Random renormalization group for fast renormalizing ultra-large complex systems (supplementary material)*

Yang Tian†
Infplane Computing Technologies Ltd, Beijing, 100080, China

Yizhou Xu‡
Infplane Computing 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 "Random Renormalization Group for Fast Renormalizing Ultra-Large Complex Systems." In Section I, we introduce research background and review several existing works that are closely related to our framework. Section II details our motivations underlying the random renormalization group (RRG) framework. Section III presents additional experimental results, including tests on random weighted graphs and real-world datasets. Section IV provides the RRG code implementation with usage examples, while Section V includes code for analyzing macroscopic observables and scaling behaviors.

I. ELEMENTARIES OF THE RENORMALIZATION GROUP

A. Basic properties of the renormalization mapping

Given a system X, the data Y is recorded to reflect its properties such that each system unit X_i is characterized by Y_i (e.g., each Y_i can be a time series of neural activities if X_i is a neuron). We assume that the input of the renormalization flow, $Y^{(1)} = Y$, and its associated probability distribution, $P^{(1)} = P\left(Y^{(1)}\right)$, are defined on certain microscopic length scale, $\ell^{(1)} \in \mathbb{R}^+$. The renormalization mapping $R(\cdot)$ defines a recursive equation

$$P^{(l+1)} = R\left(P^{(l)}\right). \tag{1}$$

This RG equation averages out all the details of $P^{(l)}$ that are defined on the length scales below $\ell^{(l)}$ to obtain $P^{(l+1)}$, a probability distribution of the coarse grained system. By gradually enlarging the length scale over iterations (i.e., we set $\ell^{(l)} \geq \ell^{(l-1)}$ for each l), the RG equation generates a flow that starts from the concerned system on a relatively microscopic scale and ends at a renormalized system on a relatively macroscopic scale. This probabilistic perspective serves as a general view about the renormalization group theory [1, 2].

In general, the renormalization group theory studies how to define an effective mapping $R(\cdot)$ and analyzes how the system X evolves during renormalization [1, 3]. For the precise definition of $R(\cdot)$, it can be proposed in real space or momentum space. Here we summarize the key functions of the mapping $R(\cdot)$ during each iteration, regardless of what kind of spaces it is defined in:

(i) Find system units to coarse grain on the current scale. In principle, a set of system units to be coarse grained should share short-range interactions among each other (e.g., form small-scale cells in real space or correspond to short wavelength modes in momentum space).

(ii) Define macro-units on the next scale that the current system units should be aggregated into. Given each set of system units to coarse grain on the current scale, the mapping $R(\cdot)$ defines a macro-unit to inherit their properties in some ways and represent them on the next scale. In the theory with a continuum of freedom

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

degrees (e.g., statistical and quantum field theories), a renormalization procedure is necessary for dealing with the infinities that occur during coarse graining. If the effects of infinities are completely predictable (e.g., when renormalization only creates a linear transformation on scaling exponents and has no effect on scaling relations) or the emergence of infinities is impossible (e.g., for discrete and finite complex networks), one can choose whether to apply the renormalization step after coarse graining (i.e., the results obtained with or without renormalization can be easily inter-converted) [4].

(iii) Re-confirm the relations among macro-units on the next scale according to the coarse graining process and system unit relations on the current scale.

By properly realizing the above three functions, we can define a meaningful mapping $R(\cdot)$. By iteratively applying $R(\cdot)$ on the system X, we obtain a renormalization flow. In real application, we need to explicitly define $R(\cdot)$. Below, we respectively elaborate the renormalization mappings for system dynamics (i.e., the activities of system units over time) and structures (i.e., the networks formed by system units) in Sec. I B and Sec. I C.

B. Renormalization mapping for dynamics

In the task of dynamics renormalization, we aim at verifying the potential existence of scaling behaviours in unit activities. Unlike the conventional research targets defined on regular lattices or homogeneous Euclidean spaces, most complex systems in the real world lack an explicit definition of locality (i.e., there is no *a priori* definition of neighborhood relations among system units). The absence of locality proposes a challenge for developing RG for complex systems. Below, we review the representative methods proposed for overcoming this challenge in real space and momentum space, respectively.

Real space approach for dynamics For real space, we are particularly interested in the form of $R(\cdot)$ used in the phenomenological RG [5, 6]. The phenomenological RG implements a statistical method to detect the latent neighborhood relations among system units when there is no explicit locality that determines system dynamics. Their method is based on a simple and reasonable assumption on the underlying locality, i.e., the system units that share latent local interactions should be more correlated than those that do not interact with each other [2]. In general, this assumption is valid in most real systems unless the system is fully governed by an external source that is sufficiently strong to determine system dynamics (i.e., when this external source can globally and completely control every unit, unit correlations are determined by the behaviours of the external source and can no longer reflect local unit interactions within the system).

Specifically, the authors of the phenomenological RG [5, 6] suggest to consider a general form

$$Y_i^{(l)} \to Y_i^{(l+1)} = f\left(\left\{Y_j^{(l)} \middle| j \in V_i^{(l)}\right\}\right), \tag{2}$$

where each $V_i^{(l)}$ serves as a correlation-based counterpart of the Kadanoff block that contains unit $X_i^{(l)}$ (i.e., the set of all the nearest neighbors of unit $X_i^{(l)}$ in the correlation space) [2, 5, 6]. The mapping $f(\cdot)$ is a function that describes how the coarse grained macro-unit inherits the properties of the units aggregated into it [2]. This function can be defined in diverse manner. For instance, we can define $f(\cdot)$ as a linear mapping to average over the properties of units in $V_i^{(l)}$ (i.e., we define $X_i^{(l+1)} = r\sum_{j\in V_i^{(l)}} X_j^{(l)}$ for an appropriate $r \in \mathbb{R}$). This procedure transforms discrete Ising–like variables to a more continuous local magnetization across iterations [2]. As another instance, we can let $f(\cdot)$ be a frequency-counting-based function that implements the majority rule on $\{X_j^{(l)}|j \in V_i^{(l)}\}$ (i.e., we define $X_i^{(l+1)} = \sigma$ where σ is the mode number in $\{X_j^{(l)}|j \in V_i^{(l)}\}$). In this case, we can use $f(\cdot)$ to transform Ising–like variables on the original structure into Ising–like variables on a sparser structure (this procedure actually shares the same spirit with the original block-spin construction. The only difference is that the considered structure in the original construction is lattice while that in Eq. (2) is correlation-based network [2]).

In the original works of the phenomenological RG [2, 6], the authors realize Eq. (2) as a special formula

$$Y_i^{(l)} \to Y_i^{(l+1)} = Y_i^{(l)} + Y_{k(i)}^{(l)}, \tag{3}$$

where $k(i)$ is an element in $V_i^{(l)}$ that makes $Y_{k(i)}^{(l)}$ and $Y_i^{(l)}$ maximally correlated. In fact, the mapping $f(\cdot)$ considered in Eq. (3) has an equivalent expression

$$f\left(\left\{Y_j^{(l)} \middle| j \in V_i^{(l)}\right\}\right) = \sum_{j\in V_i^{(l)}} w_{ij} Y_j^{(l)}, \tag{4}$$

where Eq. (3) defines $w_{ij} = \delta(j, k(i))$ using the Kronecker delta function $\delta(\cdot,\cdot)$.

Momentum approach for dynamics In momentum space, the authors of the phenomenological RG also suggest a way to implement renormalization [2, 5, 6]. Their method starts by assuming the translational invariance of the system X. Given this assumption, the Fourier transform of the correlation function in X coincides with the eigenvalue spectrum of the covariance matrix calculated among system units [2, 5, 6]. Consequently, coarse graining in momentum space (i.e., integrating out the Fourier modes with small wavelength) is expected to be same as integrating out low-variability contributions (i.e., those with small eigenvalues) in the covariance matrix (one can also find similar ideas in the PCA-like RG [7]). This momentum approach shares the same sprit with the real space one described in Eq. (2) since the correlation matrix is nothing more but a normalized covariance matrix. The covariance matrix serves as a non-parametric description of unit interactions within the system.

Mathematically, this momentum approach can be realized by several expressions [2, 5, 6]

$$Y_i^{(l)} \to Y_i^{(l+1)} = \mathrm{std}\left(Q_i^{(l)}\right)^{-1} Q_i^{(l)}, \tag{5}$$

where $\mathrm{std}\,(\cdot)$ denotes the standard deviation and $Q_i^{(l)}$ is the i-th row of matrix $Q^{(l)}$

$$Q^{(l)} = \left(\sum_{o=1}^{O^{(l)}} L_o^\top L_o\right)\left(Y^{(l)} - M^{(l)}\right). \tag{6}$$

In Eq. (6), we denote L_o as the eigenvector corresponding to the o-th largest eigenvalue of the covariance matrix formed among the feature vectors of all system units (i.e., the (i,j)-element of this matrix denotes the covariance between $Y_i^{(l)}$ and $Y_j^{(l)}$). The matrix $M^{(l)}$ is defined as

$$M^{(l)} = \left[\mu_1^{(l)}, \ldots, \mu_{N^{(l)}}^{(l)}\right]^\top \mathbf{1}, \tag{7}$$

where each $\mu_i^{(l)}$ denotes the mean value of $Y_i^{(l)}$ and $\left[\mu_1^{(l)}, \ldots, \mu_{N^{(l)}}^{(l)}\right]^\top$ defines a column vector. Notion $\mathbf{1}$ is an all-one row vector of an appropriate size.

The projector $\sum_{o=1}^{O^{(l)}} L_o^\top L_o$ is the key to coarse grain the system since it defines a cut-off in the momentum space when $O^{(l)} < N^{(l)}$ (i.e., there are $N^{(l)}$ eigenvalues in total and we only keep the contributions from $O^{(l)}$ larger eigenvalues). In Eq. (6), we standardize the system on the current scale by subtracting the mean from the data. Then we coarse grain the system and project it to the next scale. Finally, we rescale the system on the new scale in Eq. (5) to ensure that each $Y_i^{(l+1)}$ has a unit variance (i.e., a kind of renormalization).

C. Renormalization mapping for structures

In the task of structure renormalization, we are interested in the possible existence of scale-invariance property. This is a booming research field and we refer to Ref. [4] for a comprehensive review of the latest advances. Similar to those in field of dynamics renormalization, recent structure renormalization frameworks mainly focus on complex systems with high heterogeneity and without *a priori* organization rules. Below, we summarize the previous frameworks related to our work.

Real space approach for structure Among existing frameworks, diverse shortest-path-based methods may be the most well-known ones due to their simplicity and high practicability (e.g., see Refs. [8–10]). In brief, these methods group system units within a given shortest path distance into a macro-unit to generate the coarse grained system [4]. Although these methods are highly applicable to analyzing fractal and self-similar properties, we do not follow their ideas because of two reasons. First, the potential existence of small-world property may limit the possibility for these methods to reveal accurate scaling behaviours [4]. Second, shortest path distance calculation is computationally expensive, especially when we analyze ultra-large systems. These two limits force us to explore other possibilities.

The work that inspires us is the Laplacian RG [11]. In general, the authors of the Laplacian RG considers a diffusion process to describe interactions among system units. Similar to the classic heat diffusion equation where system evolution is governed by the heat kernel, we can write a diffusion evolution operator to describe the unit interactions placed over complicated structures [11]

$$K^{(l)}(\tau) = \exp\left(-\tau \mathcal{L}^{(l)}\right). \tag{8}$$

In Eq. (8), we use τ to control the diffusion time scale and let $\mathcal{L}^{(l)}$ be the graph Laplacian of the system structure on the current scale. The element $K_{ij}^{(l)}(\tau)$ describes the fraction of information diffused from unit $X_i^{(l)}$ to unit $X_j^{(l)}$ along all the possible paths connecting between these two units in the structure during a duration of τ [12, 13]. The coarse graining is implemented based on a binary reference matrix

$$B_{ij}^{(l)}(\tau) = \Theta\left(K_{ij}^{(l)}(\tau) - \min\{K_{ii}^{(l)}(\tau), K_{jj}^{(l)}(\tau)\}\right), \tag{9}$$

where $\Theta(\cdot)$ denotes the unit step function (i.e., $\Theta(x) = 1$ if $x \geq 0$ and $\Theta(x) = 0$ otherwise). Given this binary reference matrix, we can construct a reference network by using $B^{(l)}(\tau)$ as the adjacency matrix. Then, the units in each cluster (i.e., connected component) of the reference network are treated as strongly correlated. In $X^{(l)}$, we coarse grain these strongly correlated units into a macro-unit in $X^{(l+1)}$. If the units aggregated into macro-unit $X_i^{(l+1)}$ share at least one edge with the units aggregated into macro-unit $X_j^{(l+1)}$, then we define an edge between macro-units $X_i^{(l+1)}$ and $X_j^{(l+1)}$.

Momentum space approach for structure Parallel to real space renormalization, the Laplacian RG [11] also consists of a momentum space approach. By its nature, the key idea of this momentum space approach is to use Laplacian eigenvectors as the counterpart of Fourier basis and treat Laplacian eigenvalues as wave modes. Let us consider the eigendecomposition of the Laplacian using the bra-ket notation

$$\mathcal{L}^{(l)} = \sum_{\kappa^{(l)}} \kappa^{(l)} \left|\kappa^{(l)}\right\rangle \left\langle\kappa^{(l)}\right|, \tag{10}$$

where the positive semi-definite property of the Laplacian ensures that each eigenvalue satisfies $\kappa^{(l)} \geq 0$.

The momentum shell renormalization is implemented as the following: First, we determine a time scale τ^* to control the resolution scale (see Ref. [11] for the selection method of τ^*). Then, we use $1/\tau^*$ as a threshold to subdivide the Laplacian spectrum into two sets, which respectively correspond to $O^{(l)}$ small eigenvalues (i.e., no more than $1/\tau^*$) and $N^{(l)} - O^{(l)}$ large eigenvalues (i.e., larger than $1/\tau^*$). According to

$$K^{(l)}(\tau^*) = \sum_{\kappa^{(l)}} \exp\left(-\tau^* \kappa^{(l)}\right) \left|\kappa^{(l)}\right\rangle \left\langle\kappa^{(l)}\right|, \tag{11}$$

we can see that smaller eigenvalues lead to the slower decay of diffusion. Therefore, smaller eigenvalues contribute more to long-term interactions than larger ones. In the renormalization step, we integrate out the fast decaying components of the system by only keeping the slower eigenvectors

$$\mathcal{L}_{\text{trun}}^{(l)} = \sum_{\kappa^{(l)}} \Theta\left(1/\tau^* - \kappa^{(l)}\right) \kappa^{(l)} \left|\kappa^{(l)}\right\rangle \left\langle\kappa^{(l)}\right|. \tag{12}$$

Finally, we re-scale the truncated Laplacian to ensure that τ^* becomes the new unit time scale, $\mathcal{L}^{(l+1)} = \tau^* \mathcal{L}_{\text{trun}}^{(l)}$. One can see that the above renormalization procedure is essentially consistent with the Wilson's RG.

II. ELEMENTARIES OF THE RANDOM RENORMALIZATION GROUP

Our motivations underlying the random renormalization group (RRG) are twofold. The first objective is to design a unified framework applicable to both dynamics and structure renormalziation. The second target is to reduce the computation costs of renormalization to enable researchers to analyze ultra-large systems.

A. Problem setup

Although existing frameworks such as the phenomenological RG [2, 5, 6] and the Laplacian RG [11] provide deep insights into their corresponding research questions, we find them inpefect in the following aspects:

(1) The real space and momentum space approaches in the phenomenological RG [2, 5, 6] essentially rely on correlation and covariance matrices, which lead to a computational complexity of at least $O\left(N^2 D\right)$ in each step of renormalization (here N measures the number of units and D denotes the dimension of the data concerned). This is not ideal for ultra-large system analysis.

(2) The real space approach in the Laplacian RG [11] starts from calculating the matrix exponential in Eq. (8), which approximately requires a computational complexity of $O\left(rN^3\right)$ in most situations (e.g., Al-Mohy and Higham's scaling and squaring method [14] requires $r \simeq 15$). Meanwhile, the eigendecomposition in the momentum approach leads to a computational complexity of $O\left(N^3\right)$ in each step of renormalization. These two complexities are unacceptable in ultra-large system analysis.

(3) There exist gaps between structure and dynamics renormalization. Previous frameworks mainly focus on realizing one of these two analysis tasks and do not deal with another one directly. Certainly, a plain solution is to combine the ideas of the Laplacian RG [11] with those of the phenomenological RG [2, 5, 6]. As an instance, we consider a case of dynamics renormalization where each $Y_i^{(l)}$ denotes the time series of behaviours of unit $X_i^{(l)}$. One can use the covariance matrix, the correlation matrix, or the mutual information matrix measured on data $Y^{(l)}$ as the adjacency matrix to construct a network. Then, one can use the Laplacian RG [11] to renormalize it. If there exist two units, $X_i^{(l)}$ and $X_j^{(l)}$, that are grouped into a macro-unit, one can average between $Y_i^{(l)}$ and $Y_j^{(l)}$ to define the corresponding renormalized time series. Although this idea provides a way to use the Laplacian RG [11] in dynamics renormalization, it significantly increases the computational costs of renormalization (i.e., one needs to first deal with adjacency matrix construction with a complexity of at least $O\left(N^2D\right)$ and then deals with a complexity of $O\left(N^3\right)$ for the Laplacian RG [11]). Therefore, an appropriate solution remains to be explored.

Because generalizing the Laplacian RG [11] into dynamics renormalization leads to significantly increased computational costs (as shown in the analysis above), we turn to exploring another possibility instead: can we develop a structure renormalization framework in the sprit of the phenomenological RG [2, 5, 6]? We are interested in this possibility instead of following the Laplacian RG [11] mainly because of mathematical concerns: developing universal acceleration for the matrix-exponential- and eigendecomposition-based renormalization procedure is essentially hard from the perspective of computational mathematics. There do exist various optimization methods for these mathematical operations. However, these methods only hold for certain types of matrices, whose requirements are difficult to verify in complex system analysis. As for the phenomenological RG [2, 5, 6], we find possible directions to reduce its computational costs and generalize it to structure renormalization in the meantime.

B. Idea underlying the random renormalization group

Our idea is rooted in random projections [15] and hashing techniques [16], which seem to be far from statistical physics and have not been considered in developing renormalization frameworks yet. The key idea is to use these techniques to realize computation acceleration, reduce memory cost, and enhance the capacity to deal with non-linearity as much as possible. All these targets are achieved without doing harm to the physics validity of renormalization.

Defining features for structure To realize our idea, our first task is to define a phenomenological-RG-like framework that is applicable to structures. Let us recall that the real space approach of the phenomenological RG [2, 5, 6] starts with dealing $Y^{(l)}$, the feature matrix of system $X^{(l)}$. The definition of $Y^{(l)}$ in dynamics renormalization is natural, i.e., each $Y_i^{(l)}$ is the sequence of activities of unit $X_i^{(l)}$. However, there is no *a priori* definition of $Y^{(l)}$ in structure renormalization.

To resolve this question, we suggest to define $W_i^{(l)}$ as the set that includes all the adjacent units of $X_i^{(l)}$ and their corresponding connection strengths. For instance, let us consider a case where $X_i^{(l)}$ only has two neighbors, $X_a^{(l)}$ and $X_b^{(l)}$. The connection weight between $X_i^{(l)}$ and $X_a^{(l)}$ is $\kappa_{ia}^{(l)}$ while the connection weight between $X_i^{(l)}$ and $X_b^{(l)}$ is $\kappa_{ib}^{(l)}$. Then, we have $W_i^{(l)} = \left\{ \left(X_a^{(l)}, \kappa_{ia}^{(l)}\right), \left(X_b^{(l)}, \kappa_{ib}^{(l)}\right) \right\}$. It is clear that the Jaccard distance between $W_i^{(l)}$ and $W_j^{(l)}$ reflects the difference between $X_i^{(l)}$ and $X_j^{(l)}$ in terms of local topology (i.e., local adjacent relations) and connection strengths.

Each $W_i^{(l)}$ serves as the basis to define $Y_i^{(l)}$. The reason why we do not directly use $W_i^{(l)}$ as the feature of unit $X_i^{(l)}$ is that the number of neighbors may varies across different units (i.e., the system is heterogeneous). To avoid the potential difficulties caused by such heterogeneity and to reduce computational complexity, we apply the MinHash method [17–19] to hash each set $W_i^{(l)}$ as a feature vector, $Y_i^{(l)}$, of a given dimension. The MinHash method is efficient in estimating the Jaccard distance between sets [17, 18]. Specifically, the normalized XOR distance between $Y_i^{(l)}$ and $Y_j^{(l)}$ approximates the concerned Jaccard distance between $W_i^{(l)}$ and $W_j^{(l)}$ while realizing significant acceleration [17, 18]. After defining $Y^{(l)}$ to reflect local topology, we can build a real space approach for structure renormalization

in a manner similar to the phenomenological RG [2, 5, 6]. Because $Y^{(l)}$ is well-justified in dynamics renormalization, we do not need this pre-processing step.

Random projection for reducing computational costs and dealing with different correlations However, we can not directly adopt the original phenomenological RG due to its dependence of correlation and covariance matrices [2, 5, 6]. This dependence is known to create a high computational complexity in each step of renormalization. In the RRG, we use random projections [15] to further reduce computational costs. Specifically, the RRG supports to apply the signed random hyperplane projection [20], the signed random Fourier feature [21, 22], and the signed Cauchy projection [23] to map the real-valued feature matrix $Y^{(l)}$ to a binary representation $Z^{(l)}$ whose hash dimension is h. This procedure has multiple benefits:

(1) We can set a hash dimension h that is smaller than the dimension of each $Y_i^{(l)}$ to reduce memory costs during computation.

(2) We can still measure correlation between any pair $Y_i^{(l)}$ and $Y_j^{(l)}$ to realize the real space approach described in Eq. (2). Thus, our framework remains compatible with the phenomenological RG [2, 5, 6]. Moreover, a more substantial advantage is that our approach is no longer constrained to the linear correlation used in Eq. (2). By utilizing different random projections, our framework is enabled to measure both linear and nonlinear dependencies (i.e., in different kernel spaces) between $Y_i^{(l)}$ and $Y_j^{(l)}$, thereby significantly enhancing the descriptive capability of the RRG for complex systems.

(3) We can define the correlation-based counterpart of the Kadanoff block $V_i^{(l)}$ used in Eq. (2) more efficiently. This is because the projected binary representation $Z^{(l)}$ naturally supports an approximate nearest neighbor search, which is significantly faster than the plain search (even in small systems).

C. Framework of the random renormalization group

Our idea described above allows us to define a unified real space approach for both structure and dynamics renormalization. As an illustration, we let X be the input of the renormalization flow, i.e., $X^{(1)} = X$. In each l-th iteration ($l \geq 1$), the RRG implements the following procedures:

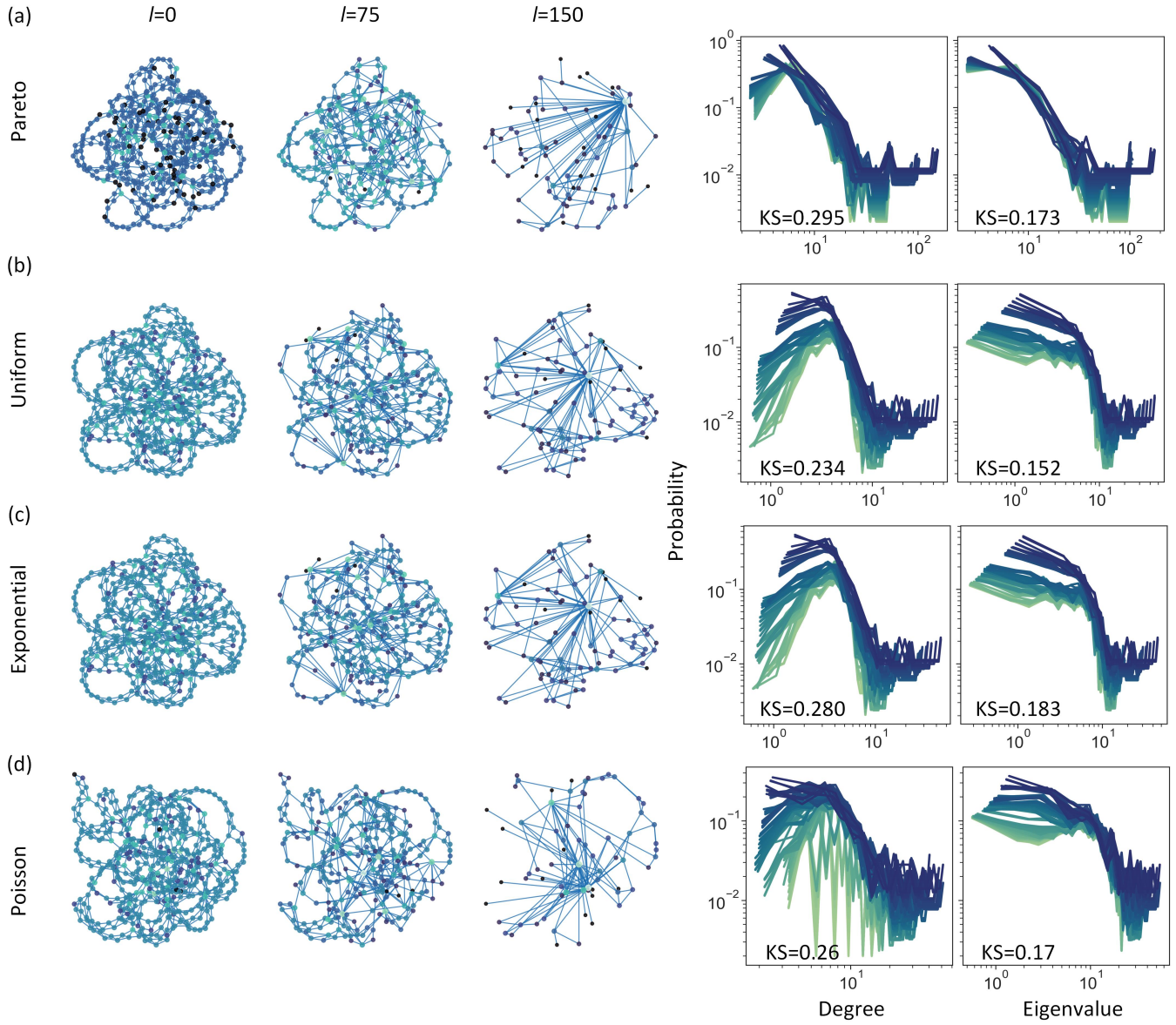
(1) Define a feature representation, $Y^{(l)}$, of $X^{(l)}$ such that each unit $X_i^{(l)}$ has a feature vector $Y_i^{(l)}$ to represent its properties. Normalize each $Y_i^{(l)}$ following certain criteria for preparation (note that this normalization is required by random projections [15]). As explained in Sec. II B, in this step, we create the possibility to realize structure renormalization in a phenomenological-RG-like manner [2, 5, 6] by defining $Y_i^{(l)}$ based on local topology and connection strengths.

(2) Apply a signed random projection to hash $Y^{(l)}$ as a binary representation $Z^{(l)}$ such that the Hamming distance between $Z_i^{(l)}$ and $Z_j^{(l)}$ approximates the correlation distance (i.e., a distance changes inversely with correlation) between $Y_i^{(l)}$ and $Y_j^{(l)}$ in a specific kernel space. This step creates the three benefits mentioned in Sec. II B.

(3) Realize an approximate nearest neighbor search on $Z^{(l)}$ to find the nearest neighbor of each unit $X_i^{(l)}$. All pairs of nearest neighbor relations are included in space $U^{(l)}$. The approximate nearest neighbor search on binary sequences is significantly faster than plain search on real-valued sequences.

(4) Define a null network, $G^{(l)}$, of all units (i.e., a network without edge). For dynamics renormalization, edges are added between all pairs of nearest neighbors in $U^{(l)}$. For structure renormalization, an edge is added between a pair of nearest neighbors in $U^{(l)}$ only if they are adjacent in $X^{(l)}$ too. After edge adding, every connected cluster, $C_k^{(l)}$, of network $G^{(l)}$ includes the units sharing strong correlations. Despite minor discrepancies in implementation details, this step essentially shares the same sprit with the reference network construction step in the real space approach in the Laplacian RG [11].

(5) Coarse grain the units in each connected cluster $C_k^{(l)}$ into a macro-unit $X_k^{(l+1)}$. For dynamics renormalization, macro-unit $X_k^{(l+1)}$ is defined with a summed feature vector $Y_k^{(l+1)} = \sum_{i \in I_k^{(l)}} Y_i^{(l)}$, where $I_k^{(l)}$ denotes the index set of all units in $C_k^{(l)}$. For structure renormalization, two macro-units are connected in $X^{(l+1)}$ if the units aggregated into them share at least one edge in $X^{(l)}$. The weight of this macro-connection is averaged across the aggregated edges (if weight information is necessary).

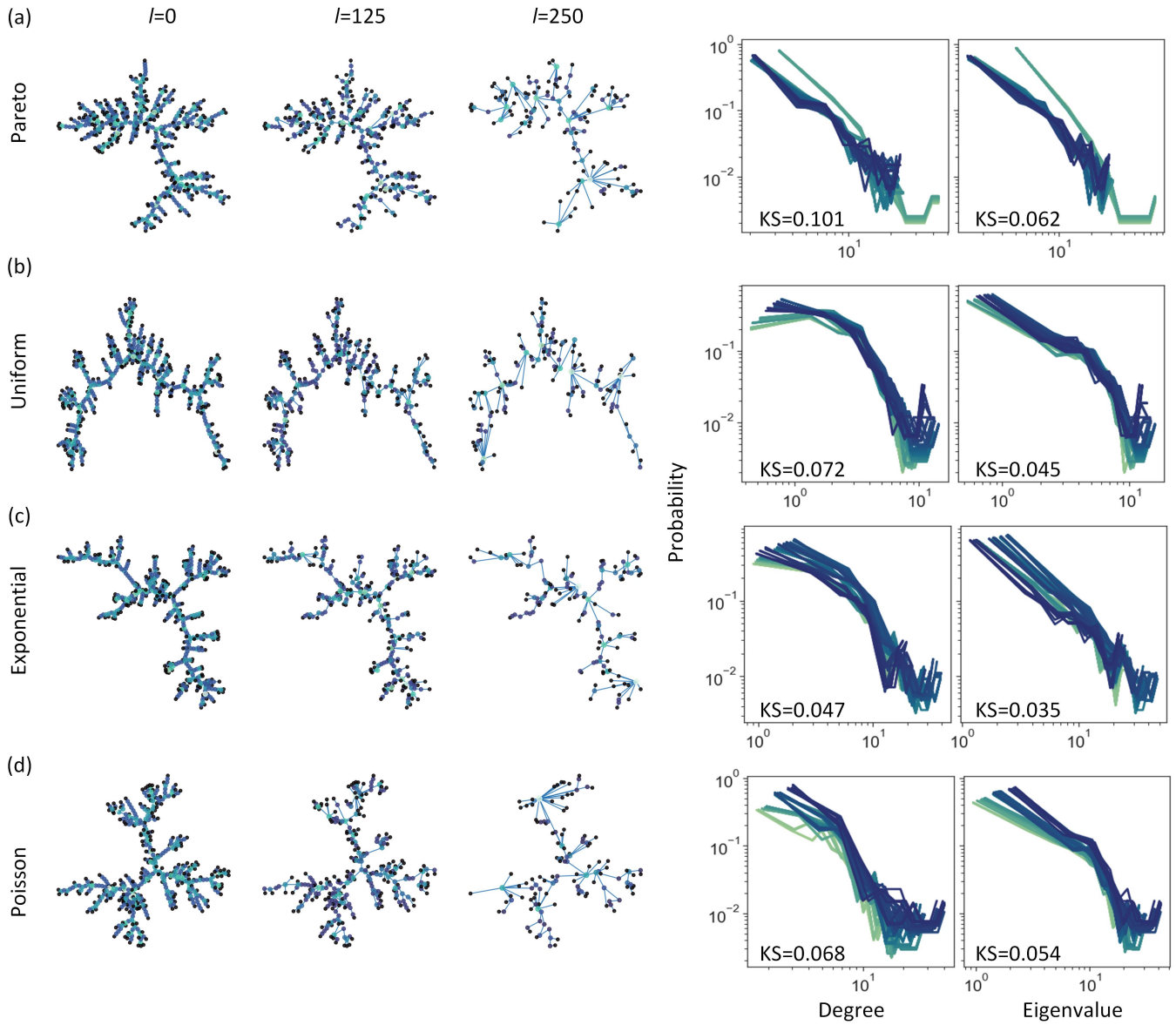


SFig. 1. Structure renormalization on the weighted Watts-Strogatz network (WS) [24]. Each WS network is generated by initializing a regular network where the degree of each unit is 6 and rewiring each edge according to a probability of $p = 0.3$. The system initially contains 500 units. We respectively use a Pareto distribution whose exponent is 2, a uniform distribution over $(0, 2)$, an exponential distribution whose exponent is 2, and a Poisson distribution whose exponent is 1 (we add the random samples generated by the Poisson distribution with 1 to avoid zero values) to generate edge weights. For each network replica, we use the RRG to renormalize it to generate the renormalization flow. Meanwhile, we also use the degree distribution of units and the Laplacian eigenvalue distributions as the macroscopic observables to implement the Kolmogorov-Smirnov (KS) analysis.

III. SUPPORTING RESULTS OF COMPUTATIONAL EXPERIMENTS

A. Structure renormalization: applicability on weighted structures

In the main text, we have mentioned that the set $W_i^{(l)}$ defined within our structure renormalization framework can incorporate weighting information (i.e., coupling strengths between units). For clarity of demonstration in the main text, we mainly focus on unweighted cases. In this section, we employ multiple random network models combined with various weight probability distributions to systematically demonstrate how the RRG can be extended to weighted

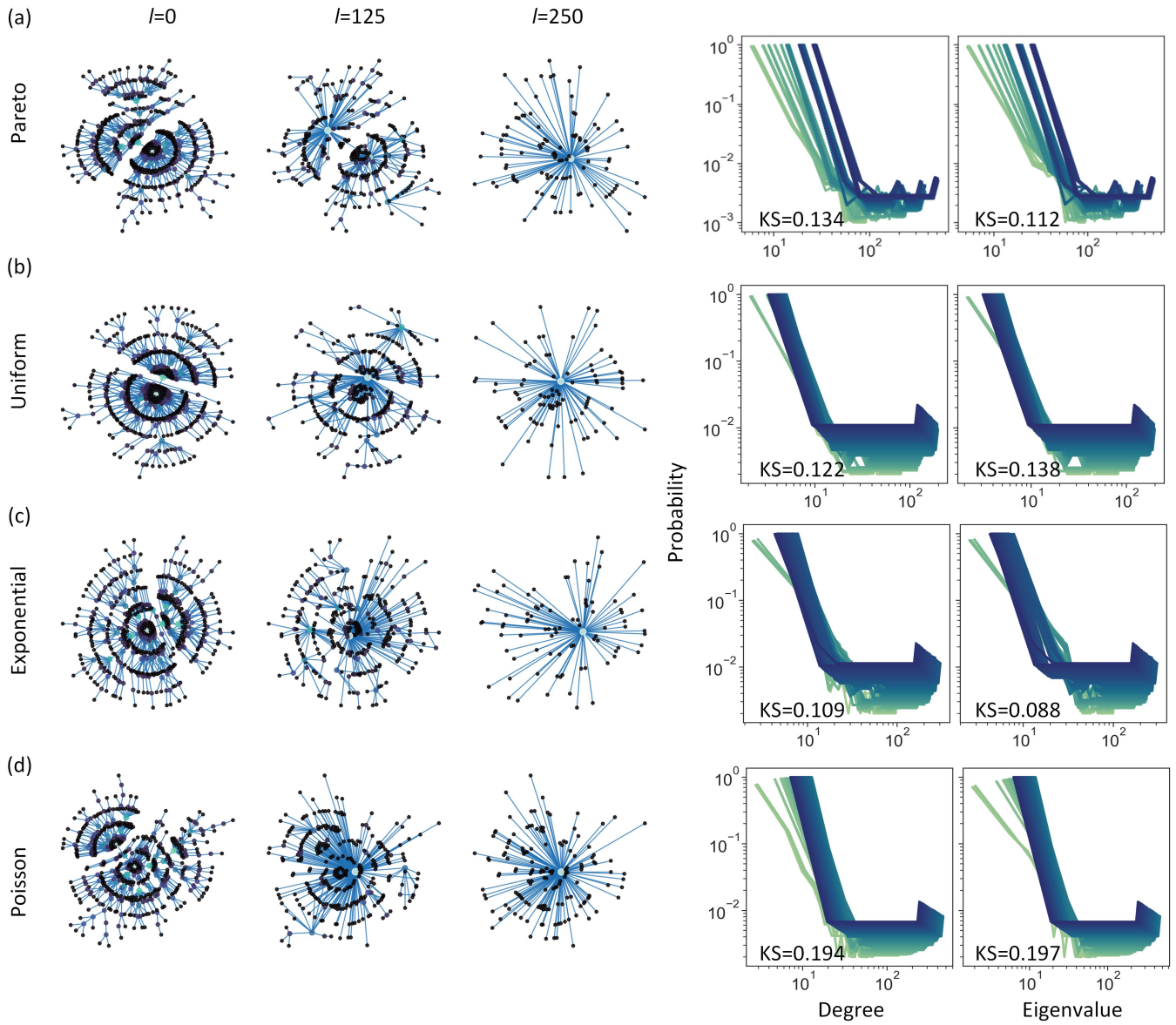


SFig. 2. Structure renormalization on the weighted the random tree (RT) [25]. Each RT network initially contains 500 units. We respectively use a Pareto distribution whose exponent is 2, a uniform distribution over $(0, 2)$, an exponential distribution whose exponent is 2, and a Poisson distribution whose exponent is 1 (we add the random samples generated by the Poisson distribution with 1 to avoid zero values) to generate edge weights. For each network replica, we use the RRG to renormalize it to generate the renormalization flow. Meanwhile, we also use the degree distribution of units and the Laplacian eigenvalue distributions as the macroscopic observables to implement the Kolmogorov-Smirnov (KS) analysis.

networks.

Specifically, we choose the Watts-Strogatz network (WS) [24], the random tree (RT) [25], and the Barabási-Albert network (BA) [26, 27] models. Given each random network model, we generate multiple replicas of it and assign each replica with the weights sampled following a certain probability distribution. In our experiment, we use the Poisson distribution, the uniform distribution, the exponential distribution, and the Pareto distribution to generate weight samples.

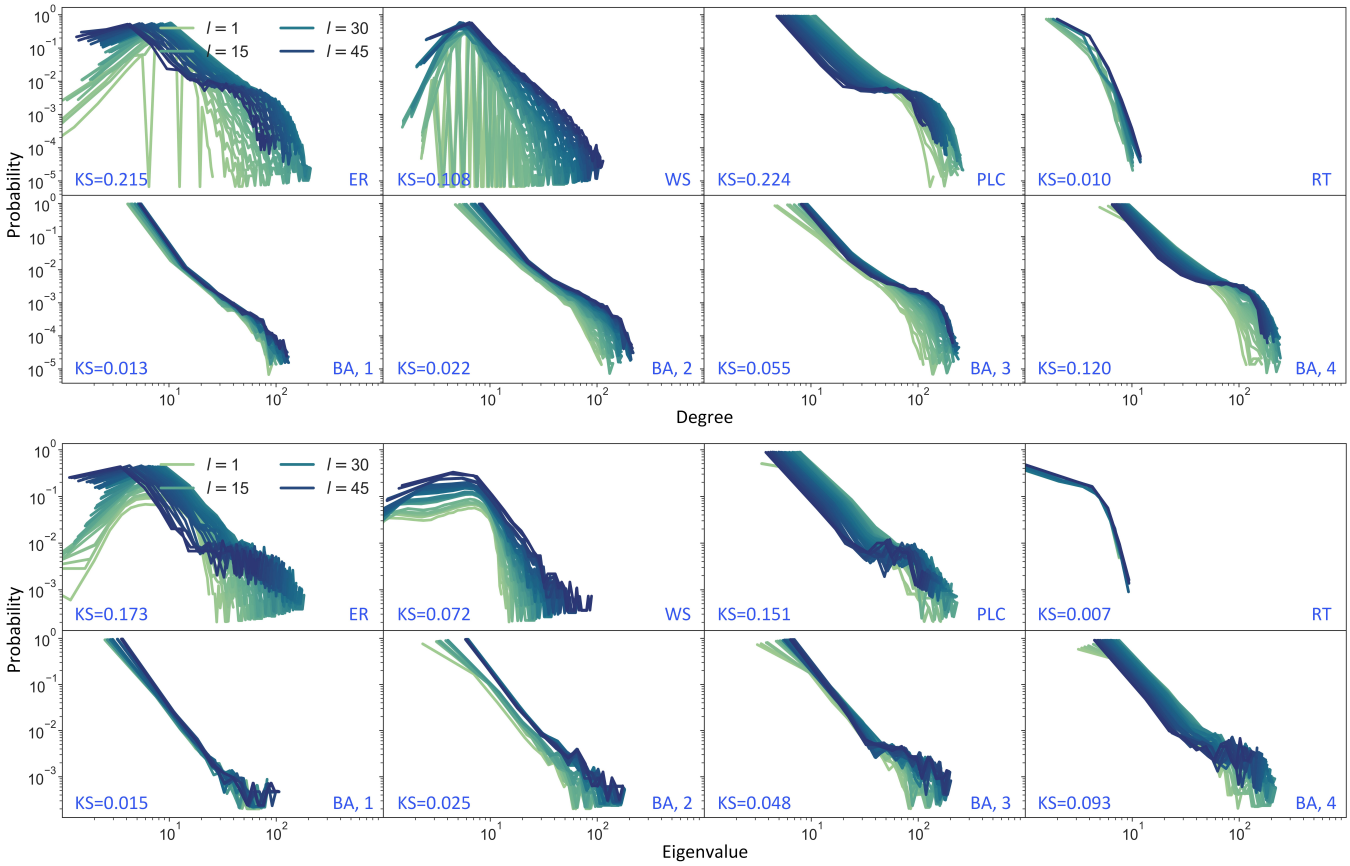
In SFigs. 1-3, we show the results of renormalization on these weighted random networks. Our analysis demonstrates that the RRG framework remains effective in performing renormalization for weighted network structures, while qualitatively preserving their critical topological information.



SFig. 3. Structure renormalization on the weighted the Barabási-Albert network (BA) [26, 27]. Each BA network is generated by a network growth process where the number of random edges to attach from a new unit to existing units is 1. Each system initially contains 500 units. We respectively use a Pareto distribution whose exponent is 2, a uniform distribution over $(0, 2)$, an exponential distribution whose exponent is 2, and a Poisson distribution whose exponent is 1 (we add the random samples generated by the Poisson distribution with 1 to avoid zero values) to generate edge weights. For each network replica, we use the RRG to renormalize it to generate the renormalization flow. Meanwhile, we also use the degree distribution of units and the Laplacian eigenvalue distributions as the macroscopic observables to implement the Kolmogorov-Smirnov (KS) analysis.

B. Structure renormalization: classifying scale-invariant, weakly scale-invariant, and scale-dependent structures

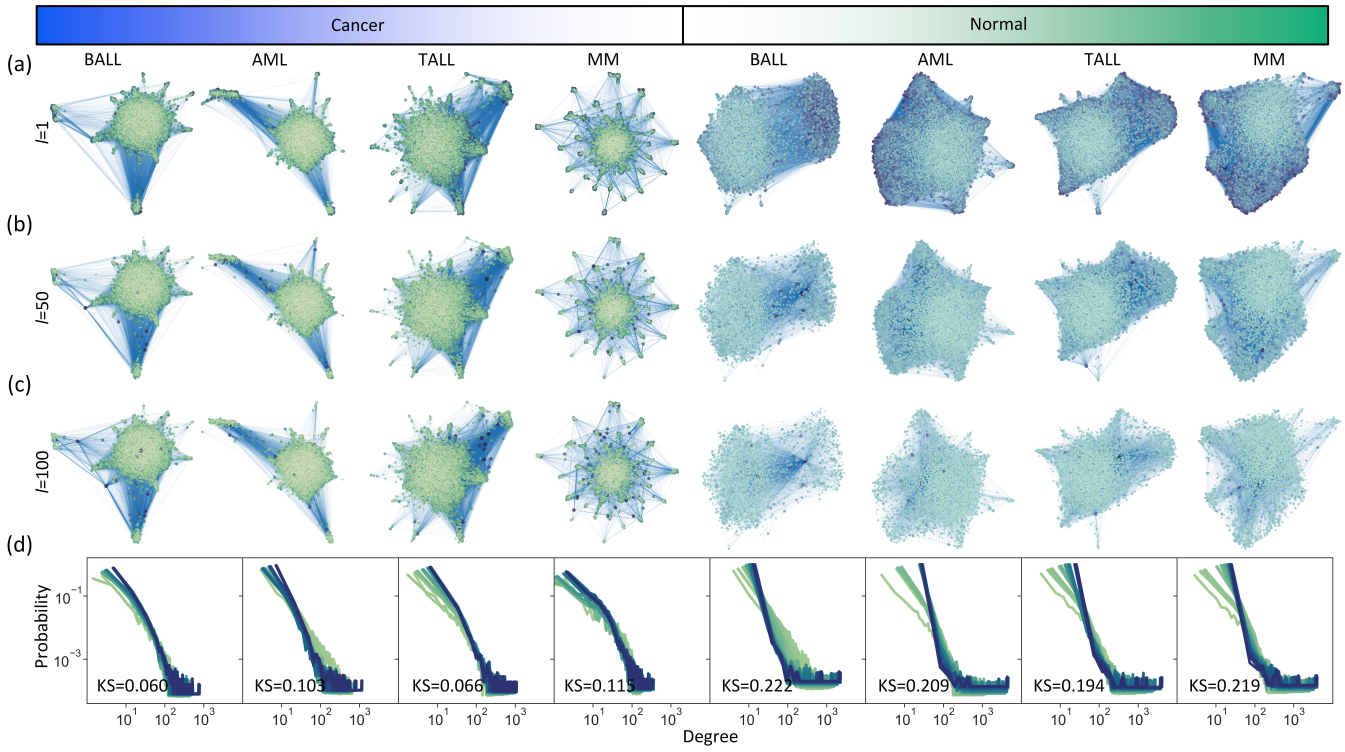
In SFig. 4, we validate this approach in classifying scale-invariant, weakly scale-invariant, and scale-dependent structures. Specifically, we use a RRG to renormalize different random network models and calculate the corresponding Kolmogorov-Smirnov statistics. Among those considered random networks, the Erdos-Renyi network (i.e., the classic binomial network) [28] and the Watts-Strogatz network (i.e., the model of small-world effects) [24] are representative scale-dependent structures. The Barabási-Albert network [26, 27] is known to be weakly scale-invariant (i.e., be conditionally scale-invariant in some cases) [11]. The random tree [25] is a typical model with scale-invariance.



SFig. 4. Macroscopic observables in structure renormalization. We generate the data of the Erdos-Renyi network (ER) [28], the Watts-Strogatz network (WS) [24], the power-law cluster network (PLC) [29], the random tree (RT) [25], and the Barabási-Albert network (BA) [26, 27]. For each random network model, we generate 300 replicas of it for subsequent processing, where each replica is initialized with 500 units before renormalization. Then, we use a RRG defined with a signed Cauchy projection and a target hashing dimension of 30 to renormalize these random networks for 50 iterations. The degree distribution of units and the Laplacian eigenvalue spectrum are used as two macroscopic observables to represent the system state in each iteration, which is average across all replicas before analysis. We show the degree distributions, the Laplacian eigenvalue distributions, and the corresponding Kolmogorov-Smirnov statistics (KS) obtained on a ER network (two units share an edge with a probability of $p = 0.02$), a WS network (generated by initializing a regular network where the degree of each unit is 6 and rewiring each edge according to a probability of $p = 0.3$), a PLC network (the number of random edges attached to each new unit is 5 and the probability of adding a triangle after adding a random edge is 0.02), and a RT. The system sizes of these four network models are reduced by 93.19%, 50.69%, 85.07%, and 15.09% after renormalization on average. Then, we report the results obtained on a family of BA networks, where "BA, n" denotes a BA network model generated by a network growth process where the number of random edges to attach from a new unit to existing units is n. The system sizes of these four network models are reduced by 18.16%, 38.17%, 57.11%, and 73.13% after renormalization on average.

The power-law cluster network [29] is a model that combines the properties of the Watts-Strogatz network and the Barabási-Albert network and, therefore, interpolates between scale-dependence and weak scale-invariance.

As shown in SFig. 4, the Kolmogorov-Smirnov statistics measured on scale-invariant structures are smallest while those obtained on scale-dependent structures are largest. Weakly scale-invariant structures and the structures interpolating between scale-dependence and weak scale-invariance have relatively small Kolmogorov-Smirnov statistics. These results are consistent with our expectation that a scale-invariant structure is close to the fixed point of the renormalization flow. A scale-dependent structure is far from the fixed point and persistently evolves under the scale transformation by a renormalization group. It should be noted that our analysis in SFig. 4 uses finite-size networks rather than idealized infinite networks. During the renormalization process, the progressive reduction in network size leads to increasingly irregular degree distributions and Laplacian eigenvalue spectra due to limited sample size. Consequently, even for scale-invariant RTs, the Kolmogorov-Smirnov statistics asymptotically approach zero rather than vanishing completely. This phenomenon arises from finite-size effects, which are inherently unavoidable when studying real-world empirical networks.



SFig. 5. Structure renormalization on the real data (additional results). Sub-figures respectively show the renormalization flows, degree distributions, and Kolmogorov-Smirnov (KS) statistics of B-cell acute leukemia (BALL), T-cell acute leukemia (TALL), acute myeloid leukemia (AML), and multiple myeloma (MM). In cancer phenotype, the system sizes of BALL, AML, TALL, MM are reduced by 21.31%, 26.02%, 26.88%, and 34.81% after renormalization. In healthy bone marrow, the system sizes of BALL, AML, TALL, MM are reduced by 44.35%, 48.08%, 39.64%, and 43.84% after renormalization.

Based on these results, the applicability of our proposed approach in analyzing the cross-scale properties of structures can be principally demonstrated.

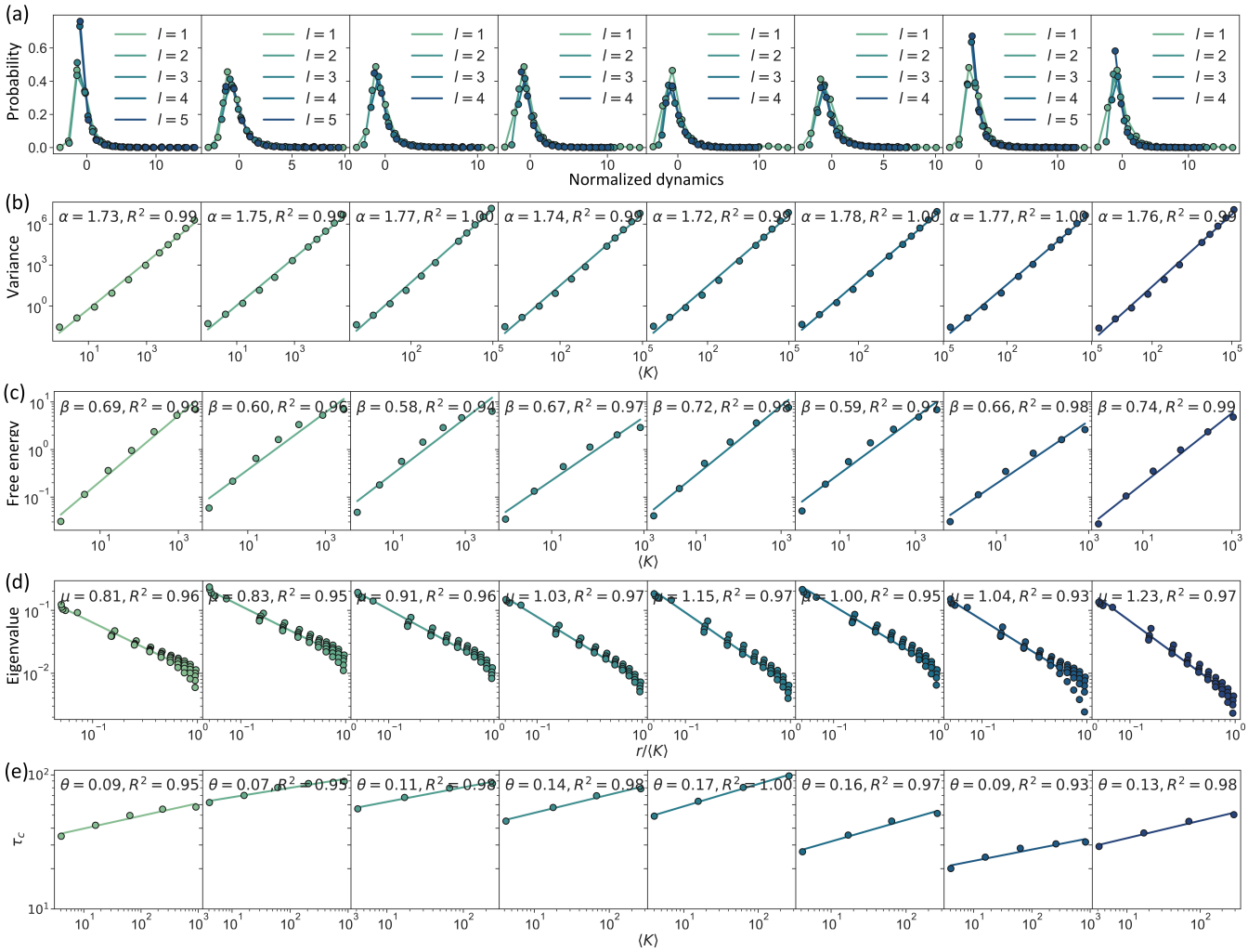
C. Structure renormalization: classifying real networks in medical data

Here we use a medical science data set to suggest the possibility for the RRG to serve as a data mining tool to classify real networks. The data set we use contains the gene co-expression networks of cancer phenotypes and healthy bone marrow released by Ref. [30], which are estimated from the expression profiles of different gene couples. The considered cancer phenotypes include B and T-cell acute leukemia, acute myeloid leukemia, as well as multiple myeloma. Note that the layouts of these large networks in SFig. 5 are derived using the combination of Graph2Vec [31] and UMAP embedding [32].

We find that the RRG helps to distinguish between cancer phenotypes (e.g., B and T-cell acute leukemia, acute myeloid leukemia, and multiple myeloma) and healthy bone marrow according to the renormalization flows of their gene co-expression networks [30]. In SFig. 5, all gene co-expression networks of normal bone marrow consist of single connected components with rich connections, whose degree distributions change across different iterations of the RRG. As comparisons, their counterparts under cancer conditions feature more sparse connectivity structures and maintain more robust during renormalization. Consequently, the closeness to scale-invariance property of a gene co-expression network under the RRG transformation may serve as an auxiliary criterion to detect cancer phenotypes.

D. Dynamics renormalization: whole brain dynamics analysis

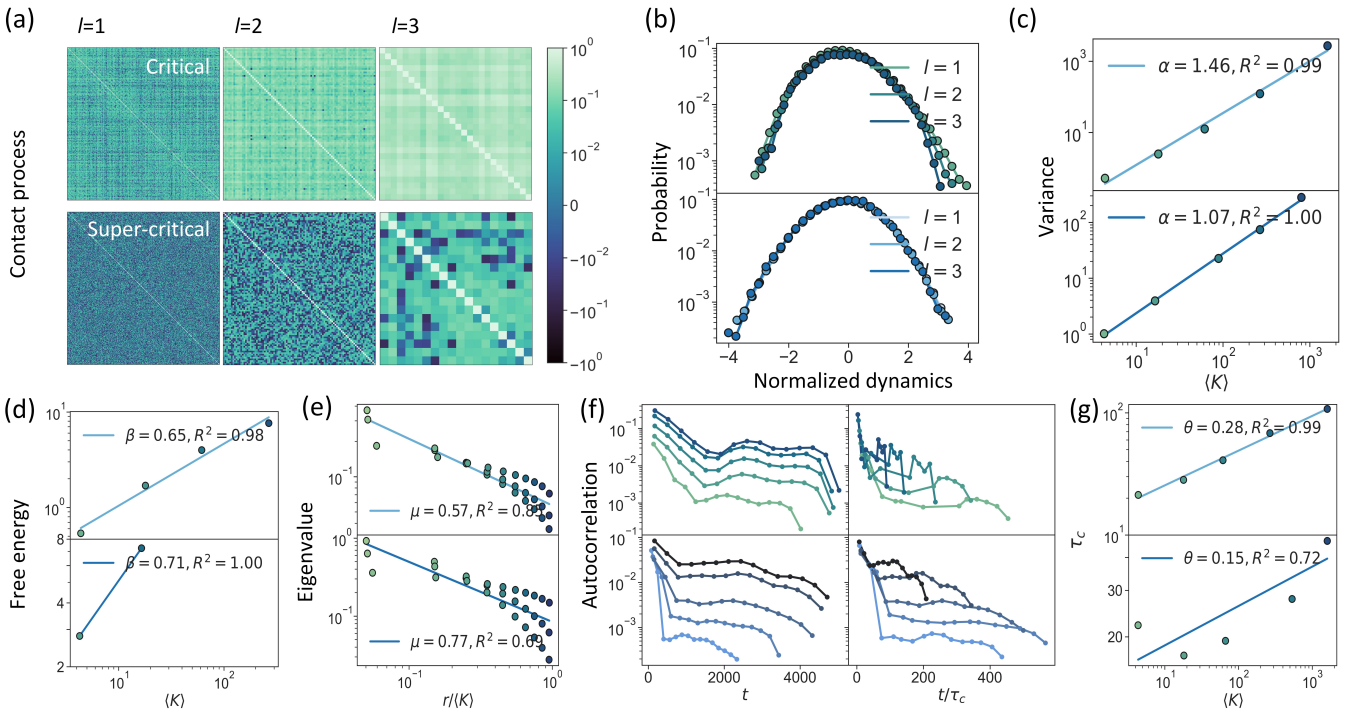
To demonstrate its applicability to complex real-world systems, we employ the RRG framework to analyze whole-brain dynamics in zebrafish larvae [33] (all settings of the RRG keep the same as those in contact process analysis).



SFig. 6. Dynamics renormalization on the real data (additional results). (a) The probability distributions of normalized dynamics of eight zebrafish larva brains during renormalization, where each column corresponds to the whole-brain dynamics data of one brain. (b-e) The scaling behaviours of whole-brain dynamics characterized by exponents α, β, μ, and θ. Scatters denote the observed data in computational experiments and lines denote the fitted scaling laws. The accuracy of fitting is measured by the R-square.

Eight brains are analyzed, where the activities of 40709 ± 13854 neurons during 1514 ± 238 seconds are recorded (the time sampling rate is $\simeq 4$).

As shown in SFig. 6(a), the renormalization flows of all brains converge to certain non-Gaussian fixed points, rejecting the possibility that the whole-brain dynamics proceeds with weakly correlated or independent neurons. The properties of non-trivially correlated neural activities are reflected by scaling features, which are estimated from the data with small errors. Exponents α and β generally suggest the non-negligible correlations among neurons (see SFigs. 6(b-c)). These correlations are self-similar since a power-law scaling phenomenon depending on mean fractional rank, $r/\langle k \rangle$, exists in every covariance matrix eigenvalue spectrum as a consequence of scale-invariant propagator (see SFig. 6(d)). In SFig. 6(e), we observe the dynamic scaling of temporal correlations, whose exponents are numerically similar to the results of a phenomenological RG implemented on mouse brain dynamics [6]. Our analysis reveals systematic discrepancies in the scaling behaviors between the two-dimensional contact process and whole-brain dynamics. These differences likely stem from the prevalence of long-range neuronal interactions, such as the small-world connectivity commonly observed in animal brains. The RRG framework is well-suited for studying systems with long-range interactions, as its coarse-graining procedure for dynamical analysis is inherently independent of initial spatial configurations (certainly, spatial constraints can be easily added into $Y^{(l)}$ if necessary).



SFig. 7. Dynamics renormalization on the real data (additional results). (a) The renormalization flows of a contact process with 1600 units under critical and super-critical conditions, where we use linear correlation matrices to represent system states during renormalization. (b) The probability distributions of normalized dynamics during renormalization. (c-e) The scaling behaviours of the contact process characterized by exponents α, β, and μ. (f) The existence and vanishing of universal collapse in the autocorrelation functions of critical and super-critical contact processes. (g) The scaling behaviour of the contact process characterized by exponent θ. Note that the upper rows in (b-g) share a common color map and correpond to the critical condition. The lower rows in (b-g) share a common color map and correspond to the super-critical condition. In (c-e) and (g), scatters denote the observed data in computational experiments and lines denote the fitted scaling laws. The accuracy of fitting is measured by the R-square. Please note that the results shown in (b-g) are all obtained based on the structure renormalization on the weighted structures defined by the covariance matrices of the two-dimensional contact process [34] under different settings.

E. Dynamics renormalization: analyzing directed percolation in the way of structure renormalization

In this section, we investigate an alternative approach using the covariance matrices measured on the two-dimensional contact process [34] under different settings. The control parameter of contact process is the spreading rate. As this parameter approaches to 1.649, the contact process exhibits an absorbing phase transition [34]. If the control parameter continues to increase, the contact process arrives in its super-critical phase [34].

Specifically, we take the absolute values of these covariance matrices as weighted adjacency matrices to directly define the corresponding weighted structures. We then apply RRG to renormalize these structures. Based on the renormalization results, we still compute scaling behaviors just like what we do during dynamics renormalization. Finally, we compare the obtained scaling analysis results with existing studies to evaluate the rationale of this approach. Based on the comparison, we can validate the possibility of realizing scaling analysis based on structure renormalization.

As shown in SFig. 7(a), we show how this new pipeline preserves the non-trivial correlation patterns (represented by correlation matrices, i.e., a kind of normalized covariance matrices) among units under both critical and super-critical conditions. Then, we use the covariance matrices as weighted adjacency matrices to directly define the weighted structures associated with the two-dimensional contact process. In Fig. 7(b), we show that the renormalization flow of this pipeline (we use the signed Cauchy projection and define the dimension of $Z^{(l)}$ as 100) under the critical condition has a non-Gaussian fixed point (i.e., the probability distribution of the normalized dynamics with a high skewness). On the other hand, the renormalization flow of the RRG under the super-critical condition has a fixed point whose distribution peak approaches zero (i.e., a generally bell-shaped distribution). These results are consistent with the findings of Ref. [5]. In SFigs. 7(c-g), we estimate the concerned scaling features from the data, which can be understood in a manner similar to what we have shown in Fig. 4 in the main text. According to our results, we can

obtain the values of scaling exponents, β and μ, that numerically satisfy the scaling relation $\eta = d - 2 + \beta/\nu_\perp$, where $\eta = 2 - d\mu$, notion $d = 2$ measures the dimension, and $\nu_\perp \simeq 0.733$ [34, 35]. As shown by the results of the critical state, the implemented pipeline derives $\eta \simeq 0.86$ and $d - 2 + \beta/\nu_\perp \simeq 0.88$, which is consistent with the scaling relation with reasonable errors. Furthermore, Fig. 6(g) reveals a dynamic scaling phenomenon characteristic of the critical state. In contrast, there is no dynamic scaling in the super-critical situation and we can estimate a value θ that is closer to zero. These findings collectively suggest the capacity of the new pipeline to distinguish between critical and non-critical dynamical regimes.

While our results empirically validate the feasibility of performing scaling analysis through structure renormalization, we caution against its practical adoption for two key reasons. First, this approach incurs substantially higher computational overhead per renormalization step compared to direct dynamics renormalization using the RRG, primarily due to the necessity of constructing covariance matrices (or equivalent coupling metrics) between system units. Second, the accuracy of the resulting scaling analysis critically depends on the specific definition of the weighted adjacency matrix. Although our experiments have achieved satisfactory results using the absolute values of covariance matrices, this success does not generalize to alternative weighting schemes without rigorous theoretical justification.

IV. CODE IMPLEMENTATION OF THE RANDOM RENORMALIZATION GROUP

The RRG is programed in Python, whose open-source code can be seen in https://github.com/doloMing/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
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
```

If the PyPI is used, one can easily install the library of the RRG via

```
pip install random-renormalization-group
```

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 [20], the signed random Fourier feature [21, 22], or the signed Cauchy projection [23], 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,Weighted=False):
    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)
            if len(X_New)==1:
                break
        elif Data_Type=="Structure":
            X_New, Corase_ID=Network_Renormalization_Function(X_Current,TargetDim,Iter,Method_Type,
    Weighted)
            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

# Generate a random tree with 10000 units, and assign the weights by a power-law distribution
tree = nx.random_tree(10000)
nx.set_edge_attributes(tree, 1, 'number')
size = tree.number_of_edges()
alpha = 2.0
weights = np.random.pareto(alpha, size) + 1
for i, (u, v) in enumerate(tree.edges()):
    tree[u][v]['weight'] = weights[i]

RG_Flow,Tracked_ID_list=Renormalization_Flow(tree,200,100,"Cauchy_Kernel","Structure",Weighted=True
    ) # Run a RRG for 200 iterations, where the dimension of hased binary vectors is 100 and weight
    information is considered
```

```
20
21 ## Dynamics renormalization
22 X_Initial = np.random.randn(10000, 50000) # Generate a system with 10000 units, where each unit
       exhibits random dynamics for 50000 time steps
23
24 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
25
26 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
27
28 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)
```

```
        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,Weighted):
    UnitNum=nx.number_of_nodes(X_Current)
    if Weighted:
        Z=Weighted_Min_Hashing(nx.adjacency_matrix(X_Current).toarray(),TargetDim)
    else:
        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))
                if Weighted:
                    for Nei in Neighbors:
                        if Unit0!=Nei:
                            if X_New.has_edge(Unit0, Nei):
                                n1 = X_New[Unit0][Nei]['number']
                                n2 = X_New[Unit][Nei]['number']
                                X_New[Unit0][Nei]['weight'] = (X_New[Unit0][Nei]['weight'] * n1 +
    X_New[Unit][Nei]['weight'] * n2) / (n1 + n2)
                                X_New[Unit0][Nei]['number'] += n2
                            else:
                                X_New.add_edge(Unit0, Nei, weight=X_New[Unit][Nei]['weight'],
    number=X_New[Unit][Nei]['number'])
                else:
                    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,Weighted=False):
    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)
            if len(X_New)==1:
                break
        elif Data_Type=="Structure":
            X_New, Corase_ID=Network_Renormalization_Function(X_Current,TargetDim,Iter,Method_Type,
    Weighted)
            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
```

V. 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 [36, 37]

```
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 autocorrelation 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]
```

```
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              Neededx=np.where((x>=bins[ID3])&(x<=bins[ID3+1]))[0]
100             Meanx[ID3]=np.mean(np.array(x)[Neededx])
101             Meany[ID3]=np.mean(np.array(y)[Neededx])
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]))]
```

```
    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] G. Jona-Lasinio, Renormalization group and probability theory, Physics Reports **352**, 439 (2001).
[2] L. Meshulam and W. Bialek, Statistical mechanics for networks of real neurons, arXiv preprint arXiv:2409.00412 (2024).
[3] A. Pelissetto and E. Vicari, Critical phenomena and renormalization-group theory, Physics Reports **368**, 549 (2002).
[4] A. Gabrielli, D. Garlaschelli, S. P. Patil, and M. Serrano, Network renormalization, Nature Reviews Physics https://doi.org/10.1038/s42254-025-00817-5 (2025).
[5] G. Nicoletti, S. Suweis, and A. Maritan, Scaling and criticality in a phenomenological renormalization group, Physical Review Research **2**, 023144 (2020).
[6] L. Meshulam, J. L. Gauthier, C. D. Brody, D. W. Tank, and W. Bialek, Coarse graining, fixed points, and scaling in a large population of neurons, Physical review letters **123**, 178103 (2019).
[7] S. Bradde and W. Bialek, Pca meets rg, Journal of statistical physics **167**, 462 (2017).
[8] K.-I. Goh, G. Salvi, B. Kahng, and D. Kim, Skeleton and fractal scaling in complex networks, Physical review letters **96**, 018701 (2006).
[9] M. Samsel, K. Makulski, M. Łepek, A. Fronczak, and P. Fronczak, Towards fractal origins of the community structure in complex networks: a model-based approach, arXiv preprint arXiv:2309.11126 (2023).
[10] H. D. Rozenfeld, C. Song, and H. A. Makse, Small-world to fractal transition in complex networks: a renormalization group approach, Physical review letters **104**, 025701 (2010).
[11] P. Villegas, T. Gili, G. Caldarelli, and A. Gabrielli, Laplacian renormalization group for heterogeneous networks, Nature Physics **19**, 445 (2023).
[12] P. Moretti and M. Zaiser, Network analysis predicts failure of materials and structures, Proceedings of the National Academy of Sciences **116**, 16666 (2019).
[13] R. Burioni and D. Cassi, Random walks on graphs: ideas, techniques and results, Journal of Physics A: Mathematical and General **38**, R45 (2005).
[14] A. H. Al-Mohy and N. J. Higham, A new scaling and squaring algorithm for the matrix exponential, SIAM Journal on Matrix Analysis and Applications **31**, 970 (2010).
[15] M. W. Mahoney *et al.*, Randomized algorithms for matrices and data, Foundations and Trends® in Machine Learning **3**, 123 (2011).
[16] D. Cai, A revisit of hashing algorithms for approximate nearest neighbor search, IEEE Transactions on Knowledge and Data Engineering **33**, 2337 (2019).
[17] A. Z. Broder, On the resemblance and containment of documents, in *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171)* (IEEE, 1997) pp. 21–29.
[18] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher, Min-wise independent permutations, in *Proceedings of the thirtieth annual ACM symposium on Theory of computing* (1998) pp. 327–336.
[19] S. Ioffe, Improved consistent sampling, weighted minhash and l1 sketching, in *2010 IEEE international conference on data mining* (IEEE, 2010) pp. 246–255.
[20] S. S. Vempala, *The random projection method*, Vol. 65 (American Mathematical Soc., 2005).
[21] X. Li and P. Li, Signrff: Sign random fourier features, Advances in Neural Information Processing Systems **35**, 17802 (2022).

[22] A. Rahimi and B. Recht, Random features for large-scale kernel machines, Advances in neural information processing systems **20** (2007).
[23] P. Li, G. Samorodnitsk, and J. Hopcroft, Sign cauchy projections and chi-square kernel, Advances in Neural Information Processing Systems **26** (2013).
[24] D. J. Watts and S. H. Strogatz, Collective dynamics of 'small-world'networks, nature **393**, 440 (1998).
[25] M. Drmota, *Random trees: an interplay between combinatorics and probability* (Springer Science & Business Media, 2009).
[26] R. Albert and A.-L. Barabási, Statistical mechanics of complex networks, Reviews of modern physics **74**, 47 (2002).
[27] A.-L. Barabási, Scale-free networks: a decade and beyond, science **325**, 412 (2009).
[28] P. Erdős, A. Rényi, *et al.*, On the evolution of random graphs, Publ. Math. Inst. Hung. Acad. Sci **5**, 17 (1960).
[29] P. Holme and B. J. Kim, Growing scale-free networks with tunable clustering, Physical review E **65**, 026107 (2002).
[30] A. K. Nakamura-García and J. Espinal-Enríquez, The network structure of hematopoietic cancers, Scientific Reports **13**, 19837 (2023).
[31] A. Narayanan, M. Chandramohan, R. Venkatesan, L. Chen, Y. Liu, and S. Jaiswal, graph2vec: Learning distributed representations of graphs, arXiv preprint arXiv:1707.05005 (2017).
[32] L. McInnes, J. Healy, and J. Melville, Umap: Uniform manifold approximation and projection for dimension reduction, arXiv preprint arXiv:1802.03426 (2018).
[33] T. L. van der Plas, J. Tubiana, G. Le Goc, G. Migault, M. Kunst, H. Baier, V. Bormuth, B. Englitz, and G. Debrégeas, Neural assemblies uncovered by generative modeling explain whole-brain activity statistics and reflect structural connectivity, Elife **12**, e83139 (2023).
[34] J. Marro and R. Dickman, Nonequilibrium phase transitions in lattice models, Nonequilibrium Phase Transitions in Lattice Models (2005).
[35] R. Dickman and M. M. de Oliveira, Quasi-stationary simulation of the contact process, Physica A: Statistical Mechanics and its Applications **357**, 134 (2005).
[36] R. Simard and P. L'Ecuyer, Computing the two-sided kolmogorov-smirnov distribution, Journal of Statistical Software **39**, 1 (2011).
[37] 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 Computing 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.