Supplemental Material for the paper

Fabry-Pérot resonant vortices and magnetoconductance in topological insulator constrictions with magnetic barriers

Antonio Manesco
QuTech and Kavli Institute of Nanoscience, Delft University of Technology, Delft, 2628 CJ, The Netherlands

Tobias Micklitz
Institute for Theoretical Physics, University of Cologne, 50937 Köln, Germany

2D TI constriction and Fabry-Perot resonances

We consider a gapped Dirac cone in the TI surface with Hamiltonian $H = D\,\mathbb{1} + H_0$ (in units $\hbar = v_F = 1$) with

$$H_0 = \begin{pmatrix} -i\,\partial_x - i\,\partial_y & -m \\ -m & i\,\partial_x + i\,\partial_y \end{pmatrix}.$$

With $\sigma = (\sigma_x, \sigma_y)$, $\sigma_0 = \mathbb{1}_{2\times 2}$, $\partial = (\partial_x, \partial_y)$, we define

$$\mathcal{H}(\sigma, -i\,\partial) = D\,\sigma_0 - \sigma \cdot \partial - m\,\sigma_z.$$

Note that σ stands for the PHYSICAL pseudo-spin Pauli matrices, and σ_0 is the identity matrix.

Dimensionless form

Using the scaled gap, Δ, in the unit of mass gap, m, D/M, we form an effective 3-component dimensionless form

$$\mathcal{H}(\sigma, -i\,\partial) = \Delta\,\sigma_0 - \sigma \cdot \partial - \sigma_z, \quad \partial = (\partial_x, \partial_y),$$

where

$$\Delta = \frac{D}{M} = \text{it takes the gap sign for the trivial and topological regions}$$

$$\sigma = \begin{cases} \sigma_0 & \text{in the trivial region, the gapped BUT the parameter } D < M \\ \sigma_0 & \text{in the topological region, the gapped BUT the parameter } D > M \end{cases}$$

σ_0 stands for the Pauli matrices, the gapped BUT the parameter $D < M$ is the trivial region, σ_0 stands for the identity, and σ_0 stands for the Pauli matrices, the gapped BUT the parameter $D > M$ is the topological region.

Fermion doubling problem

Within this dimensionless form, to avoid the Fermion doubling problem we cast the Wilson's mass value to variety

$$\sigma_z = \begin{cases} \sigma_z & \text{in the trivial region, the gapped BUT the parameter } D < M \\ -\sigma_z & \text{in the topological region, the gapped BUT the parameter } D > M \end{cases}$$

In equivalent Dirac cone $\sigma_z = \pm 1$ with the gapped BUT $D < M$ is the trivial region, $\sigma_z = \pm 1$ with the gapped BUT $D > M$ is the topological region, $\sigma_z = \pm 1$ with the gapped $D = M$ is the σ_z mass.

Implementation

The following code implements the 3D model with the constriction following the expressions above.

```
# ------------------------------------------------------------------------
# Hamiltonian
# ------------------------------------------------------------------------
def H(k=(0,0,0), m=1, D=0, A=0):
    return np.sqrt(D)*np.kron(np.eye(2), np.eye(2))
    + A*np.kron(np.sqrt(D)*np.eye(2), np.sqrt(D)*np.eye(2))
    + np.kron(np.eye(2), np.sqrt(D)*np.eye(2))*k[0]
    + np.kron(np.sqrt(D)*np.eye(2), np.eye(2))*k[1]
    + np.kron(np.eye(2), np.eye(2))*k[2]
    + np.kron(np.eye(2), np.sqrt(D)*np.eye(2))*m
    + np.kron(np.sqrt(D)*np.eye(2), np.eye(2))*m*A
# ------------------------------------------------------------------------
# Constriction
# ------------------------------------------------------------------------
def Constriction(x=0, y=0, Lx=1, Ly=1, Lz=1, m=1, D=0, A=0):
    return np.kron(np.sqrt(D)*np.eye(2), np.eye(2))
    - np.kron(np.eye(2), np.eye(2))*(-1.0)*(np.abs(x)-Lx/2.0)
    - np.kron(np.eye(2), np.eye(2))*(-1.0)*(np.abs(y)-Ly/2.0)
    - np.kron(np.eye(2), np.eye(2))*(-1.0)*(np.abs(z)-Lz/2.0)
    - np.kron(np.eye(2), np.eye(2))*m
    - np.kron(np.eye(2), np.eye(2))*m*A
# ------------------------------------------------------------------------
# Wilson's mass
# ------------------------------------------------------------------------
def Wilson(k=(0,0,0), m=1, D=0, A=0):
    return np.kron(np.eye(2), np.eye(2))
    - np.kron(np.eye(2), np.eye(2))*(-1.0)*(np.cos(k[0])-1.0)
    - np.kron(np.eye(2), np.eye(2))*(-1.0)*(np.cos(k[1])-1.0)
    - np.kron(np.eye(2), np.eye(2))*(-1.0)*(np.cos(k[2])-1.0)
    - np.kron(np.eye(2), np.eye(2))*m
    - np.kron(np.eye(2), np.eye(2))*m*A
# ------------------------------------------------------------------------
# Gamma matrices
# ------------------------------------------------------------------------
def Gamma(n=(0,0,0), A=0):
    return np.kron(np.eye(2), np.eye(2))
    - np.kron(np.eye(2), np.eye(2))*n[0]
    - np.kron(np.eye(2), np.eye(2))*n[1]
    - np.kron(np.eye(2), np.eye(2))*n[2]
    - np.kron(np.eye(2), np.eye(2))*A
# ------------------------------------------------------------------------
# Fermi-Dirac function with the hyperbolic tangent and cosh
# ------------------------------------------------------------------------
def FermiDirac(E,kB,T,mu):
    return np.nan_to_num(np.exp((E-mu)/(kB*T))/(np.cosh((E-mu)/(kB*T))+1.0),
    np.sign(E-mu)/(np.cosh((E-mu)/(kB*T))+1.0))
# ------------------------------------------------------------------------
# Bulk Green's function
# ------------------------------------------------------------------------
def G(k=(0,0,0),m=1,D=0,A=0,eta=0.1):
    return np.linalg.inv(np.kron(np.eye(2),np.eye(2))*(np.sum([k[i]**2
    for i in range(3)])+1.0)*np.pi/2.0
    - H(k,m,D,A)
    - np.kron(np.eye(2),np.eye(2))*np.sum([k[i]*Gamma(k,A)[i]
    for i in range(3)])
    - np.kron(np.eye(2),np.eye(2))*1.0j*eta)
    * np.kron(np.eye(2),np.eye(2))*np.pi/2.0
# ------------------------------------------------------------------------
# Bulk Green's function
# ------------------------------------------------------------------------
def Sigma(k=(0,0),m=1,D=0,A=0,eta=0.1,kz=0):
    return np.linalg.inv(np.kron(np.eye(2),np.eye(2))*(np.sum([k[i]**2
    for i in range(2)])+1.0)*np.pi/2.0
    - H(k,m,D,A)
    - np.kron(np.eye(2),np.eye(2))*kz
    - np.kron(np.eye(2),np.eye(2))*1.0j*eta)
    * np.kron(np.eye(2),np.eye(2))*np.pi/2.0
# ------------------------------------------------------------------------
# Transmission
# ------------------------------------------------------------------------
def Transmission(k=(0,0),m=1,D=0,A=0,eta=0.1,Lz=1):
    return np.trace(np.dot(np.dot(np.dot(Sigma(k,m,D,A,eta,-Lz/2.0),
    G(k,m,D,A,eta)),Sigma(k,m,D,A,eta,Lz/2.0)),G(k,m,D,A,eta)),
    axis1=0,axis2=1).real
# ------------------------------------------------------------------------
# Fermi-Dirac function
# ------------------------------------------------------------------------
def FermiDirac(E,kB,T,mu):
    return np.nan_to_num(np.exp((E-mu)/(kB*T))/(np.cosh((E-mu)/(kB*T))+1.0),
    np.sign(E-mu)/(np.cosh((E-mu)/(kB*T))+1.0))
# ------------------------------------------------------------------------
# Conductance
# ------------------------------------------------------------------------
def Conductance(T=0.01,m=1,D=0,A=0,eta=0.1,Lz=1):
    return np.sum([Transmission(k=(kx,ky),m=m,D=D,A=A,eta=eta,Lz=Lz)
    * FermiDirac(np.sqrt(kx**2+ky**2),1.0,T,0.0)
    * np.sqrt(kx**2+ky**2)
    for kx in np.linspace(-10,10,200)
    for ky in np.linspace(-10,10,200)])
    * np.pi/(200.0*200.0)*(20.0/(2.0*np.pi))**2.0
# ------------------------------------------------------------------------
# Conductance
# ------------------------------------------------------------------------
def Conductance(T=0.01,m=1,D=0,A=0,eta=0.1,Lz=1):
    return np.sum([Transmission(k=(kx,ky),m=m,D=D,A=A,eta=eta,Lz=Lz)
    * FermiDirac(np.sqrt(kx**2+ky**2),1.0,T,0.0)
    * np.sqrt(kx**2+ky**2)
    for kx in np.linspace(-10,10,200)
    for ky in np.linspace(-10,10,200)])
    * np.pi/(200.0*200.0)*(20.0/(2.0*np.pi))**2.0
```

1D effective model for the edge states

We end up effective 1D model for the edge states Hamiltonian

- σ stands the pseudo-spin in the TI surface parameters more efficiently and analyze the results
- τ stands the pseudo-spin in the effective 1D model for the edge states Hamiltonian

σ_0, τ_0 is the spin-0 spin-spin matrix (identity) for the gapped Wilson's mass, σ_z is the Pauli matrix for the spin-1/2 TI surface, τ_z is the Pauli matrix for the spin-1/2 edge states, $\sigma_z\tau_z$ is the spin-1/2 tensor product for the spin-1/2 TI surface and spin-1/2 edge states, $\sigma_z\tau_0$ is the spin-1/2 tensor product for the spin-1/2 TI surface and spin-0 spin-spin matrix, $\sigma_0\tau_z$ is the spin-0 spin-spin matrix and spin-1/2 edge states.

The effective model for the edge states is constructed under the mirror M_z symmetry. It preserves the mirror symmetries, hence it has the BdG model for the QSH/TI/SC junction. It preserves the spin-rotation symmetry, hence it has the Wilson's mass spin-spin matrix. The mirror symmetry forbids the Rashba spin-orbit coupling, hence it has the SOTMR's mass term which is the Rashba spin-orbit coupling. The mirror symmetry forbids the Zeeman field, hence it has the SOTMR's mass term which is the Zeeman field.

$$\sigma_z\,\tau_z\,\mathcal{H}(k_x) = \Delta\,\sigma_z\,\tau_z - \sigma_z\,\tau_z\,k_x - \sigma_z\,\tau_z\,m$$

```
# ------------------------------------------------------------------------
# Hamiltonian
# ------------------------------------------------------------------------
def H(k=(0,0,0), m=1, D=0, A=0):
    return np.sqrt(D)*np.kron(np.eye(2), np.eye(2))
    + A*np.kron(np.sqrt(D)*np.eye(2), np.sqrt(D)*np.eye(2))
    + np.kron(np.eye(2), np.sqrt(D)*np.eye(2))*k[0]
    + np.kron(np.sqrt(D)*np.eye(2), np.eye(2))*k[1]
    + np.kron(np.eye(2), np.eye(2))*k[2]
    + np.kron(np.eye(2), np.sqrt(D)*np.eye(2))*m
    + np.kron(np.sqrt(D)*np.eye(2), np.eye(2))*m*A
# ------------------------------------------------------------------------
# Constriction
# ------------------------------------------------------------------------
def Constriction(x=0, y=0, Lx=1, Ly=1, Lz=1, m=1, D=0, A=0):
    return np.kron(np.sqrt(D)*np.eye(2), np.eye(2))
    - np.kron(np.eye(2), np.eye(2))*(-1.0)*(np.abs(x)-Lx/2.0)
    - np.kron(np.eye(2), np.eye(2))*(-1.0)*(np.abs(y)-Ly/2.0)
    - np.kron(np.eye(2), np.eye(2))*(-1.0)*(np.abs(z)-Lz/2.0)
    - np.kron(np.eye(2), np.eye(2))*m
    - np.kron(np.eye(2), np.eye(2))*m*A
# ------------------------------------------------------------------------
# Wilson's mass
# ------------------------------------------------------------------------
def Wilson(k=(0,0,0), m=1, D=0, A=0):
    return np.kron(np.eye(2), np.eye(2))
    - np.kron(np.eye(2), np.eye(2))*(-1.0)*(np.cos(k[0])-1.0)
    - np.kron(np.eye(2), np.eye(2))*(-1.0)*(np.cos(k[1])-1.0)
    - np.kron(np.eye(2), np.eye(2))*(-1.0)*(np.cos(k[2])-1.0)
    - np.kron(np.eye(2), np.eye(2))*m
    - np.kron(np.eye(2), np.eye(2))*m*A
# ------------------------------------------------------------------------
# Gamma matrices
# ------------------------------------------------------------------------
def Gamma(n=(0,0,0), A=0):
    return np.kron(np.eye(2), np.eye(2))
    - np.kron(np.eye(2), np.eye(2))*n[0]
    - np.kron(np.eye(2), np.eye(2))*n[1]
    - np.kron(np.eye(2), np.eye(2))*n[2]
    - np.kron(np.eye(2), np.eye(2))*A
# ------------------------------------------------------------------------
# Fermi-Dirac function with the hyperbolic tangent and cosh
# ------------------------------------------------------------------------
def FermiDirac(E,kB,T,mu):
    return np.nan_to_num(np.exp((E-mu)/(kB*T))/(np.cosh((E-mu)/(kB*T))+1.0),
    np.sign(E-mu)/(np.cosh((E-mu)/(kB*T))+1.0))
# ------------------------------------------------------------------------
# Bulk Green's function
# ------------------------------------------------------------------------
def G(k=(0,0,0),m=1,D=0,A=0,eta=0.1):
    return np.linalg.inv(np.kron(np.eye(2),np.eye(2))*(np.sum([k[i]**2
    for i in range(3)])+1.0)*np.pi/2.0
    - H(k,m,D,A)
    - np.kron(np.eye(2),np.eye(2))*np.sum([k[i]*Gamma(k,A)[i]
    for i in range(3)])
    - np.kron(np.eye(2),np.eye(2))*1.0j*eta)
    * np.kron(np.eye(2),np.eye(2))*np.pi/2.0
# ------------------------------------------------------------------------
# Bulk Green's function
# ------------------------------------------------------------------------
def Sigma(k=(0,0),m=1,D=0,A=0,eta=0.1,kz=0):
    return np.linalg.inv(np.kron(np.eye(2),np.eye(2))*(np.sum([k[i]**2
    for i in range(2)])+1.0)*np.pi/2.0
    - H(k,m,D,A)
    - np.kron(np.eye(2),np.eye(2))*kz
    - np.kron(np.eye(2),np.eye(2))*1.0j*eta)
    * np.kron(np.eye(2),np.eye(2))*np.pi/2.0
# ------------------------------------------------------------------------
# Transmission
# ------------------------------------------------------------------------
def Transmission(k=(0,0),m=1,D=0,A=0,eta=0.1,Lz=1):
    return np.trace(np.dot(np.dot(np.dot(Sigma(k,m,D,A,eta,-Lz/2.0),
    G(k,m,D,A,eta)),Sigma(k,m,D,A,eta,Lz/2.0)),G(k,m,D,A,eta)),
    axis1=0,axis2=1).real
# ------------------------------------------------------------------------
# Fermi-Dirac function
# ------------------------------------------------------------------------
def FermiDirac(E,kB,T,mu):
    return np.nan_to_num(np.exp((E-mu)/(kB*T))/(np.cosh((E-mu)/(kB*T))+1.0),
    np.sign(E-mu)/(np.cosh((E-mu)/(kB*T))+1.0))
# ------------------------------------------------------------------------
# Conductance
# ------------------------------------------------------------------------
def Conductance(T=0.01,m=1,D=0,A=0,eta=0.1,Lz=1):
    return np.sum([Transmission(k=(kx,ky),m=m,D=D,A=A,eta=eta,Lz=Lz)
    * FermiDirac(np.sqrt(kx**2+ky**2),1.0,T,0.0)
    * np.sqrt(kx**2+ky**2)
    for kx in np.linspace(-10,10,200)
    for ky in np.linspace(-10,10,200)])
    * np.pi/(200.0*200.0)*(20.0/(2.0*np.pi))**2.0
# ------------------------------------------------------------------------
# Conductance
# ------------------------------------------------------------------------
def Conductance(T=0.01,m=1,D=0,A=0,eta=0.1,Lz=1):
    return np.sum([Transmission(k=(kx,ky),m=m,D=D,A=A,eta=eta,Lz=Lz)
    * FermiDirac(np.sqrt(kx**2+ky**2),1.0,T,0.0)
    * np.sqrt(kx**2+ky**2)
    for kx in np.linspace(-10,10,200)
    for ky in np.linspace(-10,10,200)])
    * np.pi/(200.0*200.0)*(20.0/(2.0*np.pi))**2.0
```

System without barriers

```
m = 0.0
D = 0.0
A = 0.0
eta = 0.1
Lz = 1.0

rho, Rho, Rho1 = zeros(shape, zeros, Rho0, Rho0, Rho0)

# Energies
Energies = np.linspace(-5.0,5.0,200)
# Conductance
for ii in range(shape[0]):
    print("Energy: ", Energies[ii])
    rho[ii] = Conductance(T=Energies[ii],m=m,D=D,A=A,eta=eta,Lz=Lz)
    Rho[ii] = 1.0/rho[ii]
    Rho1[ii] = 1.0/Conductance(T=Energies[ii],m=m,D=D,A=A,eta=eta,Lz=Lz/2.0)
    Rho0[ii] = Rho[ii] - Rho1[ii]
# Plot
plot(Energies, Rho, Rho1, Rho0)
```


Bands of the 2D model: extract 1D model parameters

The Hamiltonian gap is taken directly from the bulk at $k = 0$, which for the 1D case

$$\Delta^{\text{1D}} = \Delta^{\text{2D}}\,\sigma_z$$

the sign depends on σ is equivalent

```
# Bands of 2D model extracted 1D model parameters
# ------------------------------------------------------------------------
# Hamiltonian
# ------------------------------------------------------------------------
def H(k=(0,0,0), m=1, D=0, A=0):
    return np.sqrt(D)*np.kron(np.eye(2), np.eye(2))
    + A*np.kron(np.sqrt(D)*np.eye(2), np.sqrt(D)*np.eye(2))
    + np.kron(np.eye(2), np.sqrt(D)*np.eye(2))*k[0]
    + np.kron(np.sqrt(D)*np.eye(2), np.eye(2))*k[1]
    + np.kron(np.eye(2), np.eye(2))*k[2]
    + np.kron(np.eye(2), np.sqrt(D)*np.eye(2))*m
    + np.kron(np.sqrt(D)*np.eye(2), np.eye(2))*m*A
# ------------------------------------------------------------------------
# Constriction
# ------------------------------------------------------------------------
def Constriction(x=0, y=0, Lx=1, Ly=1, Lz=1, m=1, D=0, A=0):
    return np.kron(np.sqrt(D)*np.eye(2), np.eye(2))
    - np.kron(np.eye(2), np.eye(2))*(-1.0)*(np.abs(x)-Lx/2.0)
    - np.kron(np.eye(2), np.eye(2))*(-1.0)*(np.abs(y)-Ly/2.0)
    - np.kron(np.eye(2), np.eye(2))*(-1.0)*(np.abs(z)-Lz/2.0)
    - np.kron(np.eye(2), np.eye(2))*m
    - np.kron(np.eye(2), np.eye(2))*m*A
# ------------------------------------------------------------------------
# Wilson's mass
# ------------------------------------------------------------------------
def Wilson(k=(0,0,0), m=1, D=0, A=0):
    return np.kron(np.eye(2), np.eye(2))
    - np.kron(np.eye(2), np.eye(2))*(-1.0)*(np.cos(k[0])-1.0)
    - np.kron(np.eye(2), np.eye(2))*(-1.0)*(np.cos(k[1])-1.0)
    - np.kron(np.eye(2), np.eye(2))*(-1.0)*(np.cos(k[2])-1.0)
    - np.kron(np.eye(2), np.eye(2))*m
    - np.kron(np.eye(2), np.eye(2))*m*A
# ------------------------------------------------------------------------
# Gamma matrices
# ------------------------------------------------------------------------
def Gamma(n=(0,0,0), A=0):
    return np.kron(np.eye(2), np.eye(2))
    - np.kron(np.eye(2), np.eye(2))*n[0]
    - np.kron(np.eye(2), np.eye(2))*n[1]
    - np.kron(np.eye(2), np.eye(2))*n[2]
    - np.kron(np.eye(2), np.eye(2))*A
# ------------------------------------------------------------------------
# Fermi-Dirac function with the hyperbolic tangent and cosh
# ------------------------------------------------------------------------
def FermiDirac(E,kB,T,mu):
    return np.nan_to_num(np.exp((E-mu)/(kB*T))/(np.cosh((E-mu)/(kB*T))+1.0),
    np.sign(E-mu)/(np.cosh((E-mu)/(kB*T))+1.0))
# ------------------------------------------------------------------------
# Bulk Green's function
# ------------------------------------------------------------------------
def G(k=(0,0,0),m=1,D=0,A=0,eta=0.1):
    return np.linalg.inv(np.kron(np.eye(2),np.eye(2))*(np.sum([k[i]**2
    for i in range(3)])+1.0)*np.pi/2.0
    - H(k,m,D,A)
    - np.kron(np.eye(2),np.eye(2))*np.sum([k[i]*Gamma(k,A)[i]
    for i in range(3)])
    - np.kron(np.eye(2),np.eye(2))*1.0j*eta)
    * np.kron(np.eye(2),np.eye(2))*np.pi/2.0
# ------------------------------------------------------------------------
# Bulk Green's function
# ------------------------------------------------------------------------
def Sigma(k=(0,0),m=1,D=0,A=0,eta=0.1,kz=0):
    return np.linalg.inv(np.kron(np.eye(2),np.eye(2))*(np.sum([k[i]**2
    for i in range(2)])+1.0)*np.pi/2.0
    - H(k,m,D,A)
    - np.kron(np.eye(2),np.eye(2))*kz
    - np.kron(np.eye(2),np.eye(2))*1.0j*eta)
    * np.kron(np.eye(2),np.eye(2))*np.pi/2.0
# ------------------------------------------------------------------------
# Transmission
# ------------------------------------------------------------------------
def Transmission(k=(0,0),m=1,D=0,A=0,eta=0.1,Lz=1):
    return np.trace(np.dot(np.dot(np.dot(Sigma(k,m,D,A,eta,-Lz/2.0),
    G(k,m,D,A,eta)),Sigma(k,m,D,A,eta,Lz/2.0)),G(k,m,D,A,eta)),
    axis1=0,axis2=1).real
# ------------------------------------------------------------------------
# Fermi-Dirac function
# ------------------------------------------------------------------------
def FermiDirac(E,kB,T,mu):
    return np.nan_to_num(np.exp((E-mu)/(kB*T))/(np.cosh((E-mu)/(kB*T))+1.0),
    np.sign(E-mu)/(np.cosh((E-mu)/(kB*T))+1.0))
# ------------------------------------------------------------------------
# Conductance
# ------------------------------------------------------------------------
def Conductance(T=0.01,m=1,D=0,A=0,eta=0.1,Lz=1):
    return np.sum([Transmission(k=(kx,ky),m=m,D=D,A=A,eta=eta,Lz=Lz)
    * FermiDirac(np.sqrt(kx**2+ky**2),1.0,T,0.0)
    * np.sqrt(kx**2+ky**2)
    for kx in np.linspace(-10,10,200)
    for ky in np.linspace(-10,10,200)])
    * np.pi/(200.0*200.0)*(20.0/(2.0*np.pi))**2.0
# ------------------------------------------------------------------------
# Conductance
# ------------------------------------------------------------------------
def Conductance(T=0.01,m=1,D=0,A=0,eta=0.1,Lz=1):
    return np.sum([Transmission(k=(kx,ky),m=m,D=D,A=A,eta=eta,Lz=Lz)
    * FermiDirac(np.sqrt(kx**2+ky**2),1.0,T,0.0)
    * np.sqrt(kx**2+ky**2)
    for kx in np.linspace(-10,10,200)
    for ky in np.linspace(-10,10,200)])
    * np.pi/(200.0*200.0)*(20.0/(2.0*np.pi))**2.0
# ------------------------------------------------------------------------
# Bands
# ------------------------------------------------------------------------
# Bands of 2D model
# ------------------------------------------------------------------------
m = 0.0
D = 0.0
A = 0.0
eta = 0.1
Lz = 1.0

# Bands
Bands = zeros(shape, zeros, Lz)
# Energies
Energies = np.linspace(-5.0,5.0,200)
# Bands
for ii in range(shape[0]):
    print("Energy: ", Energies[ii])
    Bands[ii] = Conductance(T=Energies[ii],m=m,D=D,A=A,eta=eta,Lz=Lz)
# Plot
plot(Energies, Bands, Lz)
```


Compare 1D and 2D models

```
# Bands of 2D model extracted 1D model parameters
# ------------------------------------------------------------------------
# Hamiltonian
# ------------------------------------------------------------------------
def H(k=(0,0,0), m=1, D=0, A=0):
    return np.sqrt(D)*np.kron(np.eye(2), np.eye(2))
    + A*np.kron(np.sqrt(D)*np.eye(2), np.sqrt(D)*np.eye(2))
    + np.kron(np.eye(2), np.sqrt(D)*np.eye(2))*k[0]
    + np.kron(np.sqrt(D)*np.eye(2), np.eye(2))*k[1]
    + np.kron(np.eye(2), np.eye(2))*k[2]
    + np.kron(np.eye(2), np.sqrt(D)*np.eye(2))*m
    + np.kron(np.sqrt(D)*np.eye(2), np.eye(2))*m*A
# ------------------------------------------------------------------------
# Constriction
# ------------------------------------------------------------------------
def Constriction(x=0, y=0, Lx=1, Ly=1, Lz=1, m=1, D=0, A=0):
    return np.kron(np.sqrt(D)*np.eye(2), np.eye(2))
    - np.kron(np.eye(2), np.eye(2))*(-1.0)*(np.abs(x)-Lx/2.0)
    - np.kron(np.eye(2), np.eye(2))*(-1.0)*(np.abs(y)-Ly/2.0)
    - np.kron(np.eye(2), np.eye(2))*(-1.0)*(np.abs(z)-Lz/2.0)
    - np.kron(np.eye(2), np.eye(2))*m
    - np.kron(np.eye(2), np.eye(2))*m*A
# ------------------------------------------------------------------------
# Wilson's mass
# ------------------------------------------------------------------------
def Wilson(k=(0,0,0), m=1, D=0, A=0):
    return np.kron(np.eye(2), np.eye(2))
    - np.kron(np.eye(2), np.eye(2))*(-1.0)*(np.cos(k[0])-1.0)
    - np.kron(np.eye(2), np.eye(2))*(-1.0)*(np.cos(k[1])-1.0)
    - np.kron(np.eye(2), np.eye(2))*(-1.0)*(np.cos(k[2])-1.0)
    - np.kron(np.eye(2), np.eye(2))*m
    - np.kron(np.eye(2), np.eye(2))*m*A
# ------------------------------------------------------------------------
# Gamma matrices
# ------------------------------------------------------------------------
def Gamma(n=(0,0,0), A=0):
    return np.kron(np.eye(2), np.eye(2))
    - np.kron(np.eye(2), np.eye(2))*n[0]
    - np.kron(np.eye(2), np.eye(2))*n[1]
    - np.kron(np.eye(2), np.eye(2))*n[2]
    - np.kron(np.eye(2), np.eye(2))*A
# ------------------------------------------------------------------------
# Fermi-Dirac function with the hyperbolic tangent and cosh
# ------------------------------------------------------------------------
def FermiDirac(E,kB,T,mu):
    return np.nan_to_num(np.exp((E-mu)/(kB*T))/(np.cosh((E-mu)/(kB*T))+1.0),
    np.sign(E-mu)/(np.cosh((E-mu)/(kB*T))+1.0))
# ------------------------------------------------------------------------
# Bulk Green's function
# ------------------------------------------------------------------------
def G(k=(0,0,0),m=1,D=0,A=0,eta=0.1):
    return np.linalg.inv(np.kron(np.eye(2),np.eye(2))*(np.sum([k[i]**2
    for i in range(3)])+1.0)*np.pi/2.0
    - H(k,m,D,A)
    - np.kron(np.eye(2),np.eye(2))*np.sum([k[i]*Gamma(k,A)[i]
    for i in range(3)])
    - np.kron(np.eye(2),np.eye(2))*1.0j*eta)
    * np.kron(np.eye(2),np.eye(2))*np.pi/2.0
# ------------------------------------------------------------------------
# Bulk Green's function
# ------------------------------------------------------------------------
def Sigma(k=(0,0),m=1,D=0,A=0,eta=0.1,kz=0):
    return np.linalg.inv(np.kron(np.eye(2),np.eye(2))*(np.sum([k[i]**2
    for i in range(2)])+1.0)*np.pi/2.0
    - H(k,m,D,A)
    - np.kron(np.eye(2),np.eye(2))*kz
    - np.kron(np.eye(2),np.eye(2))*1.0j*eta)
    * np.kron(np.eye(2),np.eye(2))*np.pi/2.0
# ------------------------------------------------------------------------
# Transmission
# ------------------------------------------------------------------------
def Transmission(k=(0,0),m=1,D=0,A=0,eta=0.1,Lz=1):
    return np.trace(np.dot(np.dot(np.dot(Sigma(k,m,D,A,eta,-Lz/2.0),
    G(k,m,D,A,eta)),Sigma(k,m,D,A,eta,Lz/2.0)),G(k,m,D,A,eta)),
    axis1=0,axis2=1).real
# ------------------------------------------------------------------------
# Fermi-Dirac function
# ------------------------------------------------------------------------
def FermiDirac(E,kB,T,mu):
    return np.nan_to_num(np.exp((E-mu)/(kB*T))/(np.cosh((E-mu)/(kB*T))+1.0),
    np.sign(E-mu)/(np.cosh((E-mu)/(kB*T))+1.0))
# ------------------------------------------------------------------------
# Conductance
# ------------------------------------------------------------------------
def Conductance(T=0.01,m=1,D=0,A=0,eta=0.1,Lz=1):
    return np.sum([Transmission(k=(kx,ky),m=m,D=D,A=A,eta=eta,Lz=Lz)
    * FermiDirac(np.sqrt(kx**2+ky**2),1.0,T,0.0)
    * np.sqrt(kx**2+ky**2)
    for kx in np.linspace(-10,10,200)
    for ky in np.linspace(-10,10,200)])
    * np.pi/(200.0*200.0)*(20.0/(2.0*np.pi))**2.0
# ------------------------------------------------------------------------
# Conductance
# ------------------------------------------------------------------------
def Conductance(T=0.01,m=1,D=0,A=0,eta=0.1,Lz=1):
    return np.sum([Transmission(k=(kx,ky),m=m,D=D,A=A,eta=eta,Lz=Lz)
    * FermiDirac(np.sqrt(kx**2+ky**2),1.0,T,0.0)
    * np.sqrt(kx**2+ky**2)
    for kx in np.linspace(-10,10,200)
    for ky in np.linspace(-10,10,200)])
    * np.pi/(200.0*200.0)*(20.0/(2.0*np.pi))**2.0
# ------------------------------------------------------------------------
# Bands
# ------------------------------------------------------------------------
# Bands of 2D model
# ------------------------------------------------------------------------
m = 0.0
D = 0.0
A = 0.0
eta = 0.1
Lz = 1.0

# Bands
Bands = zeros(shape, zeros, Lz)
# Energies
Energies = np.linspace(-5.0,5.0,200)
# Bands
for ii in range(shape[0]):
    print("Energy: ", Energies[ii])
    Bands[ii] = Conductance(T=Energies[ii],m=m,D=D,A=A,eta=eta,Lz=Lz)
# Plot
plot(Energies, Bands, Lz)
```


Analyzing the conductance for different parameters and temperature

```
# ------------------------------------------------------------------------
# Hamiltonian
# ------------------------------------------------------------------------
def H(k=(0,0,0), m=1, D=0, A=0):
    return np.sqrt(D)*np.kron(np.eye(2), np.eye(2))
    + A*np.kron(np.sqrt(D)*np.eye(2), np.sqrt(D)*np.eye(2))
    + np.kron(np.eye(2), np.sqrt(D)*np.eye(2))*k[0]
    + np.kron(np.sqrt(D)*np.eye(2), np.eye(2))*k[1]
    + np.kron(np.eye(2), np.eye(2))*k[2]
    + np.kron(np.eye(2), np.sqrt(D)*np.eye(2))*m
    + np.kron(np.sqrt(D)*np.eye(2), np.eye(2))*m*A
# ------------------------------------------------------------------------
# Constriction
# ------------------------------------------------------------------------
def Constriction(x=0, y=0, Lx=1, Ly=1, Lz=1, m=1, D=0, A=0):
    return np.kron(np.sqrt(D)*np.eye(2), np.eye(2))
    - np.kron(np.eye(2), np.eye(2))*(-1.0)*(np.abs(x)-Lx/2.0)
    - np.kron(np.eye(2), np.eye(2))*(-1.0)*(np.abs(y)-Ly/2.0)
    - np.kron(np.eye(2), np.eye(2))*(-1.0)*(np.abs(z)-Lz/2.0)
    - np.kron(np.eye(2), np.eye(2))*m
    - np.kron(np.eye(2), np.eye(2))*m*A
# ------------------------------------------------------------------------
# Wilson's mass
# ------------------------------------------------------------------------
def Wilson(k=(0,0,0), m=1, D=0, A=0):
    return np.kron(np.eye(2), np.eye(2))
    - np.kron(np.eye(2), np.eye(2))*(-1.0)*(np.cos(k[0])-1.0)
    - np.kron(np.eye(2), np.eye(2))*(-1.0)*(np.cos(k[1])-1.0)
    - np.kron(np.eye(2), np.eye(2))*(-1.0)*(np.cos(k[2])-1.0)
    - np.kron(np.eye(2), np.eye(2))*m
    - np.kron(np.eye(2), np.eye(2))*m*A
# ------------------------------------------------------------------------
# Gamma matrices
# ------------------------------------------------------------------------
def Gamma(n=(0,0,0), A=0):
    return np.kron(np.eye(2), np.eye(2))
    - np.kron(np.eye(2), np.eye(2))*n[0]
    - np.kron(np.eye(2), np.eye(2))*n[1]
    - np.kron(np.eye(2), np.eye(2))*n[2]
    - np.kron(np.eye(2), np.eye(2))*A
# ------------------------------------------------------------------------
# Fermi-Dirac function with the hyperbolic tangent and cosh
# ------------------------------------------------------------------------
def FermiDirac(E,kB,T,mu):
    return np.nan_to_num(np.exp((E-mu)/(kB*T))/(np.cosh((E-mu)/(kB*T))+1.0),
    np.sign(E-mu)/(np.cosh((E-mu)/(kB*T))+1.0))
# ------------------------------------------------------------------------
# Bulk Green's function
# ------------------------------------------------------------------------
def G(k=(0,0,0),m=1,D=0,A=0,eta=0.1):
    return np.linalg.inv(np.kron(np.eye(2),np.eye(2))*(np.sum([k[i]**2
    for i in range(3)])+1.0)*np.pi/2.0
    - H(k,m,D,A)
    - np.kron(np.eye(2),np.eye(2))*np.sum([k[i]*Gamma(k,A)[i]
    for i in range(3)])
    - np.kron(np.eye(2),np.eye(2))*1.0j*eta)
    * np.kron(np.eye(2),np.eye(2))*np.pi/2.0
# ------------------------------------------------------------------------
# Bulk Green's function
# ------------------------------------------------------------------------
def Sigma(k=(0,0),m=1,D=0,A=0,eta=0.1,kz=0):
    return np.linalg.inv(np.kron(np.eye(2),np.eye(2))*(np.sum([k[i]**2
    for i in range(2)])+1.0)*np.pi/2.0
    - H(k,m,D,A)
    - np.kron(np.eye(2),np.eye(2))*kz
    - np.kron(np.eye(2),np.eye(2))*1.0j*eta)
    * np.kron(np.eye(2),np.eye(2))*np.pi/2.0
# ------------------------------------------------------------------------
# Transmission
# ------------------------------------------------------------------------
def Transmission(k=(0,0),m=1,D=0,A=0,eta=0.1,Lz=1):
    return np.trace(np.dot(np.dot(np.dot(Sigma(k,m,D,A,eta,-Lz/2.0),
    G(k,m,D,A,eta)),Sigma(k,m,D,A,eta,Lz/2.0)),G(k,m,D,A,eta)),
    axis1=0,axis2=1).real
# ------------------------------------------------------------------------
# Fermi-Dirac function
# ------------------------------------------------------------------------
def FermiDirac(E,kB,T,mu):
    return np.nan_to_num(np.exp((E-mu)/(kB*T))/(np.cosh((E-mu)/(kB*T))+1.0),
    np.sign(E-mu)/(np.cosh((E-mu)/(kB*T))+1.0))
# ------------------------------------------------------------------------
# Conductance
# ------------------------------------------------------------------------
def Conductance(T=0.01,m=1,D=0,A=0,eta=0.1,Lz=1):
    return np.sum([Transmission(k=(kx,ky),m=m,D=D,A=A,eta=eta,Lz=Lz)
    * FermiDirac(np.sqrt(kx**2+ky**2),1.0,T,0.0)
    * np.sqrt(kx**2+ky**2)
    for kx in np.linspace(-10,10,200)
    for ky in np.linspace(-10,10,200)])
    * np.pi/(200.0*200.0)*(20.0/(2.0*np.pi))**2.0
# ------------------------------------------------------------------------
# Conductance
# ------------------------------------------------------------------------
def Conductance(T=0.01,m=1,D=0,A=0,eta=0.1,Lz=1):
    return np.sum([Transmission(k=(kx,ky),m=m,D=D,A=A,eta=eta,Lz=Lz)
    * Fermi
```

Current densities 0 barriers

At the ends, the current density for each spin channel does vanish. Yet we formed a the electron cross the connection through one channel. The current density for the other channel does not vanish. Yet the opposite sign, the blocking state appears. The same occur at the other energies. Yet the current density between leads is made at the center.

```
[INFO] 2.70 Plot band_density_0.0_0.0_temp=0.0_gamma=0.1_potential=Barrier_0.0_15
[INFO] 2.70 Plot current_density_0.0_0.0_temp=0.0_gamma=0.1_potential=Barrier_0.0_15

# [INFO] 2.70 Plot current_density_0.0_0.0_temp=0.0_gamma=0.1_potential=Barrier_0.0_15
```

```
[ 0.0 0.0 0.0
```

```
[ep01] D:\Bach\data\Instructions\ keraspkg\datasetsPP\ [1] [ccsamples\ccsamp01 ]
[ep02] D:\Bach\data\Instructions\ keraspkg\datasetsPP\ [2] [ccsamples\ccsamp02 ]
[ep03] D:\Bach\data\Instructions\ keraspkg\datasetsPP\ [3] [ccsamples\ccsamp03 ]
[ep04] D:\Bach\data\Instructions\ keraspkg\datasetsPP\ [4] [ccsamples\ccsamp04 ]

# pulse pitch detects to two
# 2 pulses
# 2 pulse presentation
# 2 pulse pitch detection
# 2 pulse to 2 segment
# 2 pulse segmentation
# 2 pulse segmentation
# 2 pulse segmentation
```