solution-hw5

April 12, 2022

[ ]:
```
import matplotlib.pyplot as plt
import numpy as np

# Define Pauli matrices
s0_ = np.array([[1, 0], [0, 1]])
sx_ = np.array([[0, 1], [1, 0]])
sy_ = np.array([[0, -1j], [1j, 0]])
sz_ = np.array([[1, 0], [0, -1]])

# discretization parameters
Nkx = 10  # number of k points along x
Nky = 250  # number of k points along y
Nw = 200  # number of omega points

# model parameters
t = 1
delta = 2*t
mu = -2*t

# point at which to evaluate the surface spectral function
x = 1

# (infinitesimal) broadening parameter. See for youself what happens
# when you change it. What happens when you set eta=0? Did you expect that?
eta = 0.05
```

Python code that uses for loops is slow. *Very slow.* Like a few-hundred times slower than C code.

Thus you should avoid for loops whenever possible. Built-in numpy functions, on the other hand, are very fast.

For this assignment, we will demonstrate a strategy that avoids the use of for loops altogether. We will use *Python Broadcasting*.

This is an extremely powerful tool that allows to write code looking very similar to written math equations. You are strongly encouraged to learn about broadcasting here: https://numpy.org/doc/stable/user/basics.broadcasting.html. Also feel free to ask Rafael.

We start by setting an indexing convention. This will be `(w,kx,ky,i,j)` where `i,j` are the matrix indices of the Hamiltonian. For example, Pauli matrices are 2 x 2 matrices but do not depend on

`w,kx,ky`. Therefore, we create an array of dimensions `(1,1,1,2,2)` for them. An array dimension of size `1` does not need extra RAM and is basically just there for book-keeping purposes as you will see in the following.

We will refer to array dimensions as *axes*. An axis of size 1 is called an *empty axis*. If you ever want to check for the axes of an array `X`, use `np.shape(X)` or `X.shape`.

We can insert this empty axis using `np.newaxis`:

[ ]:
```
ax = np.newaxis
s0 = s0_[ax, ax, ax, :, :]
sx = sx_[ax, ax, ax, :, :]
sy = sy_[ax, ax, ax, :, :]
sz = sz_[ax, ax, ax, :, :]
```

For example, `sx_` is an array of shape `(2,2)`, but now sx is of shape `(1,1,1,2,2)`

[ ]:
```
sx_.shape
```

[ ]:
```
(2, 2)
```

[ ]:
```
sx.shape
```

[ ]:
```
(1, 1, 1, 2, 2)
```

We can check the values of these matrices. For example, we find:

[ ]:
```
print(sx_[1, 0])
print(sx[0, 0, 0, 1, 0])
```

```
1
1
```

This might all seem a bit confusing and overkill but you will soon see why we are doing this.

[ ]:
```
# build k- and w-arrays
def build_k(N):
    return np.arange(-N//2, N//2)/N*2*np.pi
kx_ = build_k(Nkx)
ky_ = build_k(Nky)
dkx = kx_[1] - kx_[0]  # spacing of k-mesh
dky = ky_[1] - ky_[0]
w_ = np.linspace(-7, 7, Nw)
```

We apply the same procedure to the momentum and w-arrays. Remeber our index convention `(w,kx,ky,i,j)`.

`kx` should therefore reshaped into an array of shape `(1,Nkx,1,1,1)` and `w` should be shaped into `(Nw,1,1,1,1)`

This is how it's done:

[ ]:
```
kx = kx_[ax, :, ax, ax, ax]
ky = ky_[ax, ax, :, ax, ax]
w  =  w_[:, ax, ax, ax, ax]
```

What happens if we multiply two of our high-dimensional arrays, say `w*s0`?

At first you might think that this would result in an arrow. The operator `*` defines element-wise multiplication in Python, but the two arrays do not have the same shape. Remember `w.shape=(Nw,1,1,1,1)` and `s0.shape=(1,1,1,2,2)`.

[ ]:
```
np.shape(w*s0)
```

[ ]:
```
(200, 1, 1, 2, 2)
```

But actually, execution of `w*s0` does not produce an error. This is because of Python broadcasting. Numpy will go through each axis, one by one. The first one has size `Nw` and `1` for `w` and `s0`, respectively. Numpy will now automatically extend the first axis of `s0` from size `1` to `Nw`, copying the values at `s0[0,0,0,:,:]` to tje `s0[i,0,0,:,:]` for `i=1,...,Nw-1`. Then numpy will proceed with second axis and so on until the two arrays have the same shape and element-wise multiplication can be done.

Knowing this, we can very simply build our Hamiltonian,almost copying the derived equation from the assignment. Our work is starting to pay off.

[ ]:
```
H = delta*np.sin(kx)*sx + delta*np.sin(ky)*sy - 2*t * (np.cos(kx)*sz + np.
↪cos(ky)*sz) - mu*sz
```

But here is another advantage. If we wanted to perform the sum of `H` over `kx`, we can do so very simply, using

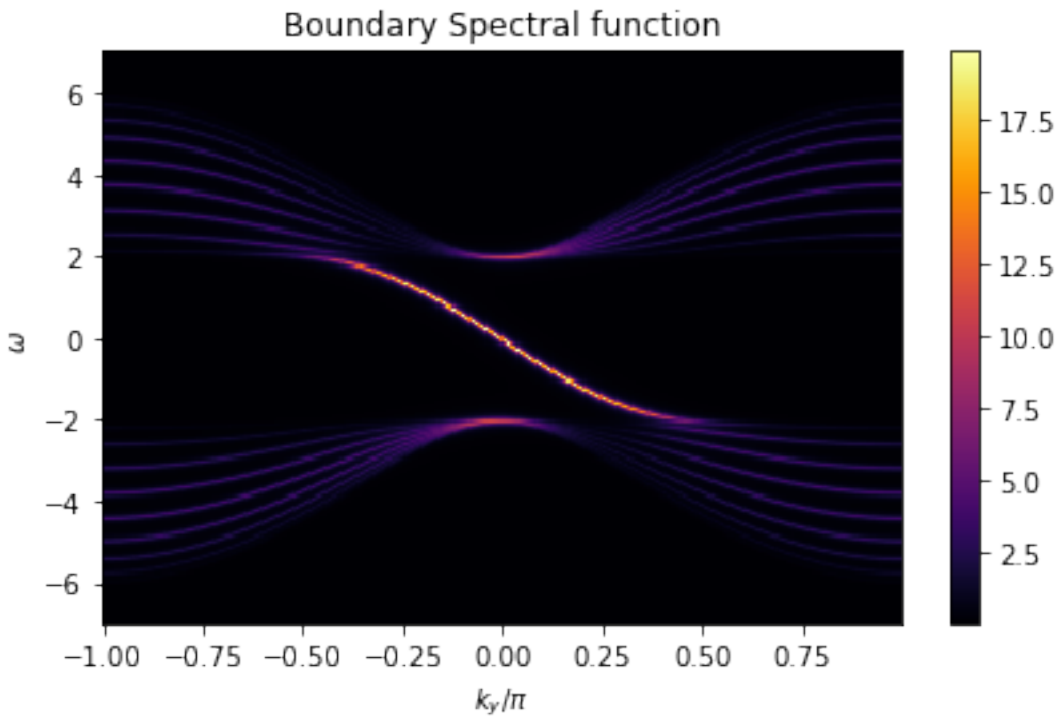
`np.sum(H, axis=1)`

All one needs is the `numpy.sum` function where we have to specify the axis along which `kx` varies `(axis=1)`.

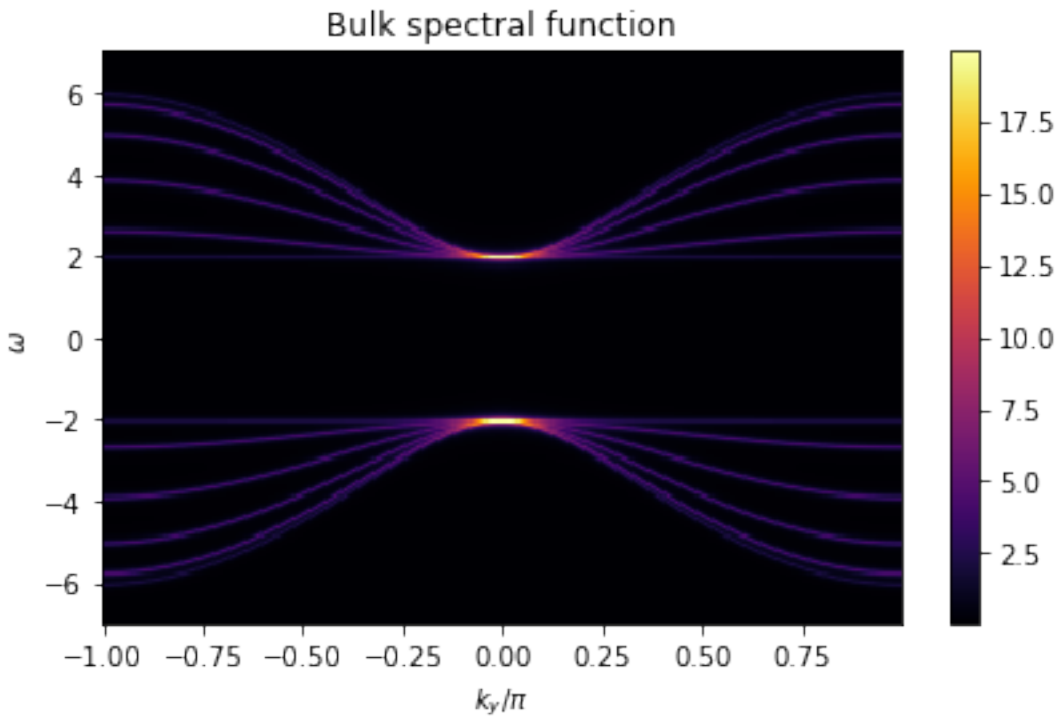
Go through the code below, to see how these fast numpy tricks are used in our case. You will find that all algebraic operations can now be done effortlessly using available numpy functions and without a single for-loop! This strategy produces very fast (close to C speed) and easily readable code. The only disadvantage is that it produces high-dimensional arrays that consume a lot of memory. If memory becomes a problem, it might be wise to switch to a compiled programming language such as *Julia* (or C or Fortran).

[ ]:
```
# build inverse Green's function
G0inv = (w + 1j*eta)*s0 - H

# Take the inverse to obtain G
# Note that np.linalg.inv computes the index with respect
# to the last two axes of the array, which is what we want.
# For reference, see bottom of https://numpy.org/doc/stable/reference/routines.
↪linalg.html#module-numpy.linalg
G0 = np.linalg.inv(G0inv)
```

```
# build transfer matrix
T = -np.linalg.inv(dkx/(2*np.pi)*np.sum(G0, axis=1))

# fourier transform G0
G0x = dkx/(2*np.pi)*np.sum(np.exp(1j*x*kx)*G0, axis=1)
G0x_ = dkx/(2*np.pi)*np.sum(np.exp(-1j*x*kx)*G0, axis=1)

# build GTG
# in numpy the matrix multiplication operator is given by @
# which is the short form for np.matmul
# by default, it interprets the last two array axes at the matrix indices,␣
 ↪which is what we want
# see reference: https://numpy.org/doc/stable/reference/generated/numpy.matmul.
 ↪html
GTG = G0x @ T @ G0x_

# compute the bulk Green's function
Gbulk = dkx/(2*np.pi)*np.sum(G0, axis=1)

# compute surface Green's function
Gsurface = Gbulk + GTG

# take the trace and imaginary part of Gsurface to get the spectral function
# remember that the last to axes (-1) and (-2) denote the matrix indices of G_ij
# Here -1 specifies the last axis and -2 specifies the second-to-last axis
As = -np.imag(np.trace(Gsurface, axis1=-1, axis2=-2))

# plot the result
plt.figure('bound spect f')
plt.clf()
plt.pcolormesh(ky_/np.pi, w_, As, cmap='inferno', shading='auto')
plt.xlabel('$k_y/\pi$')
plt.ylabel('$\omega$')
plt.title('Boundary Spectral function')
plt.colorbar()
plt.tight_layout()
```



[ ]:
```
# For comparison, we also plot the bulk spectral function integrated
# over all ky. Here, edge modes are absent, since the system
# is translationally invariant.
plt.figure('Bulk spectral function')
plt.clf()
plt.pcolormesh(ky_/np.pi, w_, -np.imag(np.trace(Gbulk, axis1=-1, axis2=-2)),␣
↪cmap='inferno', shading='auto')
plt.xlabel('$k_y/\pi$')
plt.ylabel('$\omega$')
plt.title('Bulk spectral function')
plt.colorbar()
plt.tight_layout()
```


Bulk spectral function
6
4
2
0
−2
−4
−6
ω
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
k_y/π
17.5
15.0
12.5
10.0
7.5
5.0
2.5