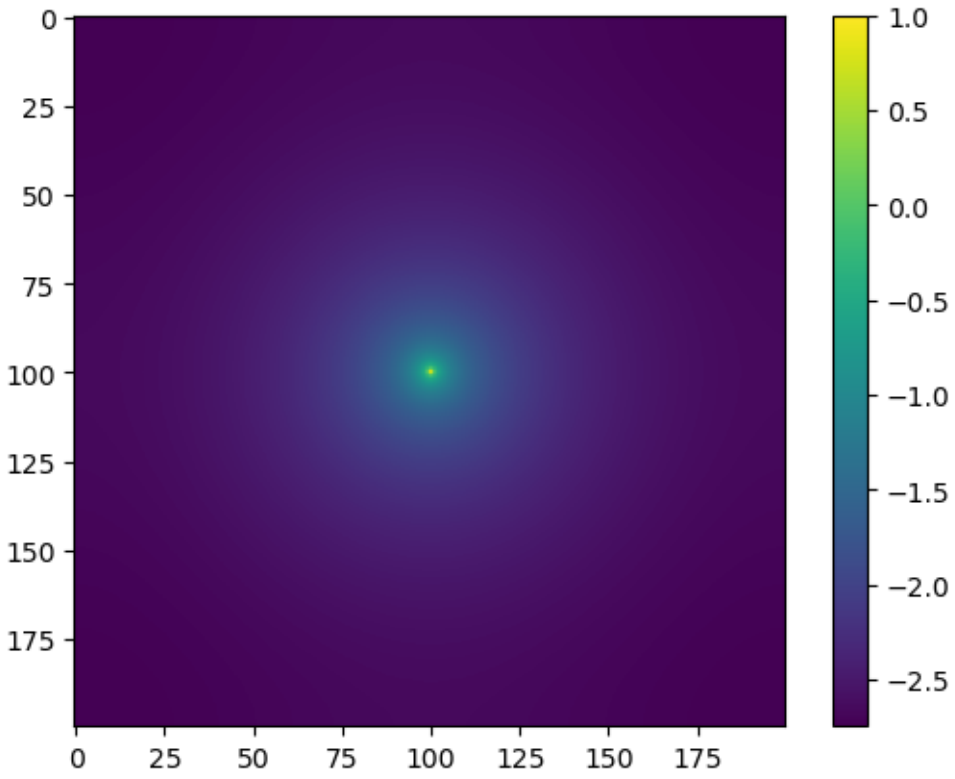
5 Now print ρ_0 and V_0

[10]:
```
rho0 = m0[size//2, size//2] + \
    laplace(m0)[size//2, size//2] # average of neighbours at the origin
v0 = m0[size//2, size//2]
v5 = m0[size//2 +5 , size//2]
print("Voltage and rho at (0,0):\t", v0,"\t and\t", rho0)
print("Voltage at (5,0):\t", v5)
```

```
Voltage and rho at (0,0):        1.0     and     1.0
Voltage at (5,0):        -1.050878727149959
```

[11]:
```
plt.imshow(m0)
plt.colorbar()
```

[11]:
```
<matplotlib.colorbar.Colorbar at 0x1374a2f70>
```



6 Conjugate gradient

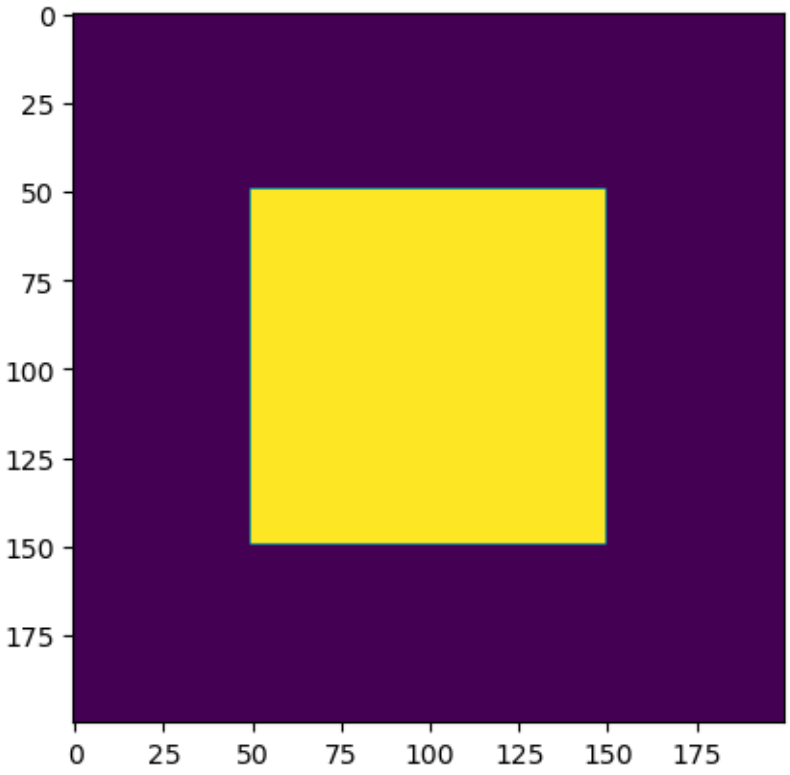
Solving $Ax = b$. Here, b is the voltage map and x is the charge distribution. A is simply b - **Avg.**(b).

The mask sets the condition of our problem

Psudocode: * Generated mask, which also acts as our boundary condition * Only solve RHS inside masked region. * The solver solves for `rho` only inside masked region. But we can extrapolate voltage in all space by padding.

[12]:
```
edge_length = size//2
mask = np.zeros([size, size], dtype = bool)
mask[size//2 - edge_length//2 : size//2 + edge_length//2, size//2 - edge_length/
 ↪/2 : size//2+edge_length//2] =True
plt.imshow(mask)
```

[12]:
```
<matplotlib.image.AxesImage at 0x1375af580>
```



[13]:
```
def conv(rho):
    conv = np.fft.fftshift(np.fft.irfft2(np.fft.rfft2(green) * np.fft.
 ↪rfft2(rho)))
#     conv = scipy.signal.convolve(green, rho, 'same', method = 'fft')
```

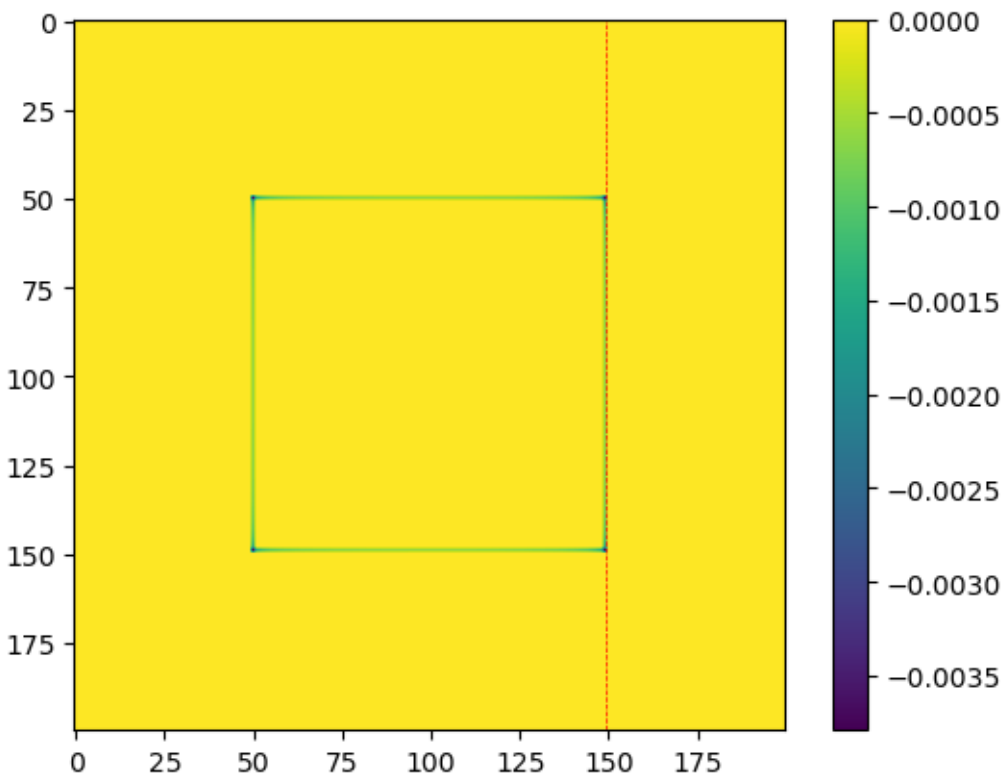
```
#     conv = scipy.signal.convolve2d(green, rho, boundary='wrap', mode='same')
    return conv

def conv_mask(rho, mask):
    masked_rho = np.zeros(mask.shape)
    masked_rho[mask] = rho

    return conv(masked_rho)[mask]


# stolen from Jon's code, but modified.

def conjgrad(A, b, mask, x, niter=5000):
    r = b - A(x, mask)
    p = r.copy()
    rtr = np.sum(r**2)

    for i in range(niter):
        Ap=A(p, mask)
        pAp=np.sum(p*Ap)
        alpha=rtr/pAp
        x=x+alpha*p
        r=r-alpha*Ap
        rtr_new=np.sum(r**2)
        beta=rtr_new/rtr
        p=r+beta*p
        rtr=rtr_new
    return x
```

[14]:
```
rho = conjgrad(conv_mask, b = mask[mask], mask = mask, x = mask[mask])
```

[15]:
```
rho_all = np.zeros(mask.shape)
rho_all[mask] = rho
```

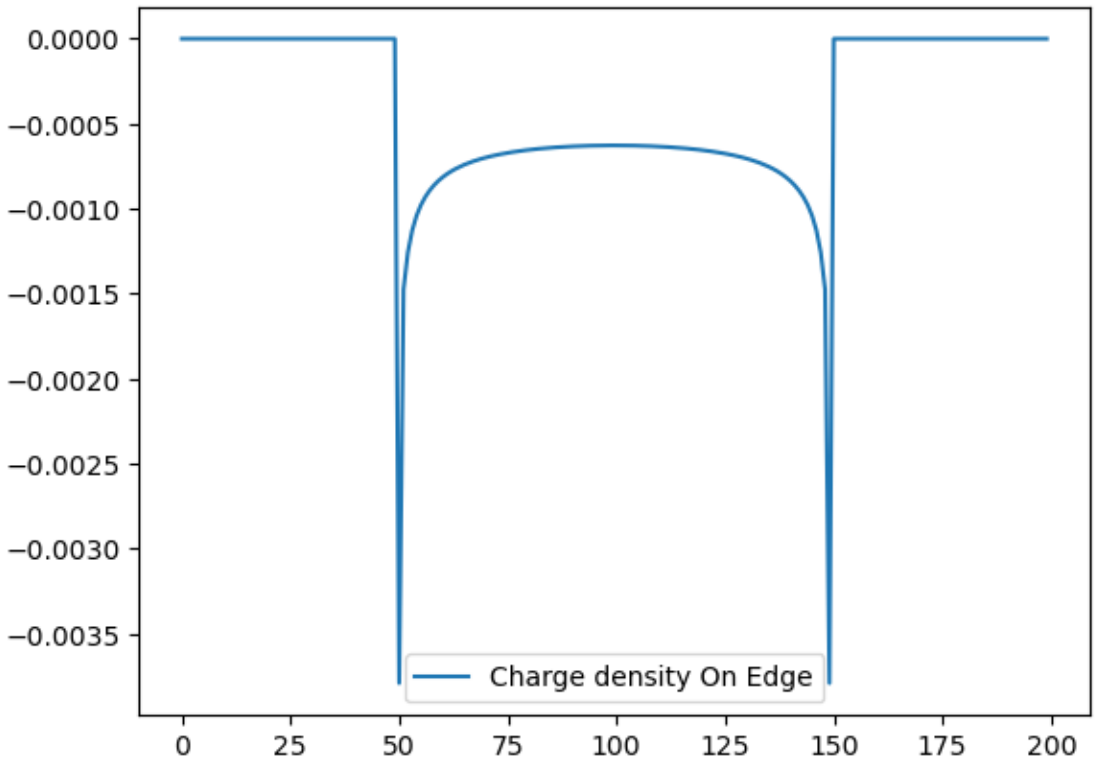
[16]:
```
plt.imshow(rho_all)
plt.colorbar()
plt.axvline(size//2 + edge_length//2 -1,lw = .5, c = 'r', ls = '--')
```

[16]:
```
<matplotlib.lines.Line2D at 0x137647880>
```



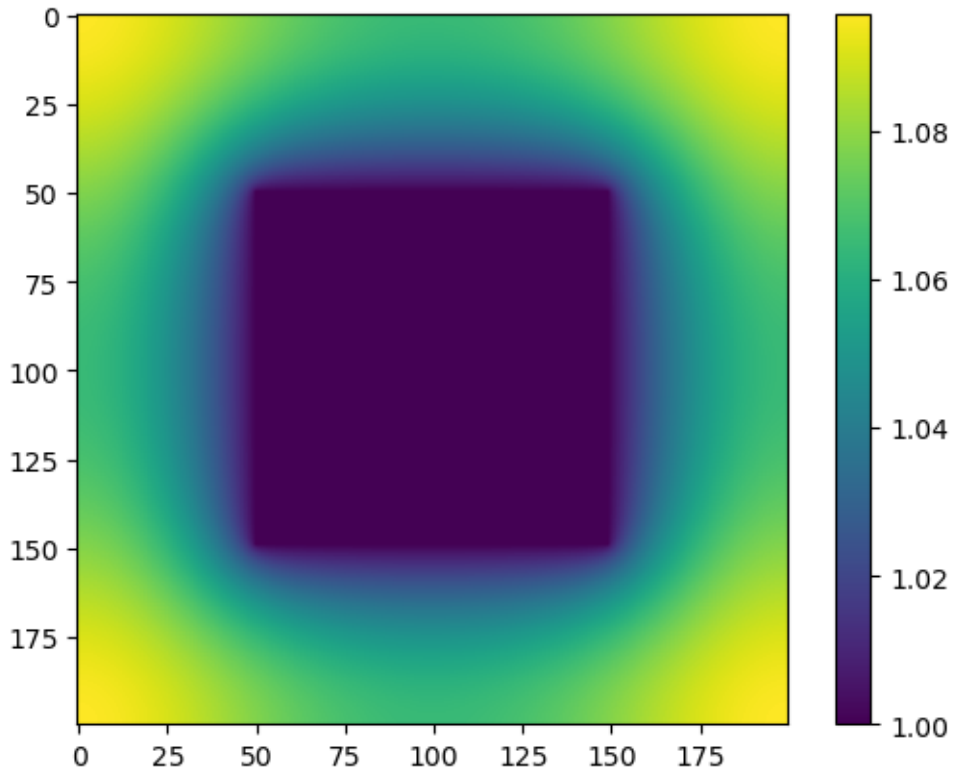
[17]:
```
plt.plot(rho_all[:, size//2 + edge_length//2 -1], label = 'Charge density On␣
↪Edge')
plt.legend()
```

[17]: <matplotlib.legend.Legend at 0x1376f79a0>



[18]:
```
v = conv(rho_all)
plt.imshow(v)
plt.colorbar()
```

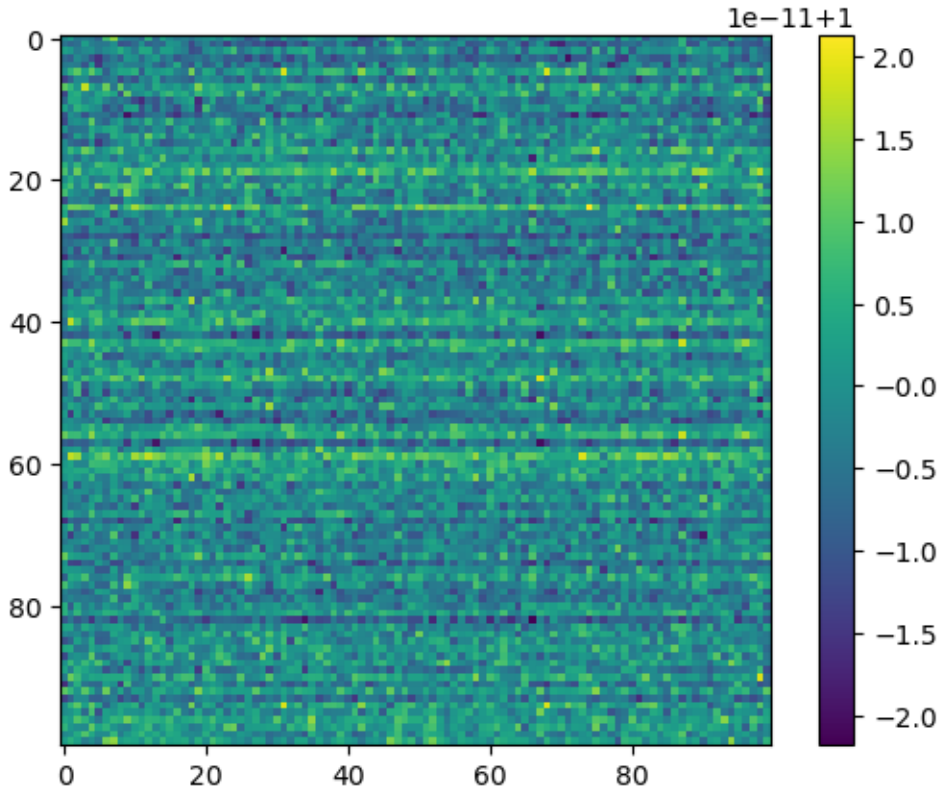
[18]: <matplotlib.colorbar.Colorbar at 0x1377cfd00>



7 Resulting voltage inside the square (100 by 100)

[19]:
```
interior = conv_mask(rho,mask).reshape((100,100))
plt.imshow(interior)
plt.colorbar()
```

[19]: <matplotlib.colorbar.Colorbar at 0x1378b7bb0>



[20]:
```
print("Voltage inside has mean of ", interior.mean(), " and STD of ", interior.
↪std())
```

```
Voltage inside has mean of  0.9999999999990221  and STD of
5.846224762358054e-12
```

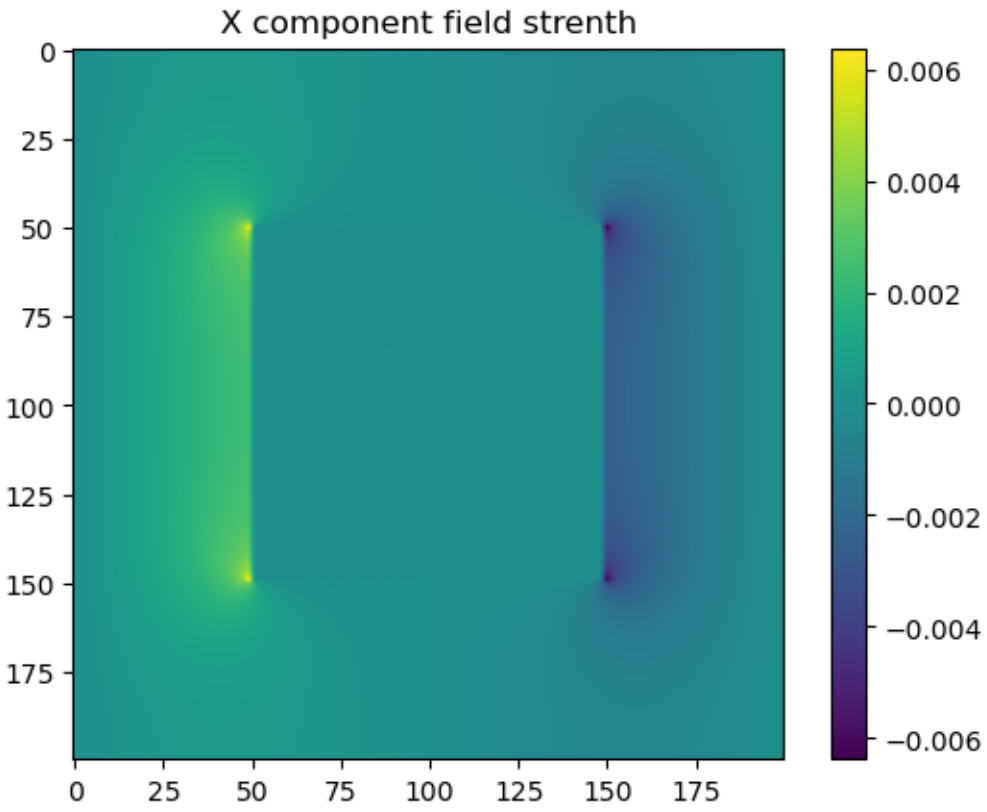
The potential inside the box is almost constant up to $10^{-}12$

8 Field

Use finite difference. Approximate gradients as central difference of neighbours.

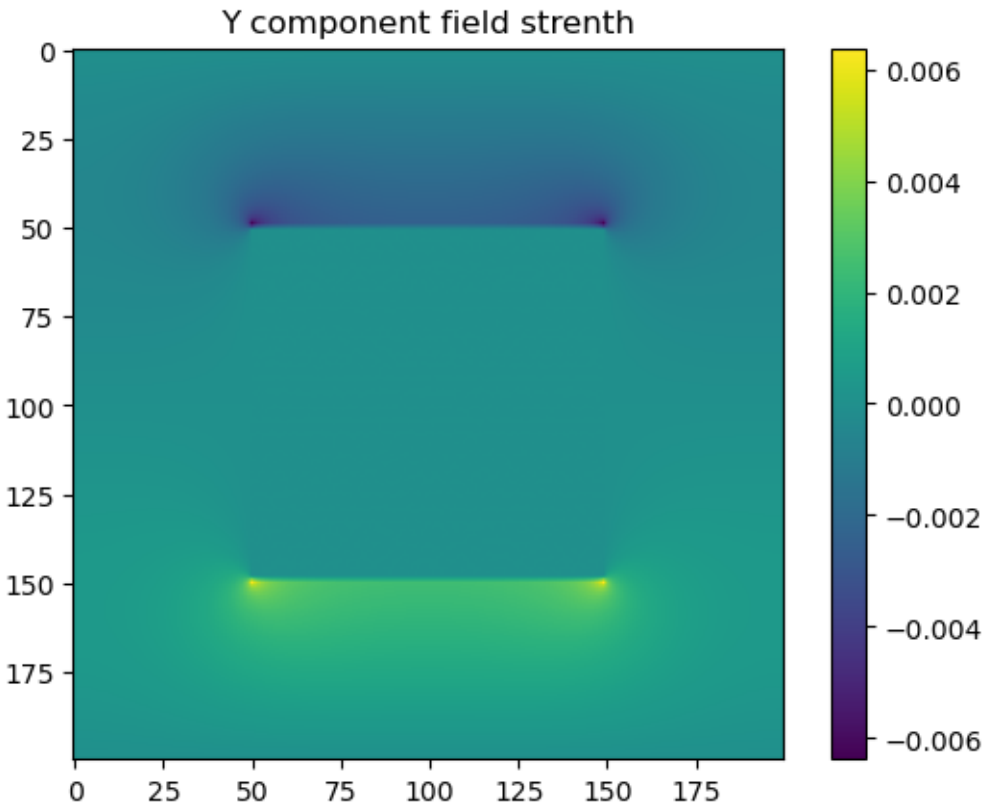
[21]:
```
Ex = (np.roll(v,1,axis=1) - np.roll(v,-1,axis=1))/2
Ey = (np.roll(v,-1,axis=0) - np.roll(v,1,axis=0))/2
plt.imshow(Ex)
plt.colorbar()
plt.title("X component field strenth")
```

[21]:
```
Text(0.5, 1.0, 'X component field strenth')
```



[22]:
```
plt.imshow(Ey)
plt.colorbar()
plt.title("Y component field strenth")
```

[22]: Text(0.5, 1.0, 'Y component field strenth')



Near the vertices of the square, curvature is high in the boundary condition. Based on EM theory we predict that the field is also strong at the points

[ ]: