```python
import numpy as np

def bloch_waves_generator(Ej: float, phase:float, r: float, Ec: float, q: float,
Nmax: int) -> np.ndarray:
    """
    Generate the matrix representation of the Andreev Hamiltonian in the Bloch
waves representation.

    Parameters:
    - Ej (float): Josephson energy.
    - phase (float): Phase.
    - r (float): Reflectivity.
    - Ec (float): Charging energy.
    - q (float): Quasimomentum.
    - N (int): Size parameter for the matrix. The actual size will be 2*(N+1) x 2*
(N+1).

    Returns:
    - np.ndarray: Generated matrix.
    """
    dimension = 2 * (2* Nmax + 1)
    matrix = np.zeros((dimension, dimension))

    G = np.repeat(np.arange(-Nmax, Nmax + 1),2) / 2 #this results in i.e.:
[...,-1/2,-1/2,0/2,0/2,1/2,1/2,...]
    q_vals = q - G
    matrix[np.diag_indices(dimension)] = 4 * Ec * q_vals ** 2

    off_diag1 = np.zeros(dimension - 1)
    off_diag1[1::2] = -r * Ej / 2 # interleaved with zeros.
    np.fill_diagonal(matrix[1:], off_diag1)
    np.fill_diagonal(matrix[:, 1:], off_diag1)

    off_diag2 = np.zeros(dimension - 2)
    off_diag2[::2] = Ej / 2
    off_diag2[1::2] = -Ej / 2
    np.fill_diagonal(matrix[2:], off_diag2)
    np.fill_diagonal(matrix[:, 2:], off_diag2)

    off_diag3 = np.zeros(dimension - 3)
    off_diag3[::2] = r * Ej / 2 # interleaved with zeros.
    np.fill_diagonal(matrix[3:], off_diag3)
    np.fill_diagonal(matrix[:, 3:], off_diag3)

    eigvals,eigvecs = np.linalg.eigh(matrix)
    phase_factor = np.exp(1j * G * phase)
    bloch_waves = phase_factor * eigvecs

    return bloch_waves

Ej = 1
Ec = 2.5
```

```
r = 0.05
Nmax = 5

dimension = 2 * (2*Nmax + 1)
phase_list = np.linspace(-2*np.pi, 2*np.pi, 101)
q_list = np.linspace(-1/4, -1/2, 61)

bloch_waves_list = np.zeros((len(phase_list),len(q_list),dimension,dimension),
dtype=complex)

for i , phase in enumerate(tqdm(phase_list)):
    for j, q in enumerate(q_list):
        bloch_waves_list[i,j] = bloch_waves_generator(Ej, phase, r, Ec, q, Nmax)

derivative_bloch_waves_list = np.gradient(bloch_waves_list,q_list,axis=1,
edge_order=1)

product_tensor = np.zeros((len(phase_list),len(q_list), dimension, dimension),
dtype= complex)
for phase_idx in tqdm(range(len(phase_list))):
    for q_idx in range(len(q_list)):
        for n in range(dimension):
            for m in range(dimension):
                product_tensor[phase_idx,q_idx,n,m] =
np.vdot(bloch_waves_list[phase_idx,q_idx,n],
derivative_bloch_waves_list[phase_idx,q_idx,m])

Omega_tensor = 1j * np.trapz(product_tensor,phase_list,axis = 0)
```