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

Yang Tian,† Yizhou Xu,‡ and Pei Sun§

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 validate the ability of the RRG to classify different random network models according to scale-invariance property.

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

```
## 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
```

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
```

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.

```
def Renormalization_Flow(X_Initial,Iteration_Num,TargetDim,Method_Type,Data_Type):
    RG_Flow=[]
    RG_Flow.append(X_Initial)
```

* Correspondence should be addressed to Yizhou Xu and Pei Sun.
† tyanyang04@gmail.com & tiany20@mails.tsinghua.edu.cn; Department of Psychology & Tsinghua Laboratory of Brain and Intelligence, Tsinghua University, Beijing, 100084, China.
‡ xuyz23@mails.tsinghua.edu.cn; Department of Mathematical Sciences, Tsinghua University, Beijing, 100084, China.
§ peisun@tsinghua.edu.cn; Department of Psychology & Tsinghua Laboratory of Brain and Intelligence, Tsinghua University, Beijing, 100084, China.

```
    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

```
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:

```
## 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
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.

```
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)
    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
```

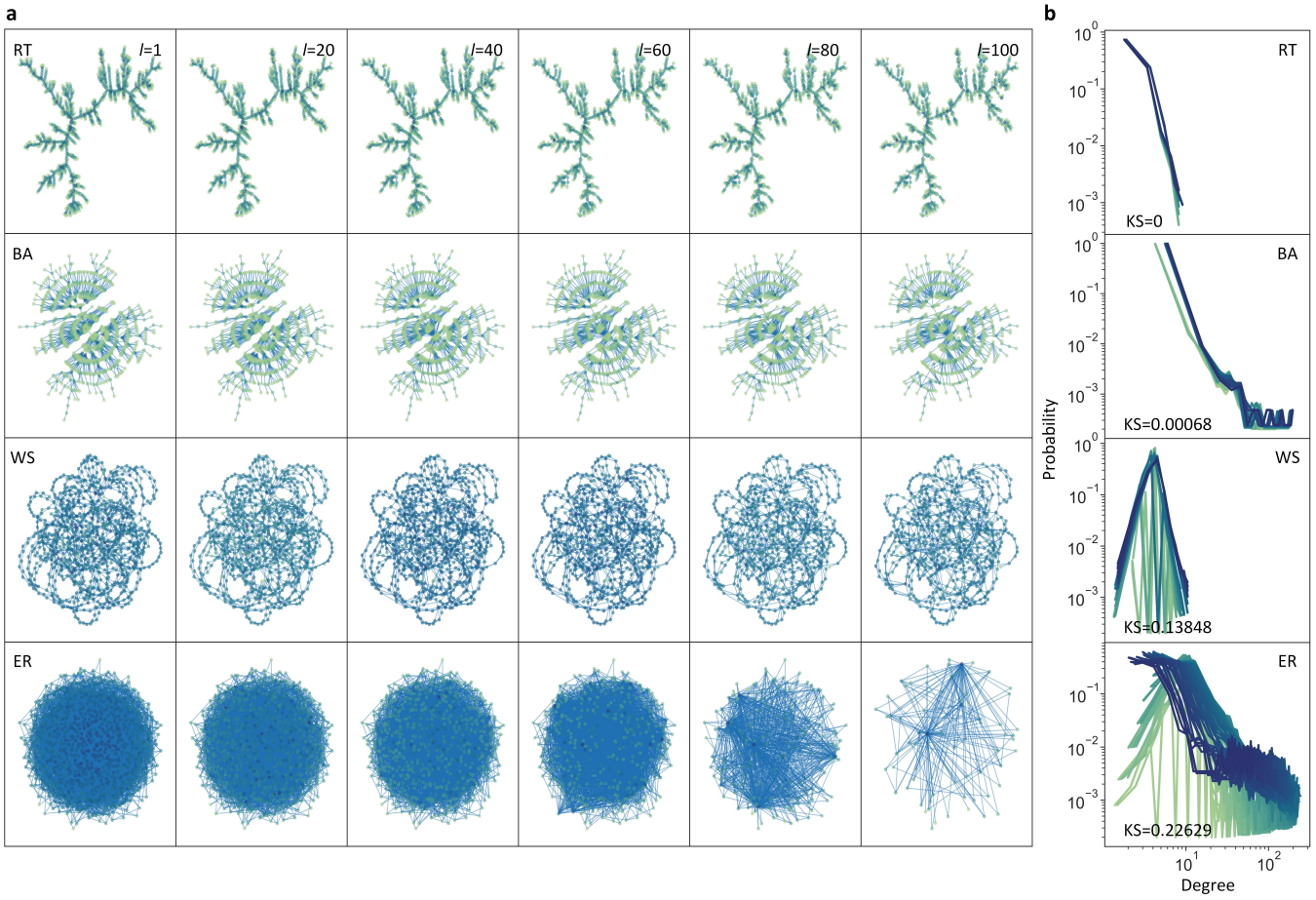

FIG. 1. Structure renormalization of random network models. **a**, Examples of the renormalization flows of the random tree (RT), the Barabási-Albert network (BA, the number of edges to attach from a new node to existing nodes is set as 1) [5], the Watts–Strogatz network (WS, each unit initially has 5 neighbors and edges are rewired according to a probability of 0.005) [6], and the Erdős-Rényi network (ER, the probability for two units to be connected is 0.01) [7]. **b**, The degree distributions of different random network models under the RRG transformation and their associated Kolmogorov–Smirnov statics [9, 10] (averaged across 10 configurations).

II. STRUCTURE RENORMALIZATION EXPERIMENT ON RANDOM NETWORK MODELS

To verify the validity of our proposed structure renormalization and macroscopic observable in evaluating scale-invariance, we apply the RRG on the random tree (RT), the Barabási-Albert network [5], the Watts–Strogatz network [6], and the Erdős-Rényi network [7]. Among these random network models, the random tree is scale-invariant while the Barabási-Albert network is weakly scale-invariant (i.e., invariant on a part of scales) [8]. The Watts–Strogatz network and the Erdős-Rényi network are not scale-invariant.

We generate 10 configurations for each random network model. Then, we implement a RRG with the signed Cauchy projection [4] and a binary hashing dimension of 50, on these configurations. Fig. 1a presents instances of the derived renormalization flows. As shown in Fig. 1b, the degree distributions averaged across configurations can be used to distinguish among scale-invariant, weakly scale-invariant, and scale-dependent random networks.

Note that the Kolmogorov–Smirnov statics [9, 10] in Fig. 1b can approach to zero because of configuration averaging. In real systems where configuration averaging is unavailable (e.g., there is only one system configuration), the finite size effects on degree distributions are still non-negligible.

[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. Albert and A.-L. Barabási, Statistical mechanics of complex networks, Reviews of modern physics **74**, 47 (2002).
[6] D. J. Watts and S. H. Strogatz, Collective dynamics of 'small-world'networks, nature **393**, 440 (1998).
[7] P. Erdős, A. Rényi, *et al.*, On the evolution of random graphs, Publ. Math. Inst. Hung. Acad. Sci **5**, 17 (1960).
[8] P. Villegas, T. Gili, G. Caldarelli, and A. Gabrielli, Laplacian renormalization group for heterogeneous networks, Nature Physics **19**, 445 (2023).
[9] R. Simard and P. L'Ecuyer, Computing the two-sided kolmogorov-smirnov distribution, Journal of Statistical Software **39**, 1 (2011).
[10] V. W. Berger and Y. Zhou, Kolmogorov–smirnov test: Overview, Wiley statsref: Statistics reference online (2014).

ACKNOWLEDGEMENTS

This project is supported by the Artificial and General Intelligence Research Program of Guo Qiang Research Institute at Tsinghua University (2020GQG1017) as well as the Tsinghua University Initiative Scientific Research Program. Authors appreciate Hedong Hou at the Institut de Mathématiques d'Orsay and Aohua Cheng at Tsinghua University for their inspiring discussions.