Fast renormalizing the structures and dynamics of ultra-large systems via random renormalization group (supplementary material)*

Yang Tian†
Infplane AI Technologies Ltd, Beijing, 100080, China
Laboratory of Computational Biology and Complex Systems,
City University of Macau, Macau, 999078, China
Faculty of Health and Wellness, City University of Macau, Macau, 999078, China and
Faculty of Data Science, City University of Macau, Macau, 999078, China

Yizhou Xu‡
Infplane AI Technologies Ltd, Beijing, 100080, China
École Polytechnique Fédérale de Lausanne, Lausanne, 1015, Switzerland and
International Center of Theoretical Physics, Trieste, I-34151, Italy.

Pei Sun§
Laboratory of Computational Biology and Complex Systems,
City University of Macau, Macau, 999078, China and
Faculty of Health and Wellness, City University of Macau, Macau, 999078, China

This is the supplementary material of the paper entitled as "Fast renormalizing the structures and dynamics of ultra-large systems via random renormalization group". In Sec. I, we introduce the code implementation of the RRG program and present instances of its usage. In Sec. II, we present the code for analyzing macroscopic observables and scaling behaviours.

I. CODE IMPLEMENTATION OF THE RRG

The RRG is programed in Python, whose open-source code can be seen in https://github.com/Asuka-Research-Group/Random-renormalization-group and used for research. The RRG depends on several external libraries listed below. Users should prepare these libraries before using the RRG.

A. Environment preparation

```python
## Dependency libraries used for the RRG:
import networkx as nx
import faiss
import time
import scipy as spy
from datasketch import MinHash
import copy

## Dependency libraries used for the scaling analysis:
from scipy.optimize import curve_fit
import statsmodels.api as sm
from scipy.stats import ks_2samp
```

Among these libraries, some users who prefer to use CPU for computation may meet difficulties in installing faiss via pip. This is a common problem faced by the faiss environment. The following conda-based command may help resolve the problem in most cases

```
conda install -c conda-forge faiss
```

---

* Correspondence should be addressed to Yizhou Xu and Pei Sun.
† tyanyang04@gmail.com
‡ yizhou.xu.cs@gmail.com
§ peisun@cityu.edu.mo

B. Main function and usage of the RRG framework

In application, we have a system, X, to process. We denote X_Initial as X in the program. For structure renormalization, we need to ensure that X_Initial is a graph object in the networkx library. For dynamics renormalization, X_Initial is expected as an array in the numpy, where each row corresponds to the dynamics of one unit.

To run the RRG for T iterations, we let Iteration_Num be T. Meanwhile, we set TargetDim as h to make each hashed binary vector $Z_i^{(l)}$ have a dimension of h. To chose the signed random hyperplane projection [1], the signed random Fourier feature [2, 3], or the signed Cauchy projection [4], we need to set Method_Type as Linear_Kernel, Gaussian_Kernel, or Cauchy_Kernel, respectively. Finally, the inform the program about the data type, we set Data_Type as Structure or Dynamics to start structure or dynamics renormalization.

```python
def Renormalization_Flow(X_Initial,Iteration_Num,TargetDim,Method_Type,Data_Type):
    RG_Flow=[]
    RG_Flow.append(X_Initial)
    Corase_ID_list=[]
    for Iter in range(Iteration_Num):
        StartT=time.time()
        X_Current=RG_Flow[Iter]
        if Data_Type=="Dynamics":
            X_New, Corase_ID=Renormalization_Function(X_Current,TargetDim,Iter,Method_Type)
        elif Data_Type=="Structure":
            X_New, Corase_ID=Network_Renormalization_Function(X_Current,TargetDim,Iter,Method_Type)
            if nx.number_of_edges(X_New)==0:
                break
        RG_Flow.append(X_New)
        Corase_ID_list.append(Corase_ID)
        EndT=time.time()
        print(['The', Iter+1, 'time of renormalization costs-', EndT-StartT])
    Tracked_ID_list=Tracking_System(Corase_ID_list)
    return RG_Flow,Tracked_ID_list
```

The main function of the RRG generates two outputs after computation. The first one is RG_Flow, the list of system X on different scales. For instance, the first element of RG_Flow is $X = X^{(1)}$, the second one is $X^{(2)}$, and so on. The number of elements in RG_Flow is determined by both Iteration_Num and system properties (i.e., the RRG stops iteration when there remain only one unit). The data types of all elements of RG_Flow keep the same as X.

The second output of the main function is Tracked_ID_list, which is used to indicate the indexes of the initial units aggregated into each macro-unit after every iteration of the RRG. Below, we present a simple instance where system X contains only six units

```python
Tracked_ID_list[0]=[[0,1],[2],[3,5],[4]]
Tracked_ID_list[1]=[[0,1,2],[3,5],[4]]
Tracked_ID_list[2]=[[0,1,2,4],[3,5]]
```

Before renormalization, each macro-unit only contains itself, which is represented by a list [[0],[1],[2],[3],[4],[5]] (note that this trivial list is not included in Tracked_ID_list for convenience). This list contains six lists as its elements, where the i-th element contains the indexes of initial units aggregated into the i-th macro-unit. As shown in the instance above, the first element of Tracked_ID_list is [[0,1],[2],[3,5],[4]], which means that there remain four macro-units after the first time of renormalization. The first macro-unit is formed by two initial units whose indexes are 0 and 1. The second element of Tracked_ID_list is [[0,1,2],[3,5],[4]], suggesting that there are three macro-units after two times of renormalization. The first macro-units contains three initial units whose indexes are 0, 1, and 2. Other elements of Tracked_ID_list can be understood in a similar way.

To run the RRG, one can consider the following instances:

```python
## Structure renormalization
X_Initial=nx.random_tree(10000) # Generate a random tree with 10000 units

RG_Flow,Tracked_ID_list=Renormalization_Flow(X_Initial,100,50,"Linear_Kernel","Structure") # Run a RRG for 100 iterations, where the dimension of hased binary vectors is 50

RG_Flow,Tracked_ID_list=Renormalization_Flow(X_Initial,50,10,"Gaussian_Kernel","Structure") # Run a RRG for 50 iterations, where the dimension of hased binary vectors is 10

RG_Flow,Tracked_ID_list=Renormalization_Flow(X_Initial,200,100,"Cauchy_Kernel","Structure") # Run a RRG for 200 iterations, where the dimension of hased binary vectors is 100

## Dynamics renormalization
```

```python
X_Initial = np.random.randn(10000, 50000) # Generate a system with 10000 units, where each unit exhibits random dynamics for 50000 time steps

RG_Flow,Tracked_ID_list=Renormalization_Flow(X_Initial,100,50,"Linear_Kernel","Dynamics") # Run a RRG for 100 iterations, where the dimension of hased binary vectors is 50

RG_Flow,Tracked_ID_list=Renormalization_Flow(X_Initial,50,10,"Gaussian_Kernel","Dynamics") # Run a RRG for 50 iterations, where the dimension of hased binary vectors is 10

RG_Flow,Tracked_ID_list=Renormalization_Flow(X_Initial,200,100,"Cauchy_Kernel","Dynamics") # Run a RRG for 200 iterations, where the dimension of hased binary vectors is 100
```

C. Full code implementation

For convenience, we attach the full code implementation below. One can also see https://github.com/Asuka-Research-Group/Random-renormalization-group for the official release of our framework, where we provide instances in the Jupyter notebook.

```python
def Random_Fourier_Feature_Hashing(X,TargetDim):
    N = np.size(X,0)
    d = np.size(X,1)
    W = np.random.normal(loc=0, scale=1, size=(d, TargetDim))
    b = np.random.uniform(0, 2*np.pi, size=TargetDim)
    B = np.repeat(b[:, np.newaxis], N, axis=1).T
    Z = 1/2* (1+ np.sign(np.cos(X @ W + B)))
    Z = np.uint8(Z)
    return Z

def Random_Cauchy_Feature_Hashing(X,TargetDim):
    N = np.size(X,0)
    d = np.size(X,1)
    W = spy.stats.cauchy.rvs(loc=0, scale=1, size=(d, TargetDim))
    b = np.random.uniform(0, 2*np.pi, size=TargetDim)
    B = np.repeat(b[:, np.newaxis], N, axis=1).T
    Z = 1/2* (1+ np.sign(np.cos(X @ W + B)))
    Z = np.uint8(Z)
    return Z

def Random_Hyperplane_Hashing(X,TargetDim):
    d = np.size(X,1)
    W = np.random.normal(loc=0, scale=1, size=(d, TargetDim))
    Z = 1/2* (1+ np.sign(X @ W))
    Z = np.uint8(Z)
    return Z

def Random_Min_Hashing(X,TargetDim):
    Z=np.zeros((len(X),TargetDim))
    for ID1 in range(len(X)):
        Hashing_Code=MinHash(num_perm=TargetDim)
        Hashing_Code.update_batch(X[ID1])
        Z[ID1,:]=Hashing_Code.hashvalues
    return Z

def Neighbor_Generator(X,UnitNum):
    Y=[]
    for Unit in range(UnitNum):
        Neighbors = [Unit] + list(X.neighbors(Unit))
        Y.append(np.array(Neighbors))
    return Y


def Normalization_Function(X_Current,Method_Type):
    if Method_Type=="Linear_Kernel":
        Normalized_X=X_Current-np.mean(X_Current,axis=1).reshape(np.size(X_Current,0),1)
    elif Method_Type=="Gaussian_Kernel":
        Normalized_X=X_Current-np.mean(X_Current,axis=1).reshape(np.size(X_Current,0),1)
        Std=np.std(Normalized_X,axis=1).reshape(np.size(Normalized_X,0),1)
        Normalized_X=np.divide(Normalized_X,Std,out=Normalized_X,where=Std!=0)
```

```
    elif Method_Type=="Cauchy_Kernel":
        Normalized_X=X_Current-np.min(X_Current,axis=1).reshape(np.size(X_Current,0),1)
        SumV=np.sum(Normalized_X,axis=1).reshape(np.size(Normalized_X,0),1)
        Normalized_X=np.divide(Normalized_X,SumV,out=Normalized_X,where=SumV!=0)
    return Normalized_X

def Binary_Hashing_Index(Z):
    if np.size(Z,0)<=50000:
        Dim=8*np.size(Z,1)
        Index = faiss.IndexBinaryFlat(Dim)
        Index.nprobe = 2
    elif (np.size(Z,0)>50000)&(np.size(Z,0)<=500000):
        Dim=8*np.size(Z,1)
        Index = faiss.IndexBinaryHash(Dim,Dim)
        Index.nprobe = 2
    elif np.size(Z,0)>500000:
        Dim=8*np.size(Z,1)
        Index = faiss.IndexBinaryHash(Dim,int(np.max([np.min([np.ceil(Dim/100),32]),16])))
        Index.nprobe = 2
    return Index


def KNN_with_Hashing_Index(Z):
    StartT=time.time()
    Index=Binary_Hashing_Index(Z)
    Index.add(Z)
    Num_neighbors=2
    D, I = Index.search(Z, Num_neighbors)
    EndT=time.time()
    print(['KNN search costs-', EndT-StartT])
    return D,I

def Hashing_Function(Normalized_X,TargetDim,Method_Type):
    if Method_Type=="Linear_Kernel":
        Z=Random_Hyperplane_Hashing(Normalized_X,TargetDim)
    elif Method_Type=="Gaussian_Kernel":
        Z=Random_Fourier_Feature_Hashing(Normalized_X,TargetDim)
    elif Method_Type=="Cauchy_Kernel":
        Z=Random_Cauchy_Feature_Hashing(Normalized_X,TargetDim)
    return Z

def Renormalization_Function(X_Current,TargetDim,Iter,Method_Type):
    Normalized_X=Normalization_Function(X_Current,Method_Type)
    Z=Hashing_Function(Normalized_X,TargetDim,Method_Type)
    _,I=KNN_with_Hashing_Index(Z)
    G = nx.empty_graph(np.size(I,0))
    Edge = np.vstack((np.arange(0, np.size(I, 0)), I[:,1])).T
    G.add_edges_from(Edge)
    Clusters=[list(c) for c in list(nx.connected_components(G))]
    ClusterNum=nx.number_connected_components(G)
    print(['There are', ClusterNum, 'macro-units after', Iter+1, 'times of renormalization'])
    X_New=np.zeros((ClusterNum, np.size(X_Current,1)))
    Corase_ID = []
    for ID1 in range(ClusterNum):
        X_New[ID1,:]=np.sum(X_Current[Clusters[ID1],:],axis=0)
        Corase_ID.append(Clusters[ID1])
    return X_New, Corase_ID

def Network_Renormalization_Function(X_Current,TargetDim,Iter,Method_Type):
    UnitNum=nx.number_of_nodes(X_Current)
    Y=Neighbor_Generator(X_Current,UnitNum)
    Z=Random_Min_Hashing(Y,TargetDim)
    Z=Hashing_Function(Z,TargetDim,Method_Type)
    _,I=KNN_with_Hashing_Index(Z)
    G = nx.empty_graph(np.size(I,0))
    Edge = np.vstack((np.arange(0, np.size(I, 0)), I[:,1])).T

    G.add_edges_from(Edge)
    Potential_Clusters=[list(c) for c in list(nx.connected_components(G))]
    Potential_ClusterNum=nx.number_connected_components(G)
```

```python
    Edge_To_Remove=[]
    for ID1 in range(Potential_ClusterNum):
        Unit_list=Potential_Clusters[ID1]
        if len(Unit_list)>1:
            H = nx.induced_subgraph(X_Current,Unit_list)
            Potential_H = nx.induced_subgraph(G,Unit_list)
            Wrong_Edge=list(set(list(Potential_H.edges))-set(list(H.edges)))
            Edge_To_Remove.extend(Wrong_Edge)

    for Wrong_Edge in Edge_To_Remove:
        G.remove_edge(*Wrong_Edge)

    Clusters=[list(c) for c in list(nx.connected_components(G))]
    ClusterNum=nx.number_connected_components(G)
    print(['There are', ClusterNum, 'macro-units after', Iter+1, 'times of renormalization'])


    X_New=copy.deepcopy(X_Current)
    Pre_Corase_ID = []
    Mappings={}
    for ID1 in range(ClusterNum):
        Unit_list=Clusters[ID1]
        Pre_Corase_ID.append(Unit_list)
        Unit0 = Unit_list[0]
        Mappings[Unit0]=ID1
        for Unit in Unit_list[1:]:
            if X_New.has_node(Unit):
                Neighbors = list(X_New.neighbors(Unit))
                New_edges = [(Unit0, Nei) for Nei in Neighbors if Unit0!=Nei]
                X_New.add_edges_from(New_edges)
                X_New.remove_node(Unit)
    Corase_ID = []
    Unit_Mappings={}
    for ID_1,ID_2 in enumerate(X_New.nodes()):
        Unit_Mappings[ID_2]=ID_1
        Corase_ID.append(Pre_Corase_ID[Mappings[ID_2]])
    X_New = nx.relabel_nodes(X_New, Unit_Mappings)

    return X_New, Corase_ID

def Tracking_System(Corase_ID_list):
    Tracked_ID_list = []
    for IterID in range(len(Corase_ID_list)):
        if IterID==0:
            Tracked_ID_list.append(Corase_ID_list[0])
        else:
            Tracked_ID = []
            if len(Corase_ID_list[IterID])>0:
                for CoarseID in range(len(Corase_ID_list[IterID])):
                    UnitsToTrack=Corase_ID_list[IterID][CoarseID]
                    Searched_ID=[]
                    for IDSearch in range(len(UnitsToTrack)):
                        Search_ID=1
                        while len(Tracked_ID_list[IterID-Search_ID])==0:
                            Search_ID=Search_ID+1
                        Searched_ID=Searched_ID+Tracked_ID_list[IterID-Search_ID][UnitsToTrack[
    IDSearch]]
                    Tracked_ID.append(Searched_ID)
            Tracked_ID_list.append(Tracked_ID)
    return Tracked_ID_list

def Renormalization_Flow(X_Initial,Iteration_Num,TargetDim,Method_Type,Data_Type):
    RG_Flow=[]
    RG_Flow.append(X_Initial)
    Corase_ID_list=[]
    for Iter in range(Iteration_Num):
        StartT=time.time()
        X_Current=RG_Flow[Iter]
        if Data_Type=="Dynamics":
            X_New, Corase_ID=Renormalization_Function(X_Current,TargetDim,Iter,Method_Type)
```

```
        elif Data_Type=="Structure":
            X_New, Corase_ID=Network_Renormalization_Function(X_Current,TargetDim,Iter,Method_Type)
            if nx.number_of_edges(X_New)==0:
                break
        RG_Flow.append(X_New)
        Corase_ID_list.append(Corase_ID)
        EndT=time.time()
        print(['The', Iter+1, 'time of renormalization costs-', EndT-StartT])
    Tracked_ID_list=Tracking_System(Corase_ID_list)
    return RG_Flow,Tracked_ID_list
```

II. CODE IMPLEMENTATION OF MACROSCOPIC OBSERVABLES AND SCALING ANALYSIS

After obtaining a renormalization flow, we can analyze macroscopic observables and scaling behaviours. Below, we elaborate the code implementation of these analyses.

A. Structure renormalization

For structure renormalization, we can run the following function to derive the mean Kolmogorov–Smirnov static [5, 6]

```
Mean_K_S_Static=KS_Analysis(RG_Flow)
```

The output Mean_K_S_Static is a scalar that reports the mean Kolmogorov–Smirnov static. The full code of this function is present below

```
def KS_Analysis(RG_Flow):
    K_S_Static=np.zeros(len(RG_Flow))
    Degrees_0=[Node[1] for Node in list(nx.degree(RG_Flow[0]))]
    for InterID in range(len(RG_Flow)):
        Degrees=[Node[1] for Node in list(nx.degree(RG_Flow[InterID]))]
        KstestResult=ks_2samp(Degrees, Degrees_0, alternative='two-sided',method='exact')
        K_S_Static[InterID]=KstestResult[0]*(KstestResult[1]<0.01)

    Mean_K_S_Static=np.mean(K_S_Static)
    return Mean_K_S_Static
```

where the Degrees generated from each element of RG_Flow can be further used to derive the degree distribution after frequency counting (e.g., using the histogram function of the numpy).

B. Dynamics renormalization

For dynamics renormalization, we can use the following function to derive the normalized dynamics

```
Cut_Off_Ratio=0.1
Normalized_activity=Normalized_Dynamics(RG_Flow,Tracked_ID_list,Cut_Off_Ratio)
```

where Cut_Off_Ratio denotes the fraction of eigenvalues to keep. The output Normalized_activity is a list of arrays, where each element is the normalized dynamics of the system on a certain scale. The probability distribution of normalized dynamics can be derived using frequency counting (e.g., using the histogram function of the numpy).

The full code implementation of the above function is

```
def Normalized_Dynamics(RG_Flow,Tracked_ID_list,Cut_Off_Ratio):
    ClusterNum=np.array([len(Tracked_ID_list[ID1]) for ID1 in range(1,len(Tracked_ID_list))])
    Max_Range=np.max(np.where(ClusterNum>1)[0])+1
    for IterID in range(Max_Range):
        X_Current=RG_Flow[IterID]
        N=np.size(X_Current,0)
        Covariance = np.cov(X_Current)
        Evals, U = np.linalg.eig(Covariance)
        Idx = Evals.argsort()[::-1]
        EigenValues = Evals[Idx]
        EigenVectors = U[:,Idx]
        k=int(np.round(N*Cut_Off_Ratio))
```

```
        P=EigenVectors[:,:k] @ EigenVectors[:,:k].T
        phi=P@(X_Current-np.mean(X_Current,axis=1,keepdims=True))
        Normalized_activity=phi/np.std(phi,axis=1,keepdims=True)
    return Normalized_activity
```

Moreover, we can carry out scaling analyses using the following commands

```
MeanClusterSize, MeanVar, Coeff, Alpha, R2, MSE, Esti_Alpha_Scaling = Alpha_Scaling(RG_Flow,
    Tracked_ID_list)

MeanClusterSize, FreeEV, Coeff, Beta, R2, MSE, Esti_Beta_Scaling = Beta_Scaling(RG_Flow,
    Tracked_ID_list)

Average_Rank_K, Average_Evals, Coeff, Mu, R2, MSE, Esti_Mu_Scaling = Mu_Scaling(RG_Flow,
    Tracked_ID_list)

ScaledT, MeanACFs, MeanClusterSize, Tau, Coeff, Theta, R2, MSE, Esti_Theta_Scaling = Theta_Scaling(
    RG_Flow,Tracked_ID_list)
```

Among the outputs of Alpha_Scaling function, MeanClusterSize stands for the sequences of $\langle K^{(l)}\rangle$ and MeanVar stands for the sequences of Var $\left(\langle K^{(l)}\rangle\right)$. Coeff and Alpha denote the coefficient and exponent α of the fitted model, whose fitting accuracy can be reflected by R2 and MSE. The estimated trend of Var $\left(\langle K^{(l)}\rangle\right)$ is contained by Esti_Alpha_Scaling.

In the outputs of Beta_Scaling function, FreeEV denotes the sequence of $F\left(\langle K^{(l)}\rangle\right)$. Beta is the exponent β of the estimated model. Esti_Beta_Scaling is the estimated sequence of $F\left(\langle K^{(l)}\rangle\right)$ by the model.

The outputs of Mu_Scaling function include Average_Rank_K, the sequence of $r/\langle K^{(l)}\rangle$, and Average_Evals, the sequence of λ_r. Meanwhile, it contains Mu, the exponent mu of the fitted model, and Esti_Mu_Scaling, the predicted trend of λ_r.

The Theta_Scaling function first generate ScaledT and MeanACFs, the sequences of re-scaled time and mean auto-correlation functions that can be used to visualize the universal collapse. Then, its output contains MeanClusterSize and Tau, the sequences of $\langle K^{(l)}\rangle$ and τ_c that can be used to fit dynamic scaling. Theta and Esti_Theta_Scaling denote the fitted exponent θ and its corresponding model.

The full code implementation of the above functions are shown below

```
## Analysis
def Linear_func(x, a, b):
    return b*x+a

def Power_func(x, a):
    return a*x


def RSquareFun(X,y,popt):
    if len(popt)==2:
        pre_y = Linear_func(X, popt[0], popt[1])
    elif len(popt)==1:
        pre_y = Power_func(X, popt[0])
    mean = np.mean(y)
    ss_tot = np.sum((y - mean) ** 2)
    ss_res = np.sum((y - pre_y) ** 2)
    r_squared = 1 - (ss_res / ss_tot)

    mse = np.sum((y - pre_y) ** 2)/ len(y)
    return r_squared, mse

def Alpha_Scaling(RG_Flow,Tracked_ID_list):
    MeanVar=np.zeros(len(RG_Flow))
    for Iter in range(len(RG_Flow)):
        X=RG_Flow[Iter]
        MeanVar[Iter]=np.mean(np.var(X,axis=1))

    MeanClusterSize=np.ones(len(RG_Flow))
    for Iter in range(len(Tracked_ID_list)):
        ClusterSize=[len(IDC) for IDC in Tracked_ID_list[Iter]]
        MeanClusterSize[Iter+1]=np.mean(ClusterSize)

    popt, _ = curve_fit(Linear_func, np.log(MeanClusterSize), np.log(MeanVar))
    Coeff = popt[0]
```

```
    Alpha = popt[1]
    R2, MSE= RSquareFun(np.log(MeanClusterSize), np.log(MeanVar), popt)
    Esti_Alpha_Scaling=np.exp(Coeff)*np.power(MeanClusterSize,Alpha)
    return MeanClusterSize, MeanVar, Coeff, Alpha, R2, MSE, Esti_Alpha_Scaling

def Beta_Scaling(RG_Flow,Tracked_ID_list):
    FreeEV=np.zeros(len(RG_Flow))
    for Iter in range(len(RG_Flow)):
        X=RG_Flow[Iter]

        P_SilenceV=np.zeros(np.size(X,0))
        for ID1 in range(np.size(X,0)):
            P_SilenceV[ID1] = 1-np.count_nonzero(X[ID1,:]) / np.size(X,1)
        P_Silence=np.mean(P_SilenceV)
        FreeEV[Iter]=-1*np.log(P_Silence)

    MeanClusterSize=np.ones(len(RG_Flow))
    for Iter in range(len(Tracked_ID_list)):
        ClusterSize=[len(IDC) for IDC in Tracked_ID_list[Iter]]
        MeanClusterSize[Iter+1]=np.mean(ClusterSize)

    Needed=np.where(np.isinf(FreeEV)==0)[0]
    FreeEV=FreeEV[Needed]
    MeanClusterSize=MeanClusterSize[Needed]

    popt, _ = curve_fit(Linear_func, np.log(MeanClusterSize), np.log(FreeEV))
    Coeff = popt[0]
    Beta = popt[1]
    R2, MSE= RSquareFun(np.log(MeanClusterSize), np.log(FreeEV), popt)
    Esti_Beta_Scaling=np.exp(Coeff)*np.power(MeanClusterSize,Beta)
    return MeanClusterSize, FreeEV, Coeff, Beta, R2, MSE, Esti_Beta_Scaling

def Mu_Scaling(RG_Flow,Tracked_ID_list):
    Initial_X = RG_Flow[0]
    Average_Rank_K=[]
    Average_Evals=[]

    ClusterNum=np.array([len(Tracked_ID_list[ID1]) for ID1 in range(1,len(Tracked_ID_list))])
    Max_Range=np.max(np.where(ClusterNum>1)[0])+2
    for ID1 in range(1,Max_Range):
        x=[]
        y=[]
        for ID2 in range(len(Tracked_ID_list[ID1])):
            WithinCluster= Tracked_ID_list[ID1][ID2]
            X_WC = Initial_X[WithinCluster,:]
            X_WC=X_WC-np.mean(X_WC,axis=1).reshape(np.size(X_WC,0),1)
            Cov=np.cov(X_WC)
            Evals, _ = np.linalg.eig(Cov)
            Evals = np.sort(np.real(Evals))
            Evals = Evals[::-1]

            Rank = np.cumsum(np.ones(len(Evals)))
            Rank_K=Rank/len(WithinCluster)

            Needed_Loc=np.where(Evals>0)[0]
            Rank_K=Rank_K[Needed_Loc]
            Evals=Evals[Needed_Loc]
            x.extend(Rank_K[:])
            y.extend(Evals[:])

        _, bins = np.histogram(x)
        Meanx=np.zeros(len(bins)-1)
        Meany=np.zeros(len(bins)-1)
        for ID3 in range(len(bins)-1):
            Neededx=np.where((x>=bins[ID3])&(x<=bins[ID3+1]))[0]
            Meanx[ID3]=np.mean(np.array(x)[Neededx])
            Meany[ID3]=np.mean(np.array(y)[Neededx])
        Average_Rank_K.extend(Meanx)
        Average_Evals.extend(Meany)

```

```
    popt, _ = curve_fit(Linear_func, np.log(Average_Rank_K), np.log(Average_Evals))
    Coeff = popt[0]
    Mu = -1* popt[1]
    R2, MSE= RSquareFun(np.log(Average_Rank_K), np.log(Average_Evals), popt)
    Esti_Mu_Scaling=np.exp(Coeff)*np.power(Average_Rank_K,-1* Mu)
    return Average_Rank_K, Average_Evals, Coeff, Mu, R2, MSE, Esti_Mu_Scaling

def Theta_Scaling(RG_Flow,Tracked_ID_list):
    Tau=np.zeros(len(RG_Flow))
    ScaledT=[]
    MeanACFs=[]

    for Iter in range(len(RG_Flow)):
        X=RG_Flow[Iter]
        SumAC = np.sum(X, axis=1)

        ACFMatrix = np.zeros_like(X)
        for ID1 in range(np.size(X,0)):
            ACFMatrix[ID1,:] = sm.tsa.acf(X[ID1,:], nlags=np.size(X,1))
        ACFMatrix = ACFMatrix[np.where(SumAC>0)[0],:]
        MeanACF = np.mean(ACFMatrix, axis=0)
        T = np.cumsum(np.ones(np.size(X,1)))-1

        Needed_ACF=np.where(MeanACF>0)[0]
        MeanACF=MeanACF[Needed_ACF]
        T=T[Needed_ACF]

        Cut_Off=int(np.max([np.ceil(0.01*len(T)),100]))
        popt, _ = curve_fit(Power_func, T[:Cut_Off], np.log(MeanACF[:Cut_Off]))
        Tau[Iter] = -1/popt[0]

        ScaledT.append(T/Tau[Iter])
        MeanACFs.append(MeanACF)

    MeanClusterSize=np.ones(len(RG_Flow))
    for Iter in range(len(Tracked_ID_list)):
        ClusterSize=[len(IDC) for IDC in Tracked_ID_list[Iter]]
        MeanClusterSize[Iter+1]=np.mean(ClusterSize)

    popt, _ = curve_fit(Linear_func, np.log(MeanClusterSize), np.log(Tau))
    Coeff = popt[0]
    Theta = popt[1]
    R2, MSE= RSquareFun(np.log(MeanClusterSize), np.log(Tau), popt)
    Esti_Theta_Scaling=np.exp(Coeff)*np.power(MeanClusterSize,Theta)

    return ScaledT, MeanACFs, MeanClusterSize, Tau, Coeff, Theta, R2, MSE, Esti_Theta_Scaling

def KS_Analysis(RG_Flow):
    K_S_Static=np.zeros(len(RG_Flow))
    Degrees_0=[Node[1] for Node in list(nx.degree(RG_Flow[0]))]
    for InterID in range(len(RG_Flow)):
        Degrees=[Node[1] for Node in list(nx.degree(RG_Flow[InterID]))]
        KstestResult=ks_2samp(Degrees, Degrees_0, alternative='two-sided',method='exact')
        K_S_Static[InterID]=KstestResult[0]*(KstestResult[1]<0.01)

    Mean_K_S_Static=np.mean(K_S_Static)
    return Mean_K_S_Static

def Normalized_Dynamics(RG_Flow,Tracked_ID_list,Cut_Off_Ratio):
    ClusterNum=np.array([len(Tracked_ID_list[ID1]) for ID1 in range(1,len(Tracked_ID_list))])
    Max_Range=np.max(np.where(ClusterNum>1)[0])+1
    for IterID in range(Max_Range):
        X_Current=RG_Flow[IterID]
        N=np.size(X_Current,0)
        Covariance = np.cov(X_Current)
        Evals, U = np.linalg.eig(Covariance)
        Idx = Evals.argsort()[::-1]
        EigenValues = Evals[Idx]
        EigenVectors = U[:,Idx]
        k=int(np.round(N*Cut_Off_Ratio))
```

```
        P=EigenVectors[:,:k] @ EigenVectors[:,:k].T
        phi=P@(X_Current-np.mean(X_Current,axis=1,keepdims=True))
        Normalized_activity=phi/np.std(phi,axis=1,keepdims=True)
    return Normalized_activity
```

[1] S. S. Vempala, *The random projection method*, Vol. 65 (American Mathematical Soc., 2005).
[2] X. Li and P. Li, Signrff: Sign random fourier features, Advances in Neural Information Processing Systems **35**, 17802 (2022).
[3] A. Rahimi and B. Recht, Random features for large-scale kernel machines, Advances in neural information processing systems **20** (2007).
[4] P. Li, G. Samorodnitsk, and J. Hopcroft, Sign cauchy projections and chi-square kernel, Advances in Neural Information Processing Systems **26** (2013).
[5] R. Simard and P. L'Ecuyer, Computing the two-sided kolmogorov-smirnov distribution, Journal of Statistical Software **39**, 1 (2011).
[6] V. W. Berger and Y. Zhou, Kolmogorov–smirnov test: Overview, Wiley statsref: Statistics reference online (2014).

ACKNOWLEDGEMENTS

Yang Tian and Yizhou Xu are supported by the High Performance Scientific Computing Research Program at Infplane AI Technologies Ltd. Pei Sun is supported by the Artificial and General Intelligence Research Program of Guo Qiang Research Institute at Tsinghua University (2020GQG1017). Authors appreciate Hedong Hou at the Institut de Mathématiques d'Orsay and Aohua Cheng at Tsinghua University for their inspiring discussions.