```
import numpy as np
import matplotlib.pyplot as plt

adj = lambda u : np.conj(u.T)

def fermi(E): #step function: make life easy
    return (1-np.sign(E))/2
```

Model

```
s0 = np.array([[1,0],[0,1]],dtype=complex)
sx = np.array([[0,1],[1,0]],dtype=complex)
sy = np.array([[0,-1j],[1j,0]],dtype=complex)
sz = np.array([[1,0],[0,-1]],dtype=complex)

t = 1
mu = 0
m = 0

h = lambda k, : mu*s0 + t*np.cos(k[0])*sx + t*np.cos(k[1])*sy + m*sz

jx = lambda k : -t*np.sin(k[0])*sx
jy = lambda k : -t*np.sin(k[1])*sy

kstar = np.array([1,1])
h(kstar)
```

```
array([[0.         +0.j         , 0.54030231-0.54030231j],
       [0.54030231+0.54030231j, 0.         +0.j         ]])
```

Naive Implementation (1x)

The paramagnetic optical conductivity is

$$\sigma^{\mu\nu} = \sum_{k,i,j} \frac{f(\epsilon_i) - f(\epsilon_j)}{\epsilon_j - \epsilon_i} \frac{\langle u_i | j^\mu | u_j \rangle \langle u_j | j^\nu | u_i \rangle}{\Omega + i\eta - (\epsilon_j - \epsilon_i)}$$

where the $i = j$ terms vanish because of the Fermi functions (more strictly this is a derivative, but we ignore this contribution here).

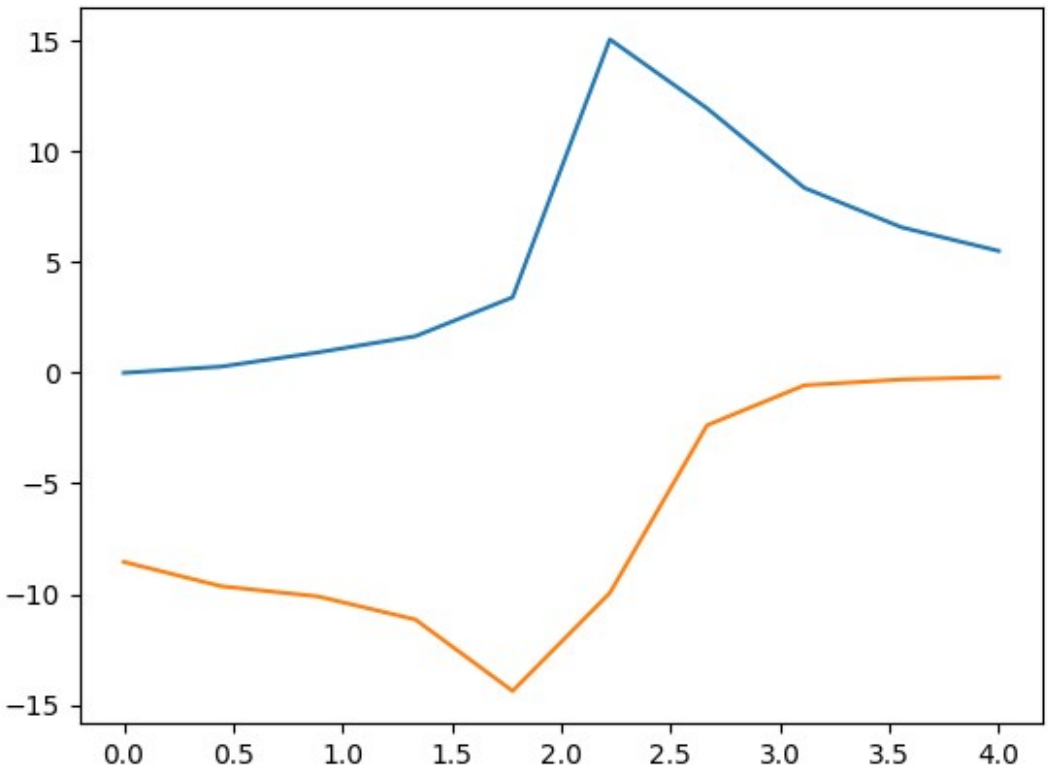
```
def kubo(k,w):
    vals,vecs = np.linalg.eigh(h(k))
    temp = 0
    for i in range(0,2,1):
        for j in range(0,2,1):
            if(i!=j):
                dE = vals[j]-vals[i]
                temp += (fermi(vals[i])-fermi(vals[j])) \
                        * adj(vecs[:,i])@jx(k)@vecs[:,j] *
adj(vecs[:,j])@jx(k)@vecs[:,i] \
```

```
                        / (dE * (w-dE)) \
                        * (2*np.pi)**2/res**2
    return temp

%%time

eta = 0.1
ws = np.linspace(0,4,10) + 1j*eta*np.ones(10)
tot = np.zeros(10,dtype=complex)
res = 100
ks = np.linspace(0,2*np.pi,res)

for w in range(0,10,1):
    W = ws[w]
    for kx in range(0,res,1):
        for ky in range(0,res,1):
            k = np.array([ks[kx],ks[ky]])
            tot[w] += kubo(k,W)
```

```
CPU times: user 2.32 s, sys: 25.2 ms, total: 2.35 s
Wall time: 2.27 s
```

```
plt.plot(np.real(ws),np.real(tot))
plt.plot(np.real(ws),np.imag(tot))
```

```
[<matplotlib.lines.Line2D at 0x10d627590>]
```



Naive Implementation + Vectorize Omega (75x)

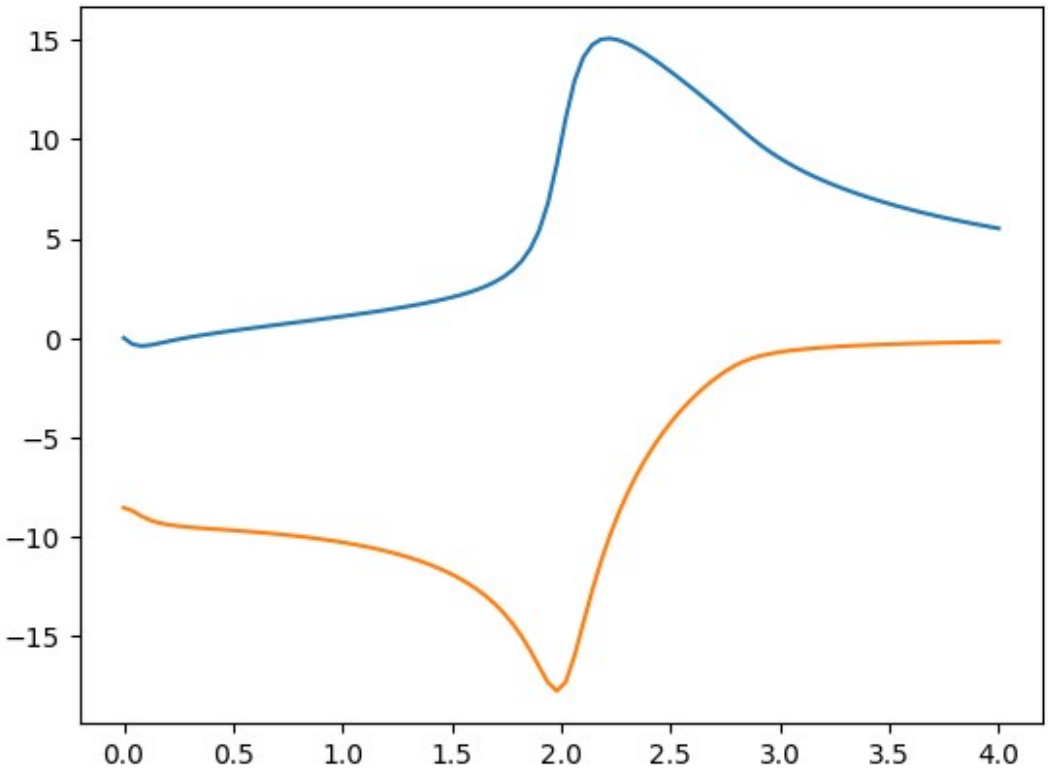
Should improve linearly with number of points (100x) but it doesn't quite do it.

```
def kubo(k,ws):
    vals,vecs = np.linalg.eigh(h(k))
    temp = np.zeros(100,dtype=complex)
    for i in range(0,2,1):
        for j in range(0,2,1):
            if(i!=j):
                dE = (vals[j]-vals[i]) * np.ones(100)
                scalar = (fermi(vals[i])-fermi(vals[j])) \
                        * adj(vecs[:,i])@jx(k)@vecs[:,j] *
adj(vecs[:,j])@jx(k)@vecs[:,i] \
                        * (2*np.pi)**2/res**2
                temp += scalar/(dE * (ws-dE))
    return temp

%%time

eta = 0.1
ws = np.linspace(0,4,100) + 1j*eta*np.ones(100)
tot = np.zeros(100,dtype=complex)
res = 100
ks = np.linspace(0,2*np.pi,res)

for kx in range(0,res,1):
    for ky in range(0,res,1):
        k = np.array([ks[kx],ks[ky]])
        tot += kubo(k,ws)
```

```
CPU times: user 789 ms, sys: 84.7 ms, total: 874 ms
Wall time: 381 ms
```

```
plt.plot(np.real(ws),np.real(tot))
plt.plot(np.real(ws),np.imag(tot))
```

```
[<matplotlib.lines.Line2D at 0x10d750310>]
```



Einsum Implementation

While this is slower for this two band example, einsum does **much** better with more bands.

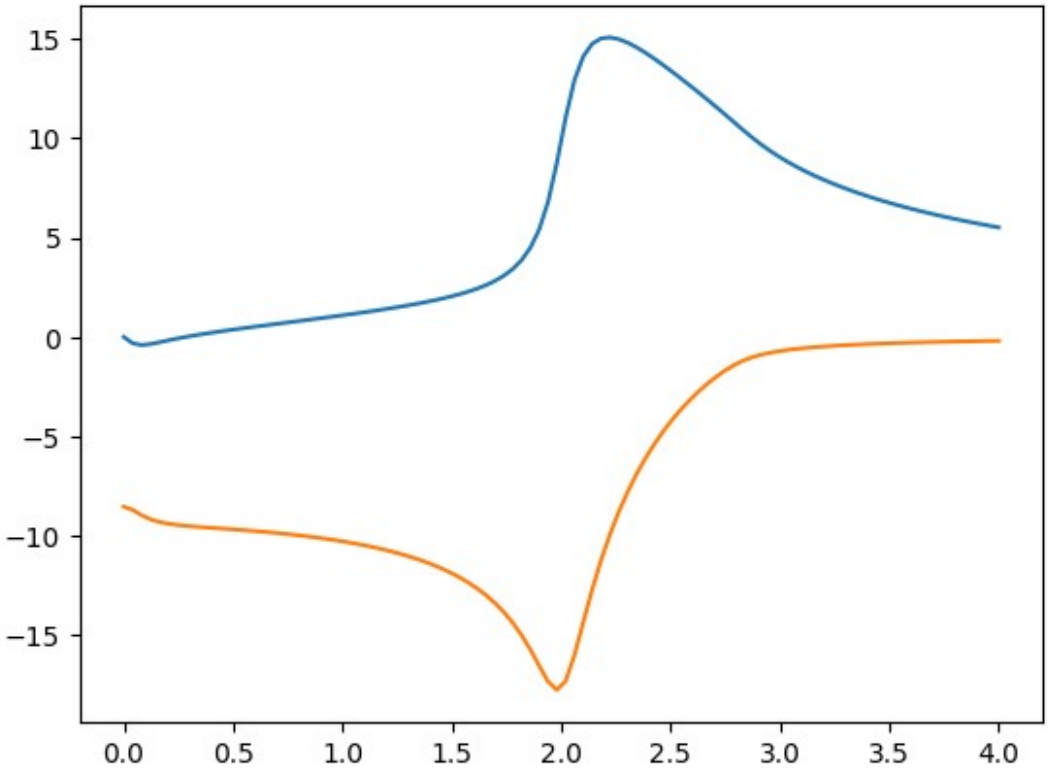
```
def kubo(k,ws):
    vals,vecs = np.linalg.eigh(h(k))
    temp = np.zeros(100,dtype=complex)
    one2 = np.ones(2) #since two bands
    oneW = np.ones(100)
    fermis = np.einsum("i,j->ij",fermi(vals),one2) - np.einsum("i,j-
>ij",one2,fermi(vals))
    matrix1 = np.einsum("qi,qr,rj->ij",np.conj(vecs),jx(k),vecs)
    matrix2 = matrix1 #np.einsum("qi,qr,rj-
>ij",np.conj(vecs),jx(k),vecs)
    denom1 = np.einsum("i,j->ij",vals,one2) - np.einsum("i,j-
>ij",one2,vals+10**-12*one2)
    denom2 = np.einsum("w,i,j->wij",ws,one2,one2) \
            - (np.einsum("w,i,j->wij",oneW,vals,one2) \
             - np.einsum("w,i,j->wij",oneW,one2,vals))
    norm = (2*np.pi)**2/res**2
    temp = norm * np.einsum("ij, ij,ji, ji,wji->w",fermis,
matrix1,matrix2, 1/denom1,1/denom2)
    return temp
```

```
%%time

eta = 0.1
ws = np.linspace(0,4,100) + 1j*eta*np.ones(100)
tot = np.zeros(100,dtype=complex)
res = 100
ks = np.linspace(0,2*np.pi,res)

for kx in range(0,res,1):
    for ky in range(0,res,1):
        k = np.array([ks[kx],ks[ky]])
        tot += kubo(k,ws)
```

```
CPU times: user 950 ms, sys: 53.8 ms, total: 1 s
Wall time: 542 ms
```

```
plt.plot(np.real(ws),np.real(tot))
plt.plot(np.real(ws),np.imag(tot))
```

```
[<matplotlib.lines.Line2D at 0x10db61390>]
```



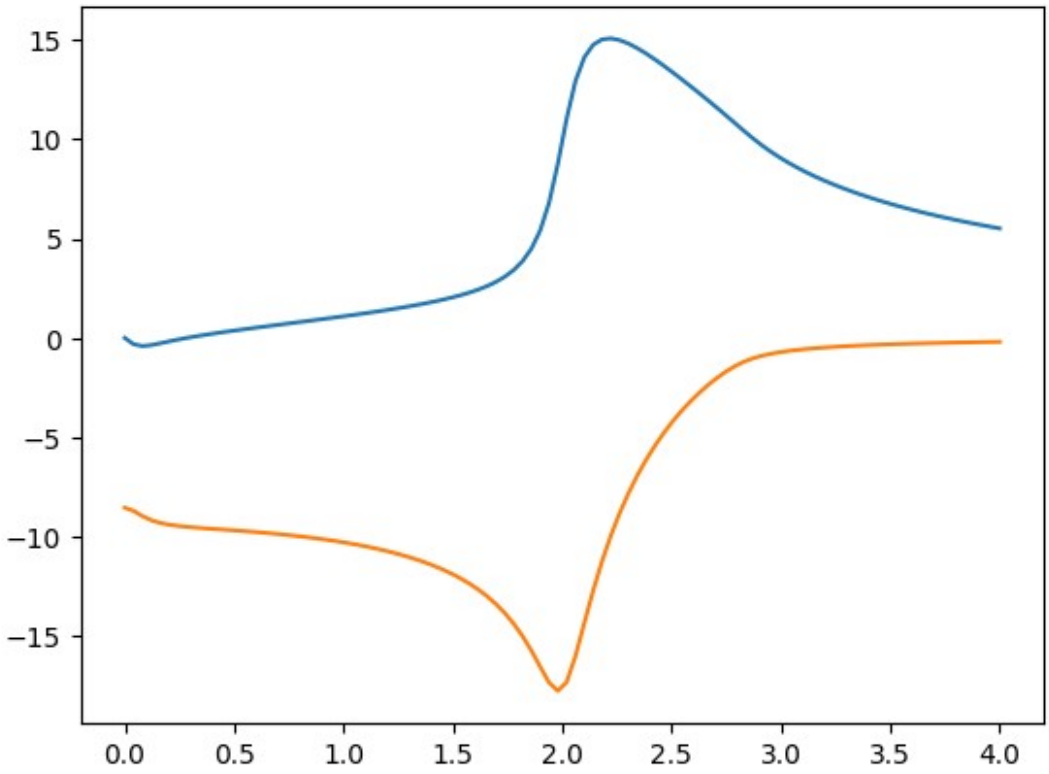
Einsum with broadcasting

```
def kubo(k,ws):
    vals,vecs = np.linalg.eigh(h(k))
    fermis = fermi(vals)[:,None] - fermi(vals)[None,:]
    matrix1 = np.einsum("qi,qr,rj->ij",np.conj(vecs),jx(k),vecs)
    matrix2 = matrix1 #np.einsum("qi,qr,rj-
>ij",np.conj(vecs),jx(k),vecs)
    denom1 = vals[:,None] - vals[None,:] + 10**-12
    denom2 = ws[:,None,None] - vals[None,:,None] + vals[None,None,:]
    norm = (2*np.pi)**2/res**2
    temp = norm * np.einsum("ij, ij,ji, ji,wji->w",fermis,
matrix1,matrix2, 1/denom1,1/denom2)
    return temp

%%time

eta = 0.1
ws = np.linspace(0,4,100) + 1j*eta*np.ones(100)
tot = np.zeros(100,dtype=complex)
res = 100
ks = np.linspace(0,2*np.pi,res)

for kx in range(0,res,1):
    for ky in range(0,res,1):
        k = np.array([ks[kx],ks[ky]])
        tot += kubo(k,ws)
```

```
CPU times: user 738 ms, sys: 9.86 ms, total: 747 ms
Wall time: 317 ms
```

```
plt.plot(np.real(ws),np.real(tot))
plt.plot(np.real(ws),np.imag(tot))
```

```
[<matplotlib.lines.Line2D at 0x10dc43290>]
```



One-line einsum

```
def kubo(k,ws):
    vals,vecs = np.linalg.eigh(h(k))
    temp = (2*np.pi)**2/res**2 * np.einsum("ij, ai,ab,bj, cj,cd,di,
ji,wji->w",fermi(vals)[:,None] - fermi(vals)[None,:],
np.conj(vecs),jx(k),vecs, np.conj(vecs),jx(k),vecs, 1/(vals[:,None] -
vals[None,:] + 10**-12),1/(ws[:,None,None]-vals[None,:,None]
+vals[None,None,:]))
    return temp

%%time

eta = 0.1
ws = np.linspace(0,4,100) + 1j*eta*np.ones(100)
tot = np.zeros(100,dtype=complex)
res = 100
ks = np.linspace(0,2*np.pi,res)

for kx in range(0,res,1):
    for ky in range(0,res,1):
        k = np.array([ks[kx],ks[ky]])
        tot += kubo(k,ws)
```

```
CPU times: user 2.01 s, sys: 82.9 ms, total: 2.09 s
Wall time: 1.66 s
```

```
plt.plot(np.real(ws),np.real(tot))
plt.plot(np.real(ws),np.imag(tot))
```

```
[<matplotlib.lines.Line2D at 0x10dcad250>]
```