

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 present the code for analyzing macroscopic observables and scaling behaviours. In Sec. III, 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

```
1 ## Dependency libraries used for the RRG:
2 import networkx as nx
3 import faiss
4 import time
5 import scipy as spy
6 from datasketch import MinHash
7 import copy
8
9 ## Dependency libraries used for the scaling analysis:
10 from scipy.optimize import curve_fit
11 import statsmodels.api as sm
12 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

```
1 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.

* Correspondence should be addressed to Yizhou Xu and Pei Sun.

[†] tyanyang04@gmail.com & tian20@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.

```

1 def Renormalization_Flow(X_Initial, Iteration_Num, TargetDim, Method_Type, Data_Type):
2     RG_Flow=[]
3     RG_Flow.append(X_Initial)
4     Corase_ID_list=[]
5     for Iter in range(Iteration_Num):
6         StartT=time.time()
7         X_Current=RG_Flow[Iter]
8         if Data_Type=="Dynamics":
9             X_New, Corase_ID=Renormalization_Function(X_Current, TargetDim, Iter, Method_Type)
10        elif Data_Type=="Structure":
11            X_New, Corase_ID=Network_Renormalization_Function(X_Current, TargetDim, Iter, Method_Type)
12            if nx.number_of_edges(X_New)==0:
13                break
14            RG_Flow.append(X_New)
15            Corase_ID_list.append(Corase_ID)
16            EndT=time.time()
17            print(['The', Iter+1, 'time of renormalization costs-', EndT-StartT])
18        Tracked_ID_list=Tracking_System(Corase_ID_list)
19    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

```

1 Tracked_ID_list[0]=[0,1],[2],[3,5],[4]
2 Tracked_ID_list[1]=[0,1,2],[3,5],[4]
3 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:

```

1 ## Structure renormalization
2 X_Initial=nx.random_tree(10000) # Generate a random tree with 10000 units
3
4 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
5
6 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
7
8 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
9
10 ## Dynamics renormalization
11 X_Initial = np.random.randn(10000, 50000) # Generate a system with 10000 units, where each unit
   exhibits random dynamics for 50000 time steps
12
13 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
14
15 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
16
17 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](#).

```

1 def Random_Fourier_Feature_Hashing(X,TargetDim):
2     N = np.size(X,0)
3     d = np.size(X,1)
4     W = np.random.normal(loc=0, scale=1, size=(d, TargetDim))
5     b = np.random.uniform(0, 2*np.pi, size=TargetDim)
6     B = np.repeat(b[:, np.newaxis], N, axis=1).T
7     Z = 1/2* (1+ np.sign(np.cos(X @ W + B)))
8     Z = np.uint8(Z)
9     return Z
10
11 def Random_Cauchy_Feature_Hashing(X,TargetDim):
12     N = np.size(X,0)
13     d = np.size(X,1)
14     W = spy.stats.cauchy.rvs(loc=0, scale=1, size=(d, TargetDim))
15     b = np.random.uniform(0, 2*np.pi, size=TargetDim)
16     B = np.repeat(b[:, np.newaxis], N, axis=1).T
17     Z = 1/2* (1+ np.sign(np.cos(X @ W + B)))
18     Z = np.uint8(Z)
19     return Z
20
21 def Random_Hyperplane_Hashing(X,TargetDim):
22     d = np.size(X,1)
23     W = np.random.normal(loc=0, scale=1, size=(d, TargetDim))
24     Z = 1/2* (1+ np.sign(X @ W))
25     Z = np.uint8(Z)
26     return Z
27
28 def Random_Min_Hashing(X,TargetDim):
29     Z=np.zeros((len(X),TargetDim))
30     for ID1 in range(len(X)):
31         Hashing_Code=MinHash(num_perm=TargetDim)
32         Hashing_Code.update_batch(X[ID1])
33         Z[ID1,:]=Hashing_Code.hashvalues
34     return Z
35
36 def Neighbor_Generator(X,UnitNum):
37     Y=[]
38     for Unit in range(UnitNum):
39         Neighbors = [Unit] + list(X.neighbors(Unit))
40         Y.append(np.array(Neighbors))
41     return Y
42
43
44 def Normalization_Function(X_Current,Method_Type):
45     if Method_Type=="Linear_Kernel":
46         Normalized_X=X_Current-np.mean(X_Current,axis=1).reshape(np.size(X_Current,0),1)
47     elif Method_Type=="Gaussian_Kernel":
48         Normalized_X=X_Current-np.mean(X_Current,axis=1).reshape(np.size(X_Current,0),1)
49         Std=np.std(Normalized_X,axis=1).reshape(np.size(Normalized_X,0),1)
50         Normalized_X=np.divide(Normalized_X,Std,out=Normalized_X,where=Std!=0)
51     elif Method_Type=="Cauchy_Kernel":
52         Normalized_X=X_Current-np.min(X_Current,axis=1).reshape(np.size(X_Current,0),1)
53         SumV=np.sum(Normalized_X,axis=1).reshape(np.size(Normalized_X,0),1)
54         Normalized_X=np.divide(Normalized_X,SumV,out=Normalized_X,where=SumV!=0)
55     return Normalized_X
56
57 def Binary_Hashing_Index(Z):
58     if np.size(Z,0)<=50000:
59         Dim=8*np.size(Z,1)
60         Index = faiss.IndexBinaryFlat(Dim)
61         Index.nprobe = 2
62     elif (np.size(Z,0)>50000)&(np.size(Z,0)<=500000):
63         Dim=8*np.size(Z,1)
64         Index = faiss.IndexBinaryHash(Dim,Dim)

```

```

65     Index.nprobe = 2
66     elif np.size(Z,0)>500000:
67         Dim=8*np.size(Z,1)
68         Index = faiss.IndexBinaryHash(Dim,int(np.max([np.min([np.ceil(Dim/100),32]),16])))
69         Index.nprobe = 2
70     return Index
71
72
73 def KNN_with_Hashing_Index(Z):
74     StartT=time.time()
75     Index=Binary_Hashing_Index(Z)
76     Index.add(Z)
77     Num_neighbors=2
78     D, I = Index.search(Z, Num_neighbors)
79     EndT=time.time()
80     print(['KNN search costs-', EndT-StartT])
81     return D,I
82
83 def Hashing_Function(Normalized_X,TargetDim,Method_Type):
84     if Method_Type=="Linear_Kernel":
85         Z=Random_Hyperplane_Hashing(Normalized_X,TargetDim)
86     elif Method_Type=="Gaussian_Kernel":
87         Z=Random_Fourier_Feature_Hashing(Normalized_X,TargetDim)
88     elif Method_Type=="Cauchy_Kernel":
89         Z=Random_Cauchy_Feature_Hashing(Normalized_X,TargetDim)
90     return Z
91
92 def Renormalization_Function(X_Current,TargetDim,Iter,Method_Type):
93     Normalized_X=Normalization_Function(X_Current,Method_Type)
94     Z=Hashing_Function(Normalized_X,TargetDim,Method_Type)
95     _,I=KNN_with_Hashing_Index(Z)
96     G = nx.empty_graph(np.size(I,0))
97     Edge = np.vstack((np.arange(0, np.size(I, 0)), I[:,1])).T
98     G.add_edges_from(Edge)
99     Clusters=[list(c) for c in list(nx.connected_components(G))]
100    ClusterNum=nx.number_connected_components(G)
101    print(['There are', ClusterNum, 'macro-units after', Iter+1, 'times of renormalization'])
102    X_New=np.zeros((ClusterNum, np.size(X_Current,1)))
103    Corase_ID = []
104    for ID1 in range(ClusterNum):
105        X_New[ID1,:]=np.sum(X_Current[Clusters[ID1],:],axis=0)
106        Corase_ID.append(Clusters[ID1])
107    return X_New, Corase_ID
108
109 def Network_Renormalization_Function(X_Current,TargetDim,Iter,Method_Type):
110    UnitNum=nx.number_of_nodes(X_Current)
111    Y=Neighbor_Generator(X_Current,UnitNum)
112    Z=Random_Min_Hashing(Y,TargetDim)
113    Z=Hashing_Function(Z,TargetDim,Method_Type)
114    _,I=KNN_with_Hashing_Index(Z)
115    G = nx.empty_graph(np.size(I,0))
116    Edge = np.vstack((np.arange(0, np.size(I, 0)), I[:,1])).T
117
118    G.add_edges_from(Edge)
119    Potential_Clusters=[list(c) for c in list(nx.connected_components(G))]
120    Potential_ClusterNum=nx.number_connected_components(G)
121    Edge_To_Remove=[]
122    for ID1 in range(Potential_ClusterNum):
123        Unit_list=Potential_Clusters[ID1]
124        if len(Unit_list)>1:
125            H = nx.induced_subgraph(X_Current,Unit_list)
126            Potential_H = nx.induced_subgraph(G,Unit_list)
127            Wrong_Edge=list(set(list(Potential_H.edges))-set(list(H.edges)))
128            Edge_To_Remove.extend(Wrong_Edge)
129
130    for Wrong_Edge in Edge_To_Remove:
131        G.remove_edge(*Wrong_Edge)
132
133    Clusters=[list(c) for c in list(nx.connected_components(G))]
134    ClusterNum=nx.number_connected_components(G)

```

```

135 print(['There are', ClusterNum, 'macro-units after', Iter+1, 'times of renormalization'])
136
137
138 X_New=copy.deepcopy(X_Current)
139 Pre_Corase_ID = []
140 Mappings={}
141 for ID1 in range(ClusterNum):
142     Unit_list=Clusters[ID1]
143     Pre_Corase_ID.append(Unit_list)
144     Unit0 = Unit_list[0]
145     Mappings[Unit0]=ID1
146     for Unit in Unit_list[1:]:
147         if X_New.has_node(Unit):
148             Neighbors = list(X_New.neighbors(Unit))
149             New_edges = [(Unit0, Nei) for Nei in Neighbors if Unit0!=Nei]
150             X_New.add_edges_from(New_edges)
151             X_New.remove_node(Unit)
152 Corase_ID = []
153 Unit_Mappings={}
154 for ID_1,ID_2 in enumerate(X_New.nodes()):
155     Unit_Mappings[ID_2]=ID_1
156     Corase_ID.append(Pre_Corase_ID[Mappings[ID_2]])
157 X_New = nx.relabel_nodes(X_New, Unit_Mappings)
158
159 return X_New, Corase_ID
160
161 def Tracking_System(Corase_ID_list):
162     Tracked_ID_list = []
163     for IterID in range(len(Corase_ID_list)):
164         if IterID==0:
165             Tracked_ID_list.append(Corase_ID_list[0])
166         else:
167             Tracked_ID = []
168             if len(Corase_ID_list[IterID])>0:
169                 for CoarseID in range(len(Corase_ID_list[IterID])):
170                     UnitsToTrack=Corase_ID_list[IterID][CoarseID]
171                     Searched_ID=[]
172                     for IDSearch in range(len(UnitsToTrack)):
173                         Search_ID=1
174                         while len(Tracked_ID_list[IterID-Search_ID])==0:
175                             Search_ID=Search_ID+1
176                             Searched_ID=Searched_ID+Tracked_ID_list[IterID-Search_ID][UnitsToTrack[
177 IDSearch]]
178                             Tracked_ID.append(Searched_ID)
179                             Tracked_ID_list.append(Tracked_ID)
180 return Tracked_ID_list
181
182 def Renormalization_Flow(X_Initial, Iteration_Num, TargetDim, Method_Type, Data_Type):
183     RG_Flow=[]
184     RG_Flow.append(X_Initial)
185     Corase_ID_list=[]
186     for Iter in range(Iteration_Num):
187         StartT=time.time()
188         X_Current=RG_Flow[Iter]
189         if Data_Type=="Dynamics":
190             X_New, Corase_ID=Renormalization_Function(X_Current, TargetDim, Iter, Method_Type)
191         elif Data_Type=="Structure":
192             X_New, Corase_ID=Network_Renormalization_Function(X_Current, TargetDim, Iter, Method_Type)
193             if nx.number_of_edges(X_New)==0:
194                 break
195         RG_Flow.append(X_New)
196         Corase_ID_list.append(Corase_ID)
197         EndT=time.time()
198         print(['The', Iter+1, 'time of renormalization costs-', EndT-StartT])
199     Tracked_ID_list=Tracking_System(Corase_ID_list)
200     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]

```
1 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

```
1 def KS_Analysis(RG_Flow):
2     K_S_Static=np.zeros(len(RG_Flow))
3     Degrees_0=[Node[1] for Node in list(nx.degree(RG_Flow[0]))]
4     for InterID in range(len(RG_Flow)):
5         Degrees=[Node[1] for Node in list(nx.degree(RG_Flow[InterID]))]
6         KstestResult=ks_2samp(Degrees, Degrees_0, alternative='two-sided',method='exact')
7         K_S_Static[InterID]=KstestResult[0]*(KstestResult[1]<0.01)
8
9     Mean_K_S_Static=np.mean(K_S_Static)
10    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

```
1 Cut_Off_Ratio=0.1
2 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

```
1 def Normalized_Dynamics(RG_Flow,Tracked_ID_list,Cut_Off_Ratio):
2     ClusterNum=np.array([len(Tracked_ID_list[ID1]) for ID1 in range(1,len(Tracked_ID_list))])
3     Max_Range=np.max(np.where(ClusterNum>1)[0])+1
4     for IterID in range(Max_Range):
5         X_Current=RG_Flow[IterID]
6         N=np.size(X_Current,0)
7         Covariance = np.cov(X_Current)
8         Evals, U = np.linalg.eig(Covariance)
9         Idx = Evals.argsort()[::-1]
10        EigenValues = Evals[Idx]
11        EigenVectors = U[:,Idx]
12        k=int(np.round(N*Cut_Off_Ratio))
13        P=EigenVectors[:,k] @ EigenVectors[:,k].T
14        phi=P@(X_Current-np.mean(X_Current,axis=1,keepdims=True))
15        Normalized_activity=phi/np.std(phi,axis=1,keepdims=True)
16    return Normalized_activity
```

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

```
1 MeanClusterSize, MeanVar, Coeff, Alpha, R2, MSE, Esti_Alpha_Scaling = Alpha_Scaling(RG_Flow,
2     Tracked_ID_list)
3 MeanClusterSize, FreeEV, Coeff, Beta, R2, MSE, Esti_Beta_Scaling = Beta_Scaling(RG_Flow,
4     Tracked_ID_list)
```

```

5 Average_Rank_K, Average_Evals, Coeff, Mu, R2, MSE, Esti_Mu_Scaling = Mu_Scaling(RG_Flow,
   Tracked_ID_list)
6
7 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 $\text{Var}(\langle K^{(l)} \rangle)$. [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 $\text{Var}(\langle K^{(l)} \rangle)$ is contained by [Esti_Alpha_Scaling](#).

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

```

1  ## Analysis
2  def Linear_func(x, a, b):
3      return b*x+a
4
5  def Power_func(x, a):
6      return a*x
7
8
9  def RSquareFun(X,y,popt):
10     if len(popt)==2:
11         pre_y = Linear_func(X, popt[0], popt[1])
12     elif len(popt)==1:
13         pre_y = Power_func(X, popt[0])
14     mean = np.mean(y)
15     ss_tot = np.sum((y - mean) ** 2)
16     ss_res = np.sum((y - pre_y) ** 2)
17     r_squared = 1 - (ss_res / ss_tot)
18
19     mse = np.sum((y - pre_y) ** 2) / len(y)
20     return r_squared, mse
21
22 def Alpha_Scaling(RG_Flow, Tracked_ID_list):
23     MeanVar=np.zeros(len(RG_Flow))
24     for Iter in range(len(RG_Flow)):
25         X=RG_Flow[Iter]
26         MeanVar[Iter]=np.mean(np.var(X,axis=1))
27
28     MeanClusterSize=np.ones(len(RG_Flow))
29     for Iter in range(len(Tracked_ID_list)):
30         ClusterSize=[len(IDC) for IDC in Tracked_ID_list[Iter]]
31         MeanClusterSize[Iter+1]=np.mean(ClusterSize)
32
33     popt, _ = curve_fit(Linear_func, np.log(MeanClusterSize), np.log(MeanVar))
34     Coeff = popt[0]
35     Alpha = popt[1]
36     R2, MSE= RSquareFun(np.log(MeanClusterSize), np.log(MeanVar), popt)
37     Esti_Alpha_Scaling=np.exp(Coeff)*np.power(MeanClusterSize,Alpha)
38     return MeanClusterSize, MeanVar, Coeff, Alpha, R2, MSE, Esti_Alpha_Scaling
39
40 def Beta_Scaling(RG_Flow, Tracked_ID_list):
41     FreeEV=np.zeros(len(RG_Flow))
42     for Iter in range(len(RG_Flow)):
43         X=RG_Flow[Iter]
44
45         P_SilenceV=np.zeros(np.size(X,0))
46         for ID1 in range(np.size(X,0)):
47             P_SilenceV[ID1] = 1-np.count_nonzero(X[ID1,:]) / np.size(X,1)

```

```

48     P_Silence=np.mean(P_SilenceV)
49     FreeEV[Iter]=-1*np.log(P_Silence)
50
51     MeanClusterSize=np.ones(len(RG_Flow))
52     for Iter in range(len(Tracked_ID_list)):
53         ClusterSize=[len(IDC) for IDC in Tracked_ID_list[Iter]]
54         MeanClusterSize[Iter+1]=np.mean(ClusterSize)
55
56     Needed=np.where(np.isinf(FreeEV)==0)[0]
57     FreeEV=FreeEV[Needed]
58     MeanClusterSize=MeanClusterSize[Needed]
59
60     popt, _ = curve_fit(Linear_func, np.log(MeanClusterSize), np.log(FreeEV))
61     Coeff = popt[0]
62     Beta = popt[1]
63     R2, MSE= RSquareFun(np.log(MeanClusterSize), np.log(FreeEV), popt)
64     Esti_Beta_Scaling=np.exp(Coeff)*np.power(MeanClusterSize,Beta)
65     return MeanClusterSize, FreeEV, Coeff, Beta, R2, MSE, Esti_Beta_Scaling
66
67 def Mu_Scaling(RG_Flow,Tracked_ID_list):
68     Initial_X = RG_Flow[0]
69     Average_Rank_K=[]
70     Average_Evals=[]
71
72     ClusterNum=np.array([len(Tracked_ID_list[ID1]) for ID1 in range(1,len(Tracked_ID_list))])
73     Max_Range=np.max(np.where(ClusterNum>1)[0])+2
74     for ID1 in range(1,Max_Range):
75         x=[]
76         y=[]
77         for ID2 in range(len(Tracked_ID_list[ID1])):
78             WithinCluster= Tracked_ID_list[ID1][ID2]
79             X_WC = Initial_X[WithinCluster,:]
80             X_WC=X_WC-np.mean(X_WC,axis=1).reshape(np.size(X_WC,0),1)
81             Cov=np.cov(X_WC)
82             Evals, _ = np.linalg.eig(Cov)
83             Evals = np.sort(np.real(Evals))
84             Evals = Evals[::-1]
85
86             Rank = np.cumsum(np.ones(len(Evals)))
87             Rank_K=Rank/len(WithinCluster)
88
89             Needed_Loc=np.where(Evals>0)[0]
90             Rank_K=Rank_K[Needed_Loc]
91             Evals=Evals[Needed_Loc]
92             x.extend(Rank_K[:])
93             y.extend(Evals[:])
94
95         _, bins = np.histogram(x)
96         Meanx=np.zeros(len(bins)-1)
97         Meany=np.zeros(len(bins)-1)
98         for ID3 in range(len(bins)-1):
99             Needed_x=np.where((x>=bins[ID3])&(x<=bins[ID3+1]))[0]
100             Meanx[ID3]=np.mean(np.array(x)[Needed_x])
101             Meany[ID3]=np.mean(np.array(y)[Needed_x])
102         Average_Rank_K.extend(Meanx)
103         Average_Evals.extend(Meany)
104
105     popt, _ = curve_fit(Linear_func, np.log(Average_Rank_K), np.log(Average_Evals))
106     Coeff = popt[0]
107     Mu = -1* popt[1]
108     R2, MSE= RSquareFun(np.log(Average_Rank_K), np.log(Average_Evals), popt)
109     Esti_Mu_Scaling=np.exp(Coeff)*np.power(Average_Rank_K,-1* Mu)
110     return Average_Rank_K, Average_Evals, Coeff, Mu, R2, MSE, Esti_Mu_Scaling
111
112 def Theta_Scaling(RG_Flow,Tracked_ID_list):
113     Tau=np.zeros(len(RG_Flow))
114     ScaledT=[]
115     MeanACFs=[]
116
117     for Iter in range(len(RG_Flow)):

```

```

118 X=RG_Flow[Iter]
119 SumAC = np.sum(X, axis=1)
120
121 ACFMatrix = np.zeros_like(X)
122 for ID1 in range(np.size(X,0)):
123     ACFMatrix[ID1,:] = sm.tsa.acf(X[ID1,:], nlags=np.size(X,1))
124 ACFMatrix = ACFMatrix[np.where(SumAC>0)[0],:]
125 MeanACF = np.mean(ACFMatrix, axis=0)
126 T = np.cumsum(np.ones(np.size(X,1)))-1
127
128 Needed_ACF=np.where(MeanACF>0)[0]
129 MeanACF=MeanACF[Needed_ACF]
130 T=T[Needed_ACF]
131
132 Cut_Off=int(np.max([np.ceil(0.01*len(T)),100]))
133 popt, _ = curve_fit(Power_func, T[:Cut_Off], np.log(MeanACF[:Cut_Off]))
134 Tau[Iter] = -1/popt[0]
135
136 ScaledT.append(T/Tau[Iter])
137 MeanACFs.append(MeanACF)
138
139 MeanClusterSize=np.ones(len(RG_Flow))
140 for Iter in range(len(Tracked_ID_list)):
141     ClusterSize=[len(IDC) for IDC in Tracked_ID_list[Iter]]
142     MeanClusterSize[Iter+1]=np.mean(ClusterSize)
143
144 popt, _ = curve_fit(Linear_func, np.log(MeanClusterSize), np.log(Tau))
145 Coeff = popt[0]
146 Theta = popt[1]
147 R2, MSE= RSquareFun(np.log(MeanClusterSize), np.log(Tau), popt)
148 Esti_Theta_Scaling=np.exp(Coeff)*np.power(MeanClusterSize,Theta)
149
150 return ScaledT, MeanACFs, MeanClusterSize, Tau, Coeff, Theta, R2, MSE, Esti_Theta_Scaling
151
152 def KS_Analysis(RG_Flow):
153     K_S_Static=np.zeros(len(RG_Flow))
154     Degrees_0=[Node[1] for Node in list(nx.degree(RG_Flow[0]))]
155     for InterID in range(len(RG_Flow)):
156         Degrees=[Node[1] for Node in list(nx.degree(RG_Flow[InterID]))]
157         KstestResult=ks_2samp(Degrees, Degrees_0, alternative='two-sided',method='exact')
158         K_S_Static[InterID]=KstestResult[0]*(KstestResult[1]<0.01)
159
160     Mean_K_S_Static=np.mean(K_S_Static)
161     return Mean_K_S_Static
162
163 def Normalized_Dynamics(RG_Flow,Tracked_ID_list,Cut_Off_Ratio):
164     ClusterNum=np.array([len(Tracked_ID_list[ID1]) for ID1 in range(1,len(Tracked_ID_list))])
165     Max_Range=np.max(np.where(ClusterNum>1)[0])+1
166     for IterID in range(Max_Range):
167         X_Current=RG_Flow[IterID]
168         N=np.size(X_Current,0)
169         Covariance = np.cov(X_Current)
170         Evals, U = np.linalg.eig(Covariance)
171         Idx = Evals.argsort()[::-1]
172         EigenValues = Evals[Idx]
173         EigenVectors = U[:,Idx]
174         k=int(np.round(N*Cut_Off_Ratio))
175         P=EigenVectors[:,k] @ EigenVectors[:,k].T
176         phi=P@(X_Current-np.mean(X_Current,axis=1,keepdims=True))
177         Normalized_activity=phi/np.std(phi,axis=1,keepdims=True)
178     return Normalized_activity

```

III. 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 [7], the Watts–Strogatz network [8], and the Erdős–Rényi network [9]. Among these random network models, the random tree is scale-invariant while

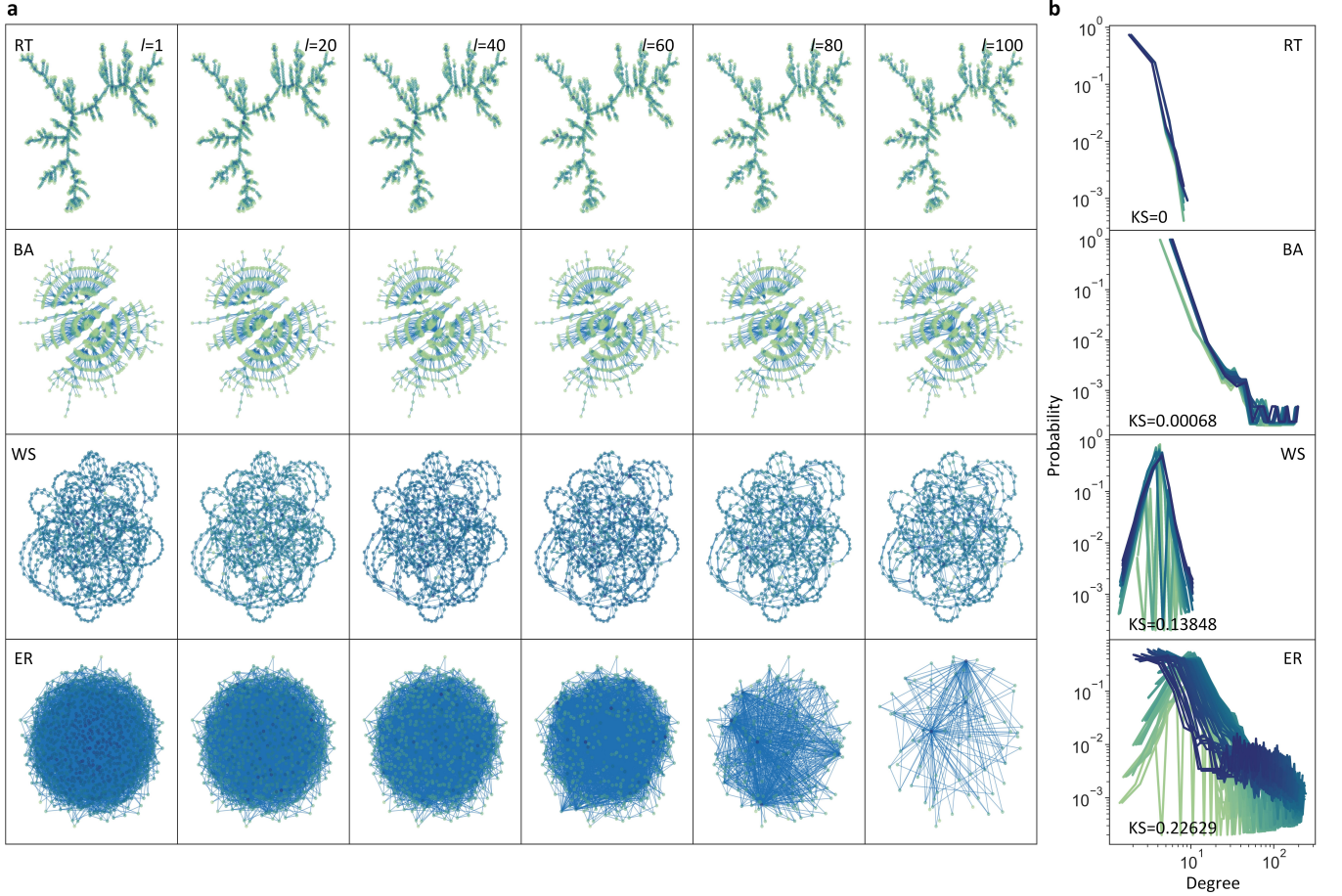


FIG. 1. Structure renormalization of random network models. **a**, Examples of the renormalization flows of the random tree (RT), the Barabási-Albert network (BA, the number of edges to attach from a new node to existing nodes is set as 1) [7], the Watts-Strogatz network (WS, each unit initially has 5 neighbors and edges are rewired according to a probability of 0.005) [8], and the Erdős-Rényi network (ER, the probability for two units to be connected is 0.01) [9]. **b**, The degree distributions of different random network models under the RRG transformation and their associated Kolmogorov-Smirnov statistics [5, 6] (averaged across 10 configurations).

the Barabási-Albert network is weakly scale-invariant (i.e., invariant on a part of scales) [10]. 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 statistics [5, 6] 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. 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).
- [7] R. Albert and A.-L. Barabási, Statistical mechanics of complex networks, *Reviews of modern physics* **74**, 47 (2002).
- [8] D. J. Watts and S. H. Strogatz, Collective dynamics of ‘small-world’ networks, *nature* **393**, 440 (1998).
- [9] P. Erdős, A. Rényi, *et al.*, On the evolution of random graphs, *Publ. Math. Inst. Hung. Acad. Sci* **5**, 17 (1960).
- [10] P. Villegas, T. Gili, G. Caldarelli, and A. Gabrielli, Laplacian renormalization group for heterogeneous networks, *Nature Physics* **19**, 445 (2023).

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.