Hand's on Session

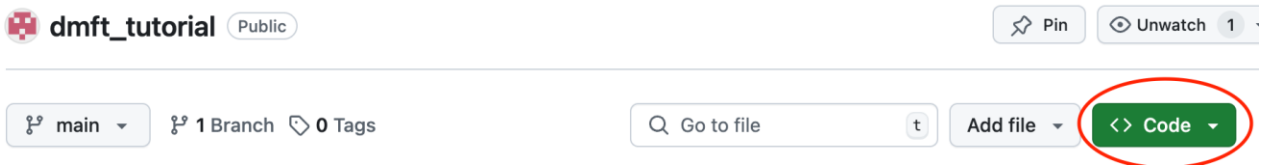
DMFT-Metal Solution

Git clone the tutorial repository

- First install the Git.

  sudo apt-get install git (for linux, ubuntu) brew install git (for mac).

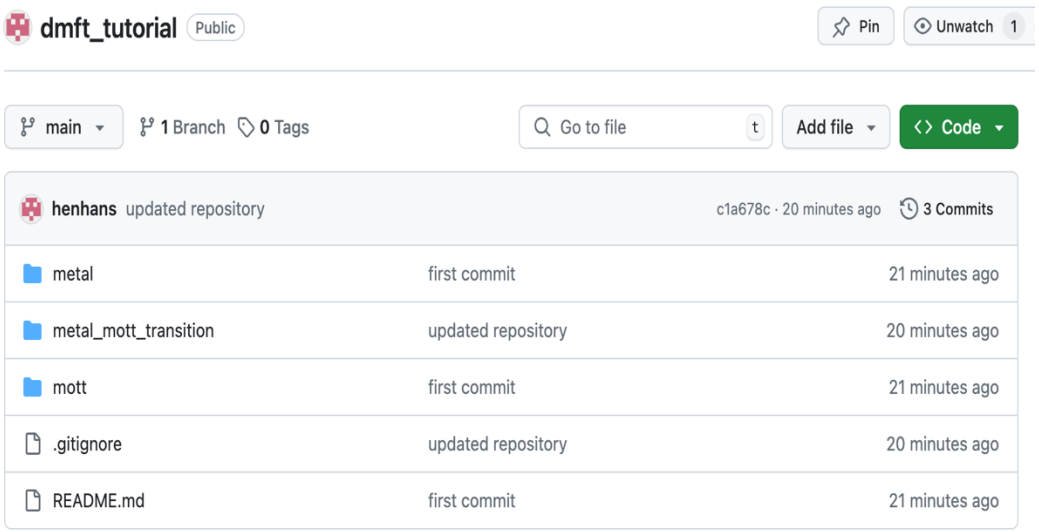
- git clone https://github.com/henhans/dmft_tutorial.
- You can also go to the website to download.



- pip3 install numpy scipy numba (install pip3 using sudo apt-get install python3-pip) or brew install python3-pip.
- Install Jupyter notebook (brew install jupyter, sudo apt-get install jupyter) or install VScode

Tutorials Today

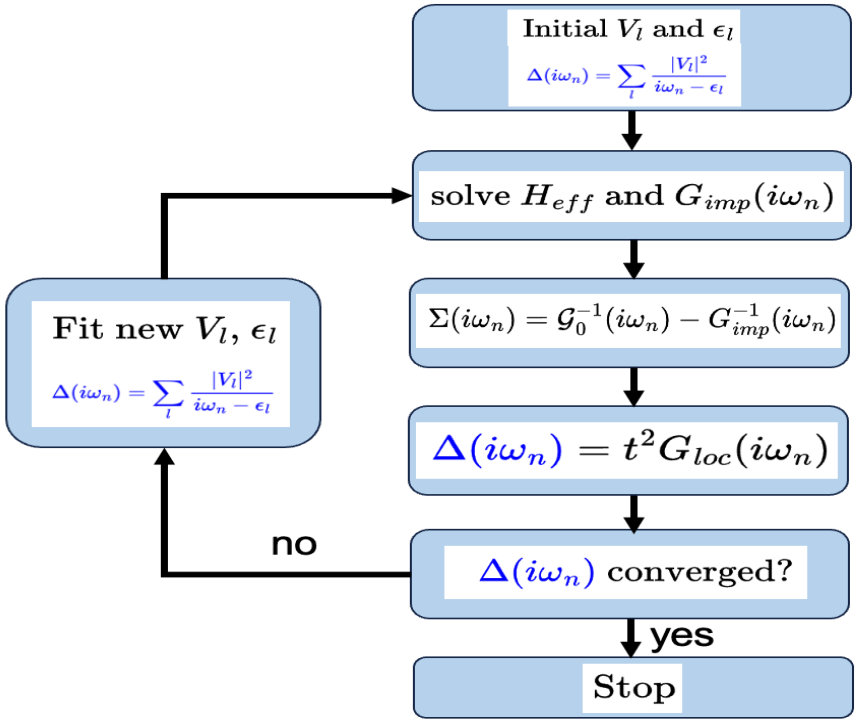
- DMFT-Metal solution
- DMFT-Mott insulator solution
- DMFT metal Mott insulator transition

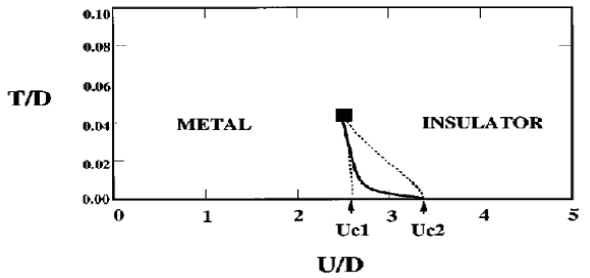


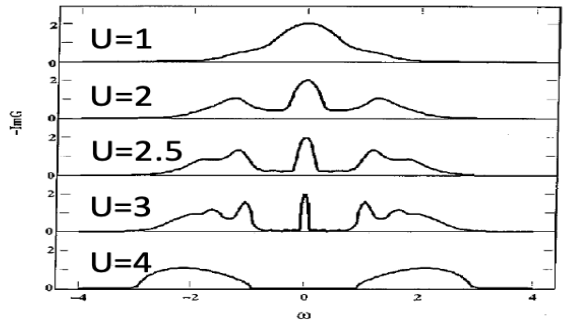
Metal insulator transition

Hubbard model

$$H = -\sum_{\langle ij\rangle,\sigma} t_{ij}(c^{+}_{i\sigma}c_{j\sigma} + c^{+}_{j\sigma}c_{i\sigma}) + U\sum_{i} n_{i\uparrow}n_{i\downarrow}$$







```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize
from scipy.integrate import simps
from ed import *
from dos import *
import warnings
warnings.filterwarnings("ignore")
```

numpy as np: This imports the NumPy library.

matplotlib.pyplot as plt: This imports the pyplot module from the matplotlib library.

from scipy.optimize import minimize: This imports the minimize function from the scipy.optimize module. minimize is used for minimizing functions numerically.

from scipy.integrate import simps: This imports the simps function from the scipy.integrate module. simps is used for numerical integration using Simpson's rule.

from ed import*: This imports all functions and classes from a module ed.

from dos import *: This imports all functions and classes from a module named dos.

import warnings: This imports the warnings module, which provides functions for issuing warnings from Python code.

warnings.filterwarnings("ignore"): This line sets up a filter to ignore all warnings generated by Python code. This can be useful when you have warnings that are not critical or expected.

```python
def get_G0_semicirc(omFs, mu, t):
    from dos import semicircle
    from scipy.integrate import simps
    ws = np.linspace(-2*t,2*t,1000)
    sc_dos = semicircle(ws,t)
    G0 = np.zeros(len(omFs),dtype=complex)
    for iomF, omF in enumerate(omFs):
        y = sc_dos/(1j*omF+mu-ws)
        G0[iomF] = simps(y, ws)
    return G0
```

Computing non-interacting Green's function for semicircle density of states :

- omFs: Frequencies at which the Green's function is evaluated.
- mu: Chemical potential.
- t: Half-bandwidth parameter of the semicircle DOS.

- **ws**: Generates an array ws of 1000 points linearly spaced between −2t and 2t. This represents the range of energies where the DOS is evaluated.
- **sc_dos**: Computes the semicircular DOS using the semicircle function imported earlier, evaluated at each point in ws with bandwidth parameter t.
- **G0**: Initializes an array G0 to store the Green's function values. The array is initialized with zeros and is complex-valued.
- **Integration Loop**:
- Iterates over each index iomF and corresponding frequency omF in the array omFs.
- y: Constructs an array y where each element is given by sc_dos / (1j * omF + mu - ws). This represents the integrand for the Green's function.
- G0[iomF]: Calculates the Green's function G0 for the frequency iomF using the simps function to perform numerical integration of y over ws.
- **Return**: Returns the array G0, which contains the computed Green's function values for each frequency in omFs.

$$G_o(\iota\omega_n) = \int d\epsilon \frac{D(\epsilon)}{\iota\omega_n + \mu - \epsilon}$$

```python
def get_G0_aim(omFs, V, eb, ed, mu):
    G0_aim = np.zeros(len(omFs),dtype=complex)
    for iomF, omF in enumerate(omFs):
        Delta = 0.0
        for ib in range(eb.shape[0]):
            Delta += (V[ib]*np.conj(V[ib])).real/(1j*omF-eb[ib])
        G0_aim[iomF] = 1./(1j*omF - ed + mu - Delta)
    return G0_aim
```

Computing Green's function for Anderson impurity model:

Parameters:

- omFs: Array-like. Frequencies at which the Green's function G0aim is evaluated.
- V: Hybridization function or coupling strengths between impurity and bath sites.
- eb: Energy levels of the bath sites.
- ed: Energy level of the impurity site.
- mu: Chemical potential.

G0_aim: Initializes an array G0_aim to store the Green's function values. The array is complex-valued and has the same length as omFs.

Nested Loop:

- Outer loop (for iomF, omF in enumerate(omFs)): Iterates over each index iomF and corresponding frequency omF in the array omFs.
- Delta: Initializes Delta to zero. It represents the self-energy term or hybridization function, calculated as a sum over all bath sites (eb.shape[0] gives the number of bath sites).
- Inner loop (for ib in range(eb.shape[0])):
  - Computes Delta by summing (V[ib] * np.conj(V[ib])).real / (1j * omF - eb[ib]). Here, V[ib] * np.conj(V[ib])).real computes the squared modulus of V[ib], which is the hybridization strength squared.
- Calculates G0_aim[iomF], the Green's function, using the formula $G0=1/(1j*omF-\epsilon d+\mu-\Delta)$

$$\mathcal{G}_0^{-1}(i\omega_n) = i\omega_n + \mu - \sum_{l=1}^{N_s} \frac{|V_l|^2}{i\omega_n - \varepsilon_l}$$

```python
def get_G0w_aim(ws, eta, V, eb, ed, mu):
    G0_aim = np.zeros((len(ws)),dtype=complex)
    for iw, w in enumerate(ws):
        Delta = 0.0
        for ib in range(eb.shape[0]):
            Delta += (V[ib]*np.conj(V[ib])).real/(w + 1j*eta - eb[ib])
        G0_aim[iw] = (1./(w + 1j*eta - ed + mu - Delta))[0]
    return G0_aim
```

Calculates a frequency-dependent Green's function G0(w) for an Anderson impurity model:

Parameters:

- ws: Frequencies w at which the Green's function G0(w) is evaluated.
- eta: Broadening parameter (smearing factor) to account for finite lifetimes or damping effects.
- V: Hybridization function or coupling strengths between impurity and bath sites.
- eb: Energy levels of the bath sites.
- ed: Energy level of the impurity site.
- mu: Chemical potential.

G0_aim: Initializes an array G0_aim to store the Green's function values. The array is complex-valued and has the same length as ws.

$$G_{o,AIM}(\omega) = \frac{1}{\omega + \iota\eta - \epsilon + \mu - \Delta}$$

Nested Loop:

- Outer loop (for iw, w in enumerate(ws)): Iterates over each index iw and corresponding frequency w in the array ws.
- Delta: Initializes Delta to zero. It represents the self-energy term or hybridization function, calculated as a sum over all bath sites (eb.shape[0] gives the number of bath sites).
- Inner loop (for ib in range(eb.shape[0])):
  - Computes Delta by summing (V[ib] * np.conj(V[ib])).real / (w + 1j * eta - eb[ib]). Here, V[ib] * np.conj(V[ib])).real computes the squared modulus of V[ib], which is the hybridization strength squared, divided by the frequency-dependent term.
- Calculates G0_aim[iw], the Green's function, using the formula $G0(w)=1/w+i\eta-\epsilon d+\mu-\Delta$, where η is the broadening parameter.

```python
def cost_func(x, *args):
    omFs, ed, mu, G0, Nb, Nmax = args
    V = x[:Nb] + 1j*x[Nb:2*Nb]
    eb = x[2*Nb:3*Nb]
    G0_aim = get_G0_aim(omFs, V, eb, ed ,mu)
    cost = 0.0
    for iomF in range(Nmax):
        diff = 1./G0_aim[iomF] - 1./G0[iomF]
        cost += (np.conj(diff)*diff).real/omFs[iomF]**1
    return cost
```

Cost function used for optimization, likely in the context of fitting:

Parameters:

- x: Parameters to be optimized.
- args: Tuple of additional arguments passed to the function, including:
  - omFs: Frequencies at which the Green's functions are evaluated.
  - ed: Energy level of the impurity site.
  - mu: Chemical potential.
  - G0: target Green's function values.
  - Nb: Number of bath sites.
  - Nmax: Number of frequencies to consider in the cost calculation.

Extract Parameters: Splits the optimization parameters x into:

- V: Array of complex numbers representing the coupling strengths between the impurity and bath sites.
- eb: Array of floats representing the energy levels of the bath sites.

$$|G_o^{-1} - G_{o,AIM}^{-1}| = |\Delta(\iota\omega_n) - \Delta_{imp}(\iota\omega_n)| = \left|\Delta(\iota\omega_n) - \sum_l \frac{|V_l|^2}{\iota\omega_n - \epsilon_l}\right|$$

G0_aim: Uses the get_G0_aim function to compute the theoretical Green's function G0_aim.

Calculate Cost: Initializes cost to zero and then computes the cost function by iterating over the first Nmax frequencies:

- diff: Computes the difference between the inverse of the model Green's function 1. / G0_aim[iomF] and the inverse of the experimental or target Green's function 1. / G0[iomF].
- cost: Accumulates the squared norm of diff, weighted by 1 / omFs[iomF] to emphasize lower frequencies

```python
np.random.seed(1234)
np.set_printoptions(suppress=True, precision=8)

# initial density of state
T = 0.005  #Temp of the system
t = 0.5    #Hopping parameter
mu = 0.0   #Chmeical potential
Nb = 3     #No. of bath levels
Nmax = 100  #Maximum Matsubara frequency to fit
mix = 0.5   #Mixing parameter used in iterative algorithms
maxit = 500  #maximum itteration for acheiving convergence

NomF = Nmax   #NomF: Numaber of Matsubara point
omFs = (2*np.arange(NomF)+1)*np.pi*T  #Generates Matsubara frequencies

G0 = get_G0_semicirc(omFs, mu, t)     #to compute the initial Green's function G0
# Exact-diagonlization intialization
no = 2*(1+Nb)        #Calculates the total number of spin and orbitals
print('no=',no)
FH_list = build_fermion_op(no)    #construct a list of fermion operators corresponding to the total number of orbital
print(len(FH_list))
FH_list_dense = [np.array(FH.todense(),dtype=complex) for FH in FH_list]
# Bath fitting parameter intialization
ebs = np.random.uniform(-1,1,Nb)    #Generates random energy levels for the bath levels.
Vrs = np.random.uniform(-0.5,0.5,Nb)  #Generates random real parts of hybridization parameters for the bath levels.
Vis = np.zeros((Nb))                 #Initializes imaginary parts of hybridization parameters
x0 = np.hstack((Vrs,Vis,ebs))       #Creates an initial vector x0 that combines the real parts of hybridization, imag

U = 1.5  # Interaction parameter
```

```
it = 0   #Itteration counter
diff = 1e20  # Initialize difference variable
while diff > 5e-5 and it<maxit:                      # Iterative loop to converge on parameters
    print("----------------------------------- it=%d U=%.2f -----------------------------------"%(it, U))

    # fit V eb
    args = omFs, 0.0, mu, G0, Nb, Nmax
    result = minimize(cost_func,x0,args=args, method='L-BFGS-B', options={'gtol': 1e-2, 'eps': 1e-12})
    print("GA root convergence message--------------------------------")
    print("sucess=",result.success)
    print(result.message)
    V = result.x[:Nb] + 1j*result.x[Nb:2*Nb]  #extracts the hybridization matrix elements V, representing the coupling strength between the impurity site and the bath levels.
    eb = result.x[2*Nb:3*Nb]    #line extracts the energy levels eb, which represent the discrete energy levels of the impurity site.
    x0 = result.x #Updates the initial guess x0 with the optimized parameter values for the next iteration.

    print('V=',V)
    print('eb=', eb)

    # ED part
    h1 = np.array([[-U/2+mu, 0.0],  #the local part of the Hamiltonian matrix h1 for the impurity site.
                   [ 0.0,-U/2+mu]])
    eb = np.kron(eb,np.ones((2)))   #Expands the energy levels eb to the full single-particle space by duplicating each energy level twice
    V = np.kron(V,np.eye(2)).T      #Expands the hybridization matrix V to the full single-particle space using the Kronecker product with the identity matrix np.eye(2)

    V2E = np.zeros((2,2,2,2))  #Initializes the two-particle interaction matrix V2E with zeros.
    V2E[1,1,0,0] = U           #Sets the matrix elements of V2E corresponding to the interaction strength U
    V2E[0,0,1,1] = U
    print('V(ED)=',V)
    print('eb(ED)=', eb)
        #Solving the Impurity Hamiltonian
    dm, evals, evecs, docc = solve_Hemb_thermal(T, h1, V, eb, V2E, FH_list, verbose=0)
```

```
    print('dm=')
    print(dm.real)
    print('trace(dm)=',np.trace(dm).real)
    print('docc=', docc.real)
    #Computing Matsubara Green's Function
    GomF = compute_GomF_thermal(T, omFs, evals, evecs, FH_list_dense)

    # update G0
    G0_new = 1./(1j*omFs + mu - t**2*GomF)
    # Difference check convergence of Go
    diff = np.sum(np.abs(G0_new  -G0))
    print('diff=', diff)
    #Mixing solution
    it += 1
    G0 = (1-mix)*G0 + mix*G0_new

# real frequency quantities
ws = np.linspace(-5,5,200) # frequency mesh
eta = 0.1 #broadening factor
Gw = compute_Gw_thermal(T, ws, eta, evals, evecs, FH_list_dense) # real frequency impurity Green's function
G0w = get_G0w_aim(ws, eta, V[::2,::2], eb[::2], 0.0, mu) # real frequency non-interacting impurity Green's function
Sigw = 1./G0w - 1./Gw # real frequency self-energy
Glattw = compute_Glattw_semcircle(ws,eta,Sigw,mu) # real frequency lattice Green's function
# Matsubara self-energy
G0 = get_G0_aim(omFs, V[::2,::2], eb[::2], 0.0, mu) # matsubara frequency impurity Green's function
Sig = 1./G0 - 1./GomF # Matsubara frequency self-energy
Z = 1./(1-(Sig[0].imag)/(omFs[0])) # quasiparticle weight estimate from Matsubara self-energy
Glatt = compute_GlattomF_semcircle(omFs,Sig,mu) # real frequency lattice Green's function
```

Hand's on Session

- Metal GF, self-energy U=1.5
- Mott GF, self-energy U=4.0
- Metal-Insulator first order transition, T=0.005
- (Optional) U-T metal to insulator transition phase diagram

U-T Metal to Insulator Transition Phase Diagram