Running Time Comparison of Two Realizations of the Multifractal Network Generator Method

Árpád Horváth

Óbuda University, Alba Regia University Centre, Székesfehérvár, Hungary
horvath.arpad@arek.uni-obuda.hu

Abstract: Over the last decade a lot of common properties were found in complex networks in several fields such as sociology, biology and computer engineering. Recently, the multifractal network generator method has been developed, and it seems to be a promising way to generate networks with prescribed statistical properties. For educational purposes, however, it would be adequate to create an easy-to-use redevelopment framework. Therefore, a software package had been developed in Python language that can generate a network with a given degree distribution or a given average degree using the multifractal generator method. This package is a part of the cxnet framework, which itself is suitable for educational applications. The present paper discusses the reasons why this framework was developed in Python. Those parts of the program that need longer running times were identified and rewritten in C++. Running times of the generations were measured, changing several parameters, and the new version turned out to be an order of magnitude faster.

Keywords: complex network; graph theory; multifractal; software

1 Introduction

Networks have a collection of entities, called nodes. These nodes can be connected or not, so the networks can be described as a graph in every moment. Complex networks are very large networks with a usually different structure from that of the random network. One of the aims of the science of complex networks is to study the general properties of real networks.

There are a lot of networks in the fields of engineering and informatics (the World Wide Web, the Internet), biology and medicine (network of protein interactions, the food chain) and sociology (acquaintances). Over the last decade, many networks and network models have been studied [1, 8].

To study the general properties of networks, one usually needs a method to create networks with prescribed properties. To create such networks, one can use optimization, which means that we change some parameters to approach the

properties we want to achieve. A promising optimizing method is the multifractal network generator [9]. This was improved to create less isolated nodes [10]; however at the size of real networks, the original method is reasonable. Using the multifractal network generator, a broad range of networks with arbitrary properties can be generated. The entropy of such generated networks is bigger than that of the other usually-used models, such as the Erdős-Rényi model, the small world model and Barabási-Albert model [4]. With this method, we can set more than one target property, for example a power-law function as the degree distribution and a clustering coefficient decreasing inversely proportional to the degrees of the nodes. These two properties can be found in many real-world networks and in the resulting networks of the hierarchical model [11]. Our goal is to develop an educational framework that is suitable for generating and analyzing networks, and then students would be able to develop standalone functions to extend the possibilities of the framework. The framework has been implemented in Python language, but some parts were written in C++ language as well. In this paper we describe the method and compare the running times of the two versions: the one written in pure Python and the other, where the calculation of the degree distribution is written in C++.

2 The Multifractal Network Generation Method

The method of generating networks with the usage of multifractals is described in detail in the article of Palla et al. [9].

In multifractal network generation, the generating measure is a central concept. The generating measure is a probability measure defined on the $[0,1[\times [0,1[$ unit square. A network can be generated from the generating measure in two steps. The first step is to create a link probability measure with the iteration of the generating measure. In the second step the program creates links between the nodes using the link probability measure. During generation, however, the program does not need to generate any networks; it calculates the estimated properties of the network from the generating measure.

2.1 Generating Measure and Link Probability Measure

Both the x and y axes of the unit squares are divided into m not necessarily equal intervals to define a generating measure. In our version, the x and y axes have the same division points. With this division we created m^2 rectangles on the unit square. Probabilities p_{ij} are assigned to each of the rectangles in a symmetric fashion,

$$p_{ij} = p_{ji}, \quad \sum_{i,j=0}^{m-1} p_{ij} = 1 \tag{1}$$

The probability assigned to the rectangle at the origin is denoted by the p_{00} and the one at the opposite corner is $p_{m-1,m-1}$.

The Kth iteration of the generating measure means a unit square divided into rectangles with assigned probabilities, as in the generating measure, but with $m^k \times m^k$ rectangles. The first iteration ($K = 1$) by definition gives the generating measure itself.

For the case of $K > 1$, we obtain the division points from the division points of the (K-1)st iteration by dividing each of its intervals into m subintervals, where the length of subintervals are proportional to the length of intervals of the original generating measure.

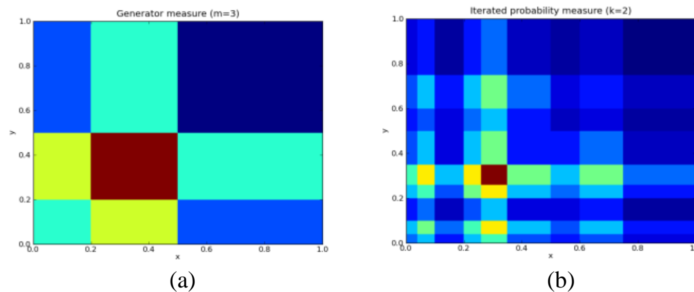
The $p_{ij}(K)$ probabilities of the Kth iteration can be calculated as

$$p_{ij}(k) = \prod_{q=1}^{K} p_{i_q j_q}, \tag{2}$$

where

$$i_q = \lfloor \frac{i \bmod m^{K-q+1}}{m^{K-q}} \rfloor \tag{3}$$

The notation (i mod d) means that the remainder of the integer division i/d and $\lfloor x \rfloor$ denotes the floor (integer part) of x. Analogous equation to (3) gives j_q as well.



(a) (b)

Figure 1

A generating measure (a) with the division points 0.2 and 0.5, and the link probability measure resulted by two iterations (b)

2.2 Generating the Network

The generation of networks proceeds in two steps. The first step is the iteration of the generating measure to get the link probability measure. The second one is the generation of the network from the link probability measure obtained after the iterations. The latter goes as follows.

First the number of iterations (K) and the number of nodes (N) in the network need to choose. If one axis of the generating measure is divided into m intervals with the division points, there will be m^K intervals in one axis of the link probability measure. We assign to each node with index l ($l \in [1, N]$ integer) a r_l

random value from a uniform distribution on the [0,1[interval. We determine the i_l index of the interval where r_l is located ($i_l \in [0, m^{K-1}]$).

For all pairs of the nodes we determine whether to connect the nodes or not. If the r_{l1} and r_{l2} random number belonging to the two nodes falls into the intervals i_{l1} and i_{l2}, respectively, we connect the two nodes with the probability $p_{i_{l1},i_{l2}}(K)$, where the values of $p_{ij}(K)$ are the probabilities after the Kth iteration of the generating measure defined in equation (3).

2.3 Adjusting the Generating Measure

The creation of the generating measure can be shown as an annealing process where the energy of the generating measure closes to the minimum as the temperature decreases.

To create a generating measure that gives a network with a given target property, we need to define an energy function (a non-negative function) that measures the goodness of the generating measure. The smaller energy, the closer the network created from the generating measure to the one with the target property.

After giving the m numbers of intervals on one axis and the K number of iteration, our program starts with equal probabilities and equal interval lengths on the axes. In each step it either relocates a division point or changes the probabilities. Then it calculates the energy belonging to the generating measure. If this E' energy is smaller than that belonging to the network of the existing generating measure E, than it changes the generating measure to the new one and stores the energy. If $E' > E$, then the new generating measure will be accepted with the probability

$$P(T) = \exp\left(-\frac{E'-E}{T}\right), \tag{4}$$

and rejected with $1 - P(T)$ probability. The arbitrary parameter T plays the role of temperature (in units of the energy).

Decreasing the temperature slowly, the generating process has the possibility to escape from local minima. The smaller the temperature, the more changes will be rejected, and the network converges to that with the target property.

2.4 Calculating the Degree Distribution

One of the targets of the generation can be the degree distribution. The degree distribution $p(k)$ is a function of degree k giving the probability of a node having the degree k. The expected values of a generating measure can be calculated as

$$p(k) = \sum_{i=0}^{m^k} p_i(k) l_i, \tag{5}$$

where

$$p_i(k) = \frac{<k_i>^k}{k!} e^{-<k_i>}, \tag{6}$$

$$< k_i > = N \sum_j p_{ij} l_j \quad (7)$$

$< k_i >$ is the expected degree of a node in the ith interval, and $l_i = d_{i+1} - d_i$ is the length of the ith interval.

We can define the energy of a link probability measure as

$$E = \sum_{k=k_{\min}}^{k_{\max}} \frac{p^*(k) - p(k)}{\max(p^*(k), p(k))} \quad (8)$$

where $p^*(k)$ is the degree distribution of the actual link probability measure, and $p(k)$ is the target degree distribution.

3 Results

3.1 The mfng Program

There is an existing implementation of the multifractal network generator written in C++ without the option of setting target properties [9]. Its source code is unfortunately not available. Our earlier works are about the *cxnet* framework (a Python package) we developed to investigate complex networks and bring them into higher education [6-7]. The *mfng* generator does not need the *cxnet* framework, but the analysis of the result needs it. During generation, the program does not create networks. It calculates the expected values of the degree distribution from equation (4) (see below). The *analyser* module of the *mfng* software package provides three main features:

1) It can generate networks from the generating measure constructed by the mfng generator.
2) It can calculate the degree distributions of these networks.
3) It can plot the degree distributions of these networks as well as the degree distribution calculated from equation (4). It can use several binning methods to create clearer plots. Figure 3 (a) is an example plot created using the analyser module.

Earlier the *mfng* generator and *analyser* was a sub-module of the *cxnet* package. To make the installation of the *mfng* easier it has become a standalone package. The documentation of cxnet with the installation of the *mfng* package and a tutorial can be reached from the page [12]. The *mfng* program can be reached from its repository [13] using the git version control system, or can be downloaded as zip or gzipped tar archive from there.

The mfng module includes the *ProbMeasure* class, the *Generator* class and some property classes. An instance of the *ProbMeasure* class contains the probabilities and the division points. It includes a function to iterate the measure, returning with a new *ProbMeasure* instance. This function makes it possible to create the link probability measure from the generating measure using the numpy module. The *mfng* program generates the generating measure for the networks with the given properties. There are two steps with the same temperature T. In one step, the generator changes the probabilities; in the second step it changes the division points. The *Generator* class stores the main parameters of the generation and the property we want to achieve. The main parameters of the generation are the initial and final temperature, the number of steps, the number of intervals in the generating measure and the K number of iteration.

3.2 Changing the Division Points and the Probabilities

In two alternating steps, the program first changes the probabilities and then changes one of the division points. Changing the probabilities is performed in three steps. First, the program chooses one of the elements of the probability matrix randomly. In the second step, it multiplies the probability with a random value from a uniform distribution on the [0.9, 1.1[interval, so the probability will not change more than 10%. For the element not in the main diagonal, the symmetric element needs to be multiplied as well. In the third step, the probability matrix is normalized.

To change the division points, the program adds zero and one to the list of the division points, so the division points will be $d_0, d_1, d_2, \ldots, d_{m-1}, d_m$, where $d_0=0$ and $d_m=1$.

Then the program chooses randomly one of the inner division points with the index i∈[1,m-1] and chooses a p random value from the uniform distribution on the [0,1[interval. The program relocates the chosen division point to $d_i + \Delta_{\mathrm{i}}(p)$, where

$$\Delta_{\mathrm{i}}(p) = \begin{cases} l\frac{\left(p-\frac{a}{l}\right)^n}{\left(1-\frac{a}{l}\right)^{n-1}}, \text{if } p > \frac{a}{l} \\ l\frac{\left(p-\frac{a}{l}\right)^n}{\left(-\frac{a}{l}\right)^{n-1}}, \text{otherwise} \end{cases} \tag{9}$$

Here, $= d_{i+1} - d_{i-1}$, and $a = d_i - d_{i-1}$ and $n > 1$ is an arbitrary exponent.

If the $\Delta_{\mathrm{i}}(p)$ function gives 0, the division point stays in the original place. With the increasing n exponent parameter, the $\Delta_{\mathrm{i}}(p)$ function is more likely to be close to zero, so the new division point is more likely to stay in the proximity of the original division point (Figure 2).

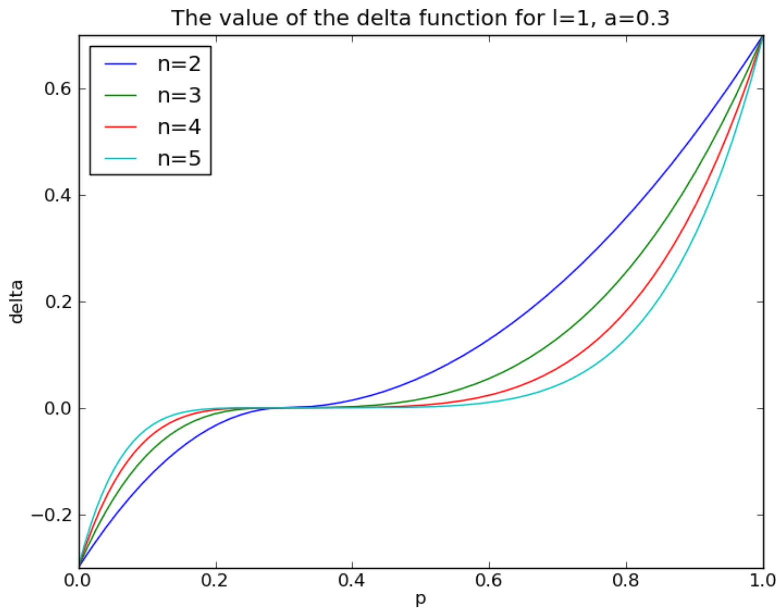


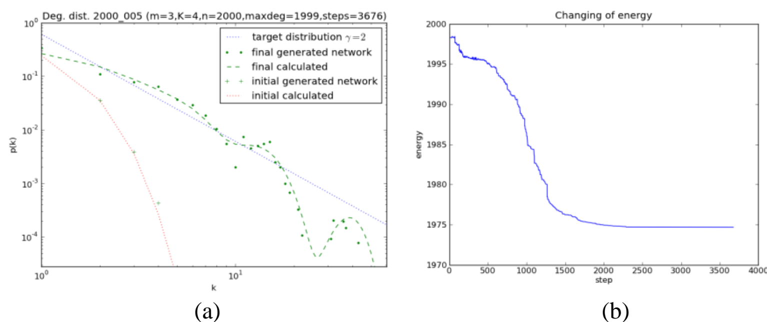
Figure 2

An example for the $\Delta_i(p)$ function used for relocating a division point. This example uses one inner division point ($m = 2$) with the actual value 0.3, so the parameters are $l = 1$ and $a = 0.3$. The function was plotted with these parameters and with the exponents $n = 2,3,4,5$. The value of the function will be in the interval $[-a, l - a[= [-0.3, 0.7[$.

3.3 Generating a Network with Given Degree Distribution

Our program calculates the degree distribution as in equation (5), and in our measurements, it used the energy function in the equation (8).

The two property classes, *DegreeDistribution* and *DegreeDistributionC,* can calculate the degree distribution of a generating measure and can return with the energy of that distribution. In the first one, the calculation of the degree distribution has been written in Python using the *numpy* package. The second one uses C++ functions for that calculation. Each version uses the same Python function to calculate the energy from the degree distribution.



(a) (b)

Figure 3

Results of a generation. The target was a power-law function degree distribution with the exponent −2. In subfigure (a) the target degree distribution was drawn with a blue dotted line, the initial degree distribution calculated from the generating measure with a red dotted line, and the degree distribution calculated from the last accepted generating measure with a green dashed line. The degree distributions of networks generated from the initial and from the last accepted one are drawn with dots. In the subfigure (b) the energy as the function of the step number can be seen.

3.4 The Advantages of Python Programming Language

There are several reasons to use the Python language for the main program. Python itself is a dynamically-typed, object-oriented language with some useful complex data types (list, dictionary, set). These data types and the dynamically-typed property make possible a very flexible argument handling of functions with default argument values and keyword arguments. We frequently use two Python shells (*ipython* and *IDLE*) to run Python commands interactively. *IDLE* is part of most installations, but *ipython* has several useful extra abilities, like the interactive plotting of the functions with the *pylab* package.

The Python language has a huge standard library that can be reached in the standard installations on many operating systems, including Windows, Linux and MacOSX. We used the *shelf* package to store the generated results as well as the energies, the divisions and the probabilities of each step in a binary form.

Another advantage is the many useful free and high quality packages not included in the standard library. One of them is the numpy package that has its own data structures like *array* and *matrix*. An *array* is a sequence of elements of the same type. *Numpy* has mathematical functions like logarithm that can calculate the logarithm of each element of the array or matrix in one step. This calculation is quite fast, because the functions of numpy are written in the C programming language. The calculations can be slow if the calculations have too many steps at the Python level. For example, if we have more additions, subtractions, multiplications, divisions and functions calls, the Python must check whether the factors, the terms or arguments are arrays or not. These steps slow down the calculations.

The other useful package is the *pylab* package, which provides mathematical and plotting functions very similar to that of in MATLAB. This package is based on the *numpy* and *matplotlib* packages. The *pylab*, *numpy* and *matplotlib* packages are not part of the standard library.

To analyse the properties of the network belonging to the generating measure, the program uses the *igraph* complex network analyser package.

A big part of the code is covered with unit tests, which allows us to check easily the functioning our program in several environments (Python versions and operating systems), as the *unittest* package is also part of the standard library.

To identify the most CPU intensive parts of the program the *cProfile* module of the standard library can be used.

3.5 Numpy Version of the Program

The first version of the program used the numpy package of the Python programming language. With this package we can use arrays (row vectors), which can be manipulated more efficiently than Python lists. We used the cProfiler module to determine the parts of the program that needs too much time. We found that the iteration and the calculation of the estimated degree distribution belonging to the link probability measure were two such parts.

We ran the generation with 2000 steps and 2000 nodes. The time of the generation was 5392 seconds. The calculation of the degree distribution from the link probability needed 3625 seconds (67%), and the calculation of the link probability measure took 1748 seconds (32%). The calculation of the energy from the actual degree distribution and the other parts of the program took less than 1% of the running time.

3.6 The C++ Version of the Program

According to the running time measurements, the iteration of the generating measure and the calculation of the degree distribution were rewritten. The C++ program gets the generating measure from Python and writes the degree distribution to the standard output, where the Python collects information. In the future we plan to implement a more appropriate coupling between the C++ and the Python part of the program. We will wrap the C++ code with SWIG or CPython [2] to call the C++ functions easier.

3.7 The Comparison of the Running Times

We carried out a sequence of generations to compare the running time of the pure Python version using the numpy module and the version using C++ code. The program ran on a Debian Linux server installed as a VMware virtual machine.

The target degree distribution of the generations was a power-law degree distribution with the exponent -2. There were three parameters that changed: the number of steps, the number of iterations and the number of nodes in the network. More details about the network generator program can be found in the Appendix. The resultant running times are in Table 1 and are plotted on Figures 4 and 5.

Table 1

The running times of the two versions and ratio of the numpy version and the C++ version. The full running times are in minutes, but the running times of one step are in milliseconds.

		2000 steps						5000 steps					
		K=4		K=5		K=6		K=4		K=5		K=6	
# nodes	**type**	full	one step	full	one step	full	one step	full	one step	full	one step	full	one step
2000	numpy	23,37	701,01	89,88	2696,40	503,47	15104,10	58,35	700,18	224,40	2692,80	1259,86	15118,32
	C++	1,68	50,52	4,44	133,20	13,59	407,64	4,28	51,42	11,10	133,20	33,52	402,19
	ratio	**13,9**		**20,2**		**37,1**		**13,6**		**20,2**		**37,6**	
5000	numpy	53,62	1608,51	175,50	5265,00	759,10	22773,00	133,81	1605,73	439,85	5278,20	1905,34	22864,08
	C++	3,79	113,80	9,75	292,50	27,78	833,49	9,70	116,44	24,20	290,40	69,59	835,08
	ratio	**14,1**		**18,0**		**27,3**		**13,8**		**18,2**		**27,4**	
10000	numpy	102,72	3081,60	319,47	9584,10			257,36	3088,28	801,56	9618,72		
	C++	7,12	213,56	17,60	528,00			16,96	203,54	43,83	525,96		
	ratio	**14,4**		**18,2**				**15,2**		**18,3**			

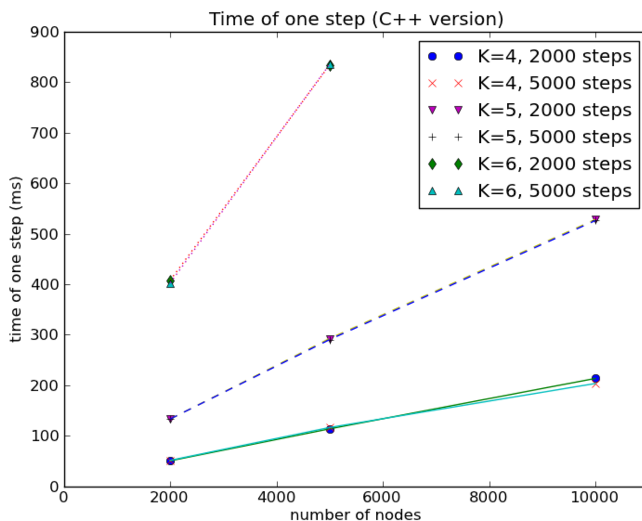


Figure 4

Running time of the C++ version of the generator as a function of number of nodes with a scale-free degree distribution as the target property. The number of iteration (K), number of nodes and number of steps have been varied.

This method of network generation makes it impossible for the network to have multiple edges between a node pair or to have self-loops (edges with the same nodes at its two endpoints), so the maximal degree cannot be bigger than the number of nodes in the network. During generation, the maximal degree in equation (8) was set to smaller by one than the number of the nodes, so with an increase in the number of nodes in the generation, the number of the terms in the

sum and the running time increases, too. With an increase in the number of iterations, the running time increases fast, because the number of probabilities in the link probability measure increases exponentially with the number of iteration. For larger values of the number of iterations, the running times of the mfng version became too large (bigger times than one day can be found in Table 1), but with the C++ version this time is acceptable. The running time of mfng version compared to that of the C++ version is 13–38 times longer in these generations, and this ratio increases with the increasing number of iterations.

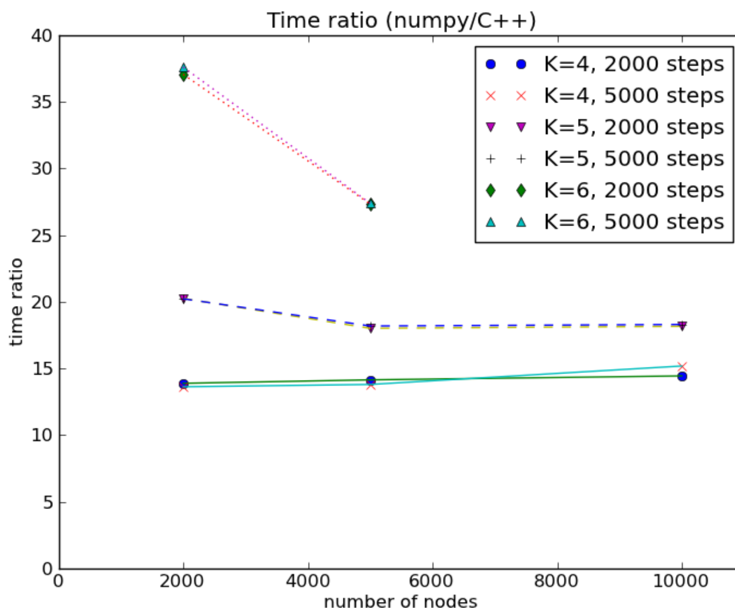


Figure 5

The ratio of the running times of the numpy version and the C++ version.

Conclusions

With the multifractal network generator (MFNG) method one can generate a wide range of networks with prescribed statistical properties. The method uses a mapping between the generator measures (a measure defined on the unit square) and the networks. It simulates an annealing process to get the optimal parameters of the generator measure. If one knows the generator measure, the degree distribution and some other statistical properties of the network can be calculated.

In our *mfng* program there are two realizations of the MFNG method. Our first realization of the MFNG method was written in Python using the *numpy* package. After rewriting some functions of the *mfng* program in C++ language, the running time of the program was reduced significantly, which allows for using a higher iteration number and more steps, so one can create generating measures that generate networks with properties closer to the target property.

Our program is part of the *cxnet* program framework intended to be used in education. As the C++ functions can be reached from Python, the easy-to-use Python framework would not necessarily be dropped. It makes it easier to use the framework in the education, especially if the students are familiar with the *cxnet* framework.

Acknowledgement

I would thank to Péter Orlik for his help in writing the C++ code. I would thank to Marianna Machata, Trócsányi Zoltán and József Lakner for their linguistic help and advice.

Appendix

In the running time comparison the program started the generation with a Python program as follows:

```
import mfng
for steps in [10000, 20000]:
   T0 = 0.2
   generator = mfng.Generator(T0=T0, steps=steps, Tlimit=T0/10000,
         m=3, K=4,
         n=2000,
         divexponent=7,
         project="power_law",
         )
   generator.append_property(
         mfng.DistributionFunction(
            "k**-2",
            maxdeg=n-1, mindeg=1
            )
         )
   generator.go()
```

The meaning of the program is as follows. First the program imports the functions and classes of the mfng module. It carries out two generations creating a generator in both generations. The temperature will decrease from 0.2 to 2×10^{-5} in 10000 and 20000 steps respectively. The generating measure in the generations would have 3×3 probabilities and it would be created for a network with n=2000 nodes. The changing of the division points will use the exponent 7 in equation (9). The result will be saved in the project_power_law directory. There is one target property with a distribution function proportional to the k^{-2} power-law function. The degree distributions in the generation will be compared to the target distribution from the degree 1 to 1999.

This version uses the numpy version to generate the generator function. If we slightly modify this program—we would add DistributionFunctionC (with C in the end) property to the generator instead of DistributionFunction—the generator runs the C++ version.

References

[1] Albert, R., Barabási A.: Statistical Mechanics of Complex Networks, Reviews of Modern Physics, Vol. 74, No. 1, 2002, pp. 47-97, doi:10.1103/RevModPhys.74.47

[2] Behnel, S., Bradshaw, R., Citro, C., Dalacin, L., Seljebotn, D. S., Smith, K.: Cython: The Best of Both Worlds, Computing in Science Engineering, Vol. 13, No. 2, 2011, pp. 31-39

[3] Csárdi, G., Nepusz, T.: The Igraph Software Package for Complex Network Research, InterJournal Complex Systems, 2006, Manuscript Number. 1695

[4] Cardanobile, S., Pernice, V., Deger, M., Rotter S.: Inferring General Relations between Network Characteristics from Specific Network Ensembles. PLoS ONE Vol. 7, Jun 2012, e37911, doi:10.1371/journal.pone.0037911

[5] Horváth, A., Trócsányi, Z.: Multifractal Network Generator with Igraph, Symposium on Applied Informatics and Related Areas, 2010

[6] Horváth, A., Trócsányi, Z.: Complex Networks in the Curriculum of the Computer Engineers, IEEE Proceedings of the 8th International Symposium on Applied Machine Intelligence and Informatics, 2010

[7] Horváth, A.: Studying Complex Networks with cxnet, Acta Physica Debrecina, Vol. XLIV, 2010, pp. 37-47

[8] M. E. J. Newman, The Structure and Function of Complex Networks, SIAM Review, Vol. 45, No. 2, 2003, pp. 167-256

[9] Palla, G., Lovász, L., Vicsek, T.: Multifractal Network Generator Proceeding of the National Academy of Sciences, Vol. 107, No. 17, Apr. 2010, pp. 7640-7645, doi:10.1073/pnas.0912983107

[10] Palla, G., Pollner, P., Vicsek, T.: Rotated Multifractal Network Generator, Journal of Statistical Mechanics: Theory and Experiment, Vol. 2011, No. 02, February 2011, P02003, doi:10.1088/1742-5468/2011/02/P02003

[11] Ravasz, E., Barabási A.: Hieararchical Organization in Complex Networks, Physical Review E, Vol. 67, No. 026112, Feb 2003

[12] The homepage of the cxnet program, http://django.arek.uni-obuda.hu/cxnet

[13] The repository of the mfng program, http://github.com/horvatha/mfng