Report

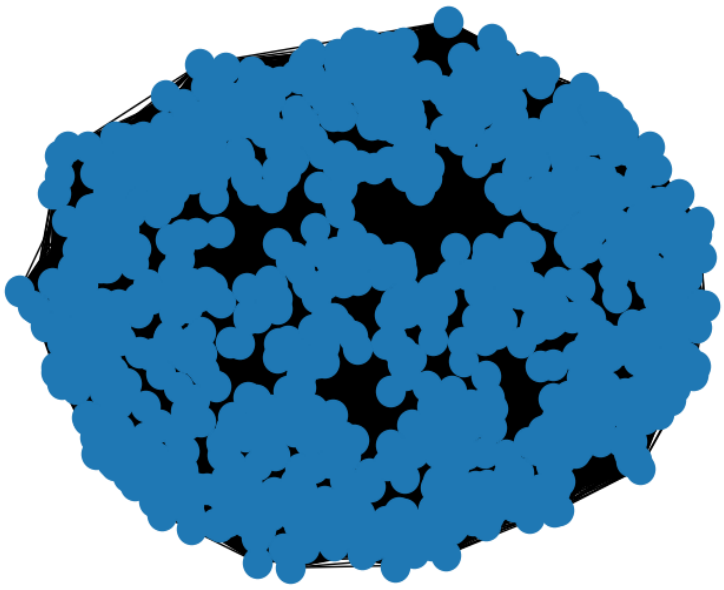
فاطمه توکلی

در این گزارش به بررسی پارمتر نظم در شبکه‌های اردوش رنیی، بی مقیاس و جهان کوچک می‌پردازیم. بدین منظور ابتدا مدل کورامتو به روش رانگ کوتای مرتبه ۴ حل کردیم و سپس پارامتر نظم را بدست آوردیم و برای شبکه‌های فوق اجرا کردیم. (الگوریتم ها در قسمت پیوست می باشند)

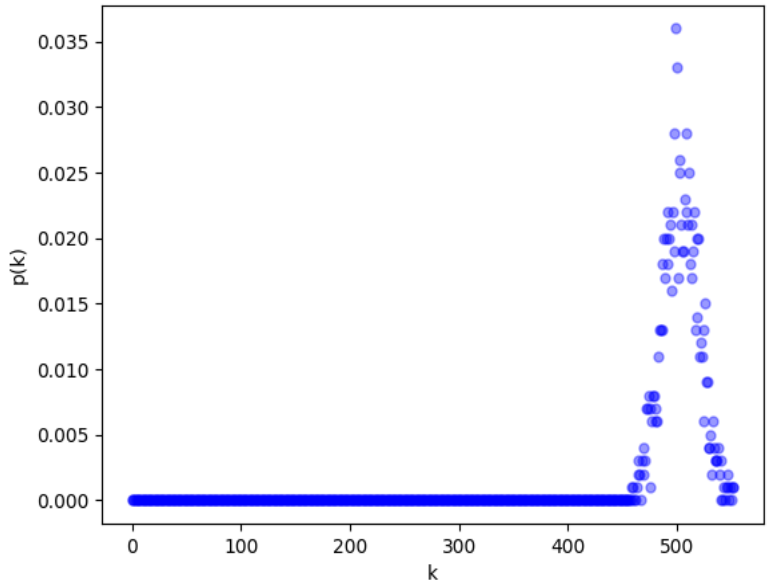
۱- شبکه اردوش رنیی:

در شبکه تصادفی اردوش رنیی ابتدا ۱۰۰۰ گره را با احتمال ثابت ۰.۵ به صورت تصادفی به یکدیگر وصل کردیم، با استفاده از رابطه‌ی ۱ متوسط درجه‌ی این شبکه، ۴۹۹.۵ می‌باشد که در شکل ۲ نیز مشاهده می‌شود.

$$P(N-1) = \langle k \rangle \qquad (۱)$$

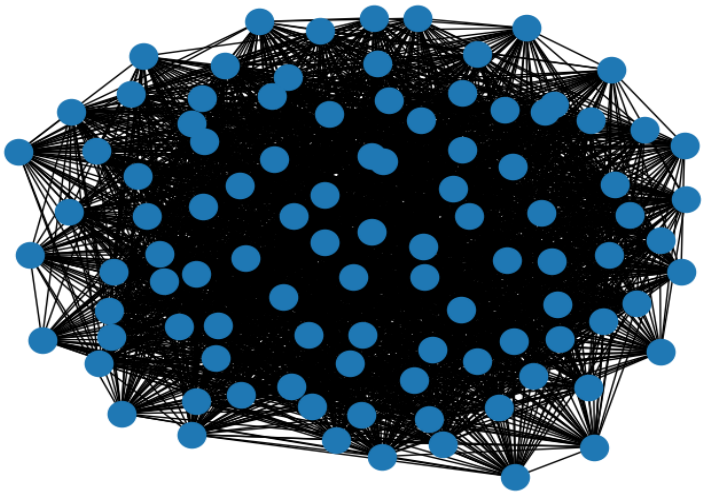


شکل ۱ : شبکه تصادفی اردوش رنی برای ۱۰۰۰ گره با احتمال ۰.۵، متوسط درجه نیز ۴۴۹.۵ می‌باشد.

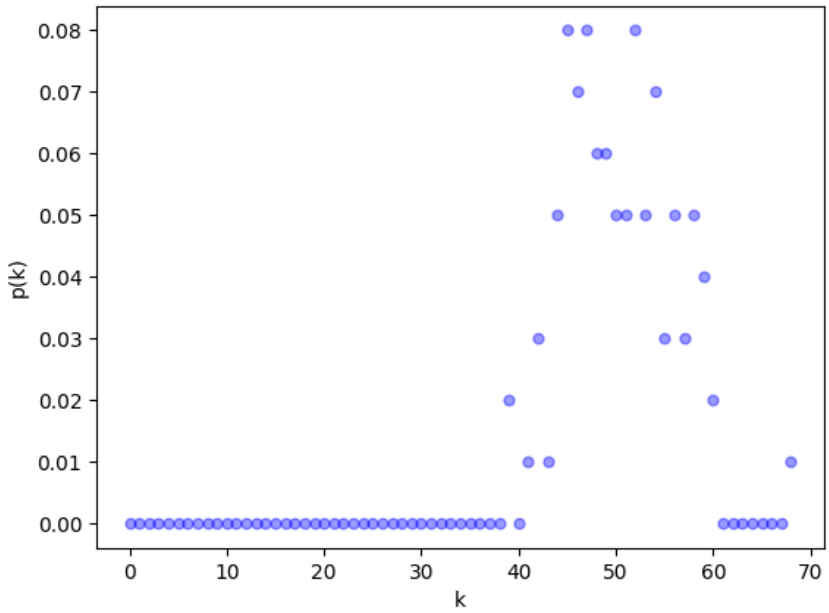


شکل۲: نمودار تابع توزیع برحسب درجه برای ۱۰۰۰ گره با احتمال ۰.۵، متوسط درجه نیز ۴۴۹.۵ می‌باشد.

برای اطمینان بیشتر همین روند را برای ۱۰۰ گره با احتمال ۰.۵ هم انجام دادیم که در شکل‌های ۳و۴ مشاهده می‌شوند.



شکل۳: شبکه تصادفی اردوش رنی برای ۱۰۰ گره با احتمال ۰.۵، متوسط درجه نیز ۴۹.۵ می‌باشد.



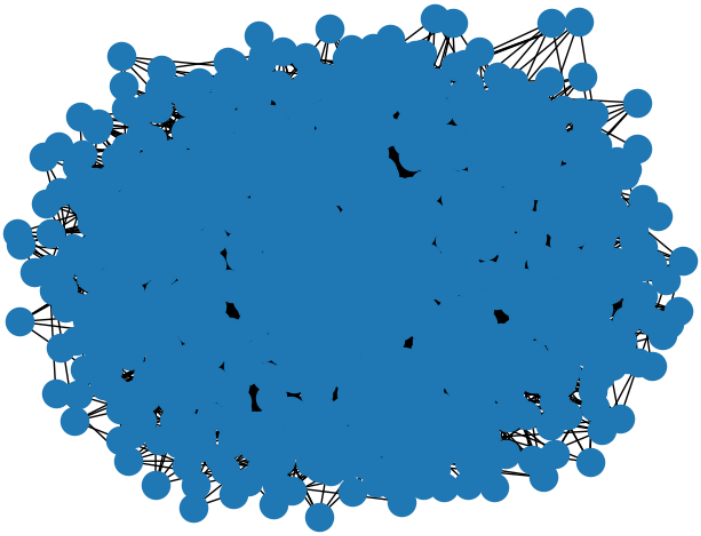
شکل ۴: نمودار تابع توزیع برحسب درجه برای ۱۰۰ گره با احتمال ۰.۵، متوسط درجه نیز ۴۹.۵ می‌باشد.

۲- شبکه بی مقیاس

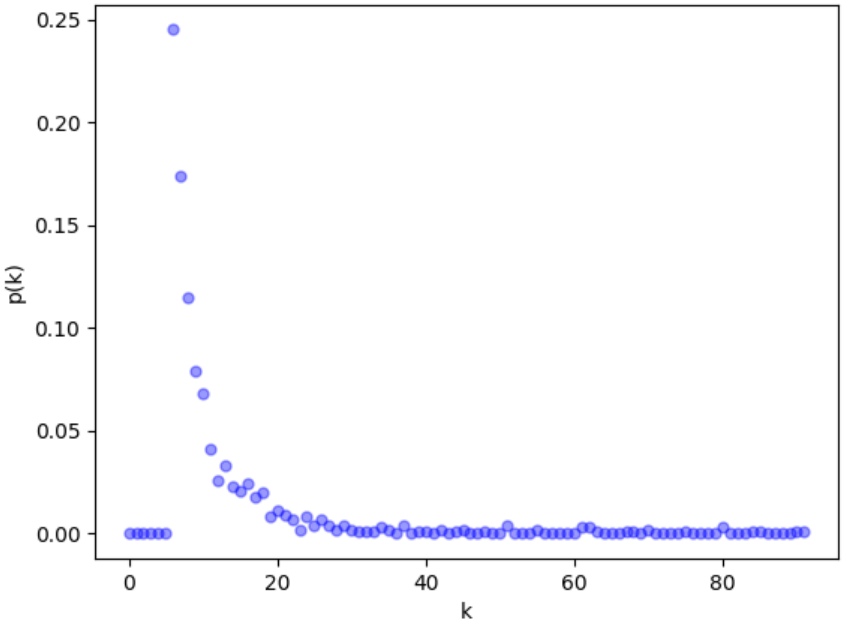
برای تولید شبکه‌ی بی مقیاس از مدل آلبرت باراباسی استفاده کردیم که با ۱۰ تا گره شروع کردیم (m_0) و در هر مرحله‌ی زمانی یک گره جدید با احتمال p_i (احتمال اینکه راس جدید به i امین راس موجود درگراف وصل بشود) به m ($m<m_0$) راس قبلی اضافه می‌کنیم که برای شکل۵، m را ۶ گرفتیم و متوسط درجه با توجه به رابطه‌ی ۳ برابر ۱۱.۹ می باشد. (در شکل ۶ نیز قابل مشاهده می‌باشد)

$$p_i = \frac{K_i}{\sum_j K_j} \quad (۲)$$

$$\langle K \rangle \approx 2m \quad (۳)$$

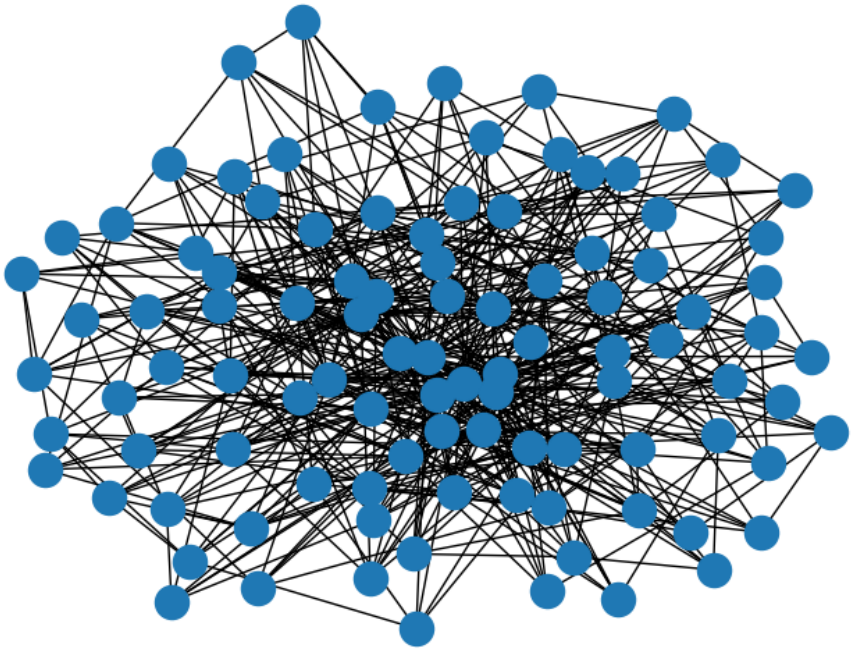


شکل ۵ : شبکه بی مقیاس برای ۱۰۰۰ گره با تعداد گره‌ی اولیه ۱۰، m =6 و متوسط درجه نیز ۱۱.۹ می‌باشد.

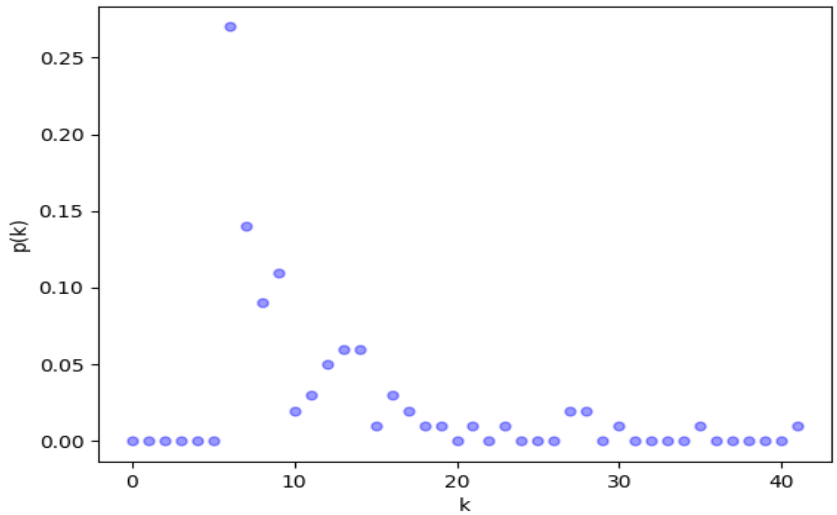


شکل۶: نمودار تابع توزیع برحسب درجه برای ۱۰۰۰ گره با تعداد گره‌ی اولیه ۱۰، m =6 و متوسط درجه نیز ۱۱.۹ می‌باشد.

برای اطمینان بیشتر همین روند را برای ۱۰۰ گره با ۱۰ گره اولیه و m=6 نیز انجام دادیم



شکل ۷ : شبکه بی مقیاس برای ۱۰۰ گره با تعداد گره‌ی اولیه ۱۰، m =6 و متوسط درجه نیز ۱۱ می‌باشد.

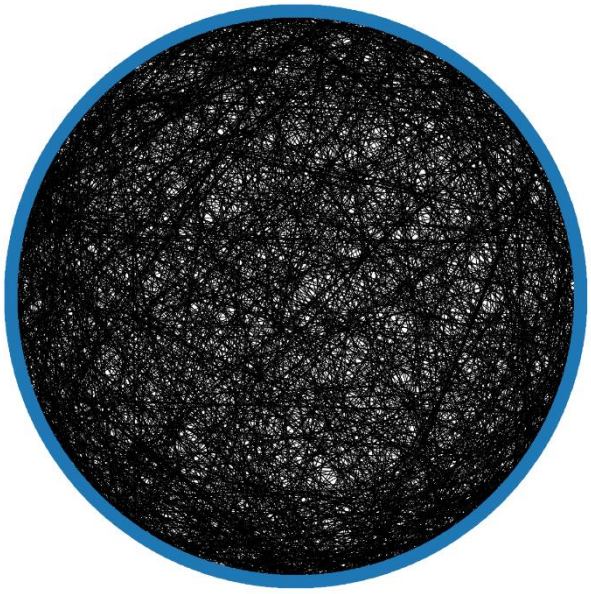


شکل۸: نمودار تابع توزیع برحسب درجه برای ۱۰۰ گره با تعداد گره‌ی اولیه ۱۰، m =6 و متوسط درجه نیز ۱۱ می‌باشد.

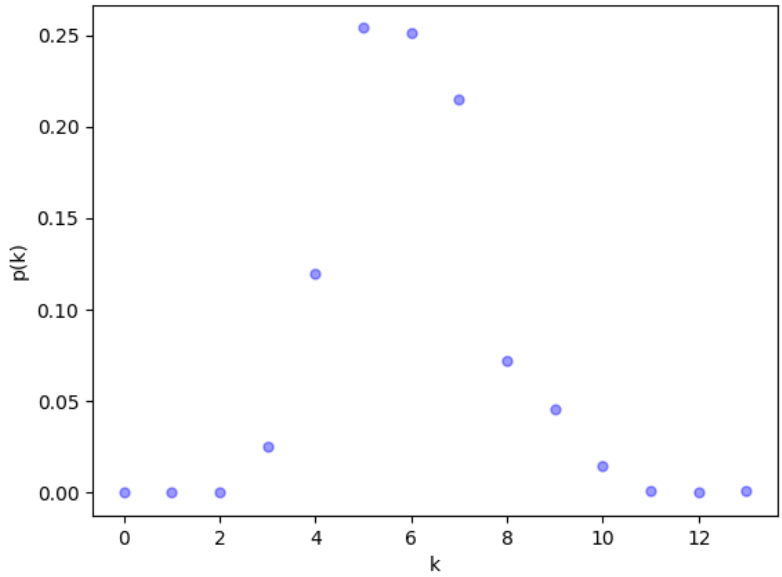
۳- شبکه جهان کوچک

در شبکه جهان کوچک ما حلقه‌ای از ۱۰۰۰ گره داریم که در آن هر گره به $\frac{6}{2}$ نودهای همسایه سمت راست و چپ خود متصل است. لازم به ذکر است که گره‌ها را با احتمال ۰.۵ به هم وصل می‌کنیم همچنین متوسط درجه ۶ می‌باشد.

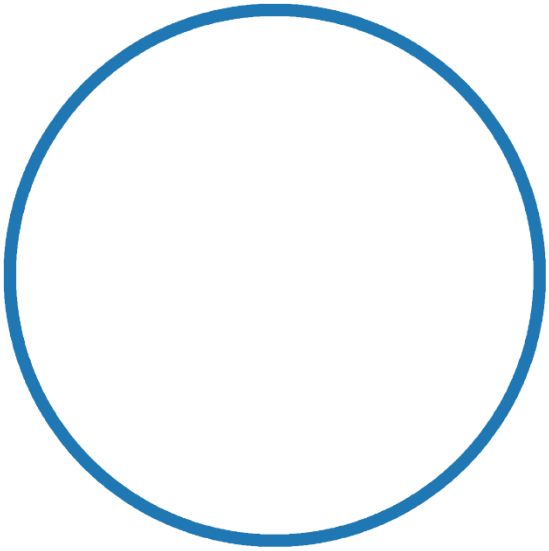
$$\langle K \rangle \approx k \text{ (nearest neighbor network, k is an even number)} \quad (۴)$$



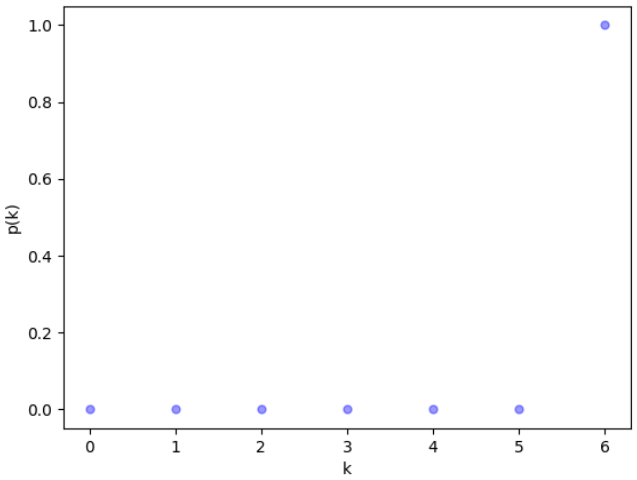
شکل ۹ : شبکه جهان کوچک برای ۱۰۰۰ گره، k= 6 ,p= 0.5 و متوسط درجه نیز ۶ می‌باشد.



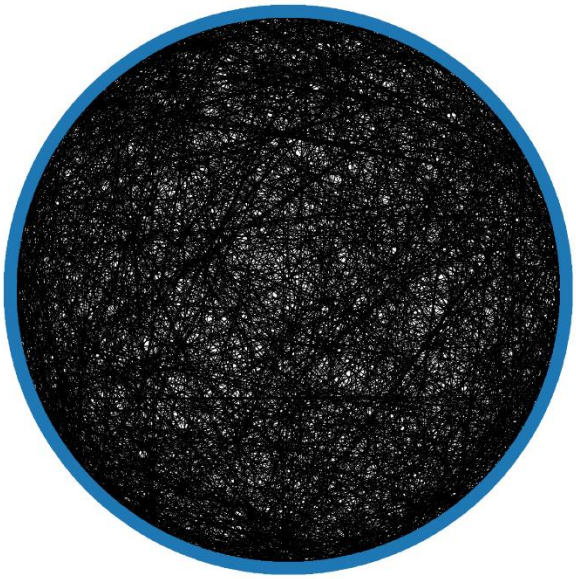
شکل ۱۰ : نمودار تابع توزیع برحسب درجه برای ۱۰۰۰ گره، $k=6$,$p=0.5$ و متوسط درجه نیز ۶ می‌باشد.



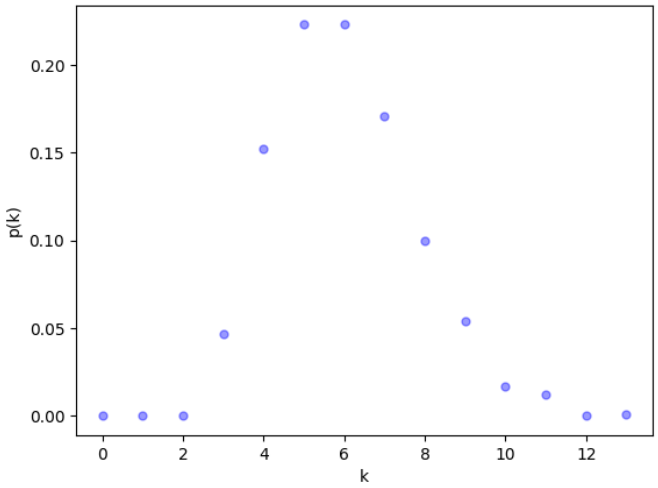
شکل ۱۱ : شبکه جهان کوچک برای ۱۰۰۰ گره، $k=6$,$p=0$ و متوسط درجه نیز ۶ می‌باشد.



شکل ۱۲ : نمودار تابع توزیع برحسب درجه برای ۱۰۰۰ گره، $p = 0, k = 6$ و متوسط درجه نیز ۶ می‌باشد.

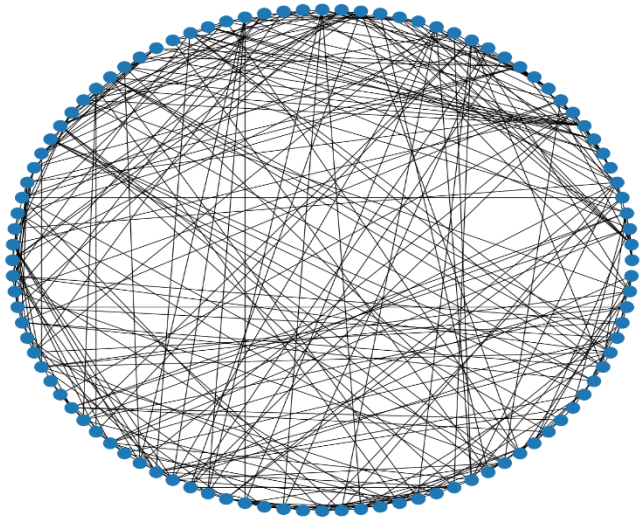


شکل ۱۳ : شبکه جهان کوچک برای ۱۰۰۰ گره، $p = 1, k = 6$ و متوسط درجه نیز ۶ می‌باشد.

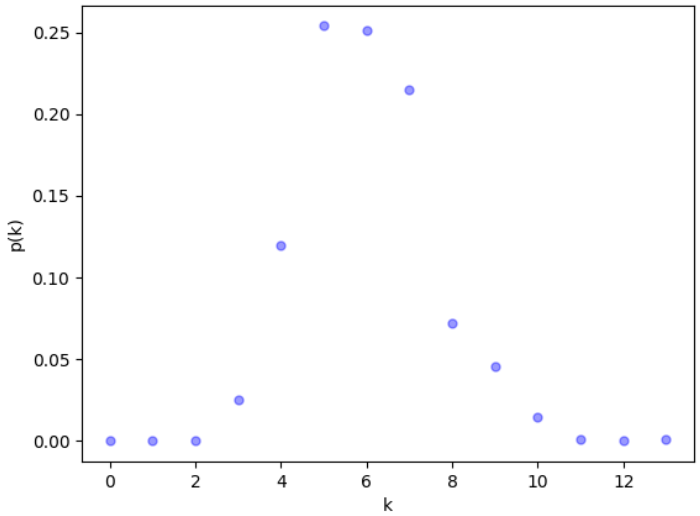


شکل ۱۴ : نمودار تابع توزیع برحسب درجه برای ۱۰۰۰ گره، p =1 ,k =6 و متوسط درجه نیز ۶ می‌باشد.

برای اطمینان بیشتر همین روند را برای ۱۰۰ گره و k=6 با احتمال ۰.۵ نیز انجام دادیم.

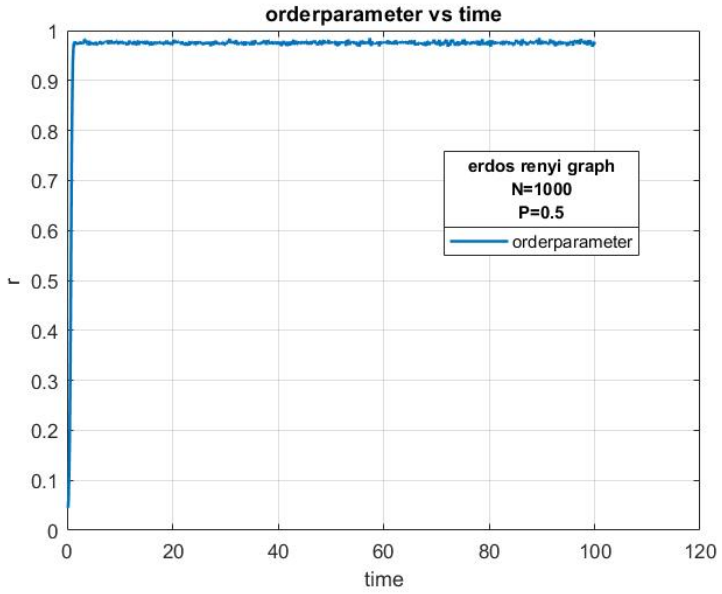


شکل ۱۵ : شبکه جهان کوچک برای ۱۰۰ گره، p =9.5 ,k =6 و متوسط درجه نیز ۶ می‌باشد.

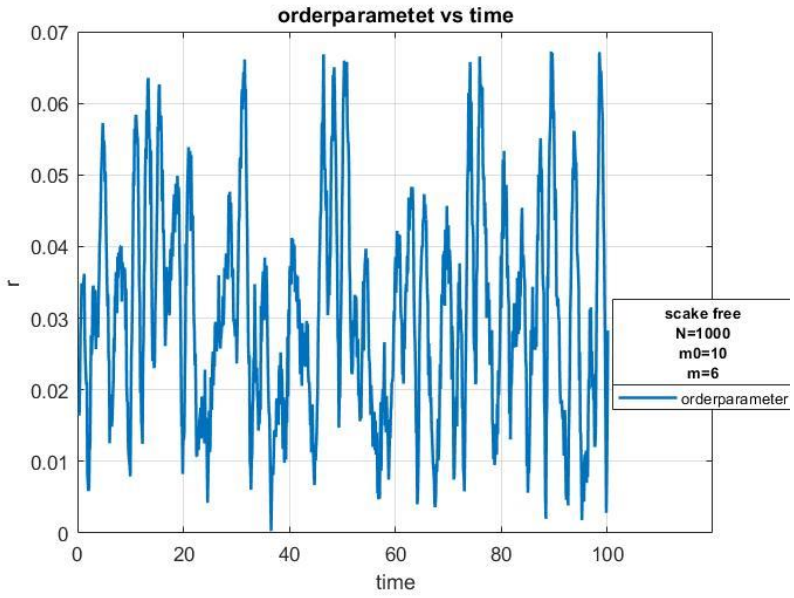


شکل ۱۶ : نمودار تابع توزیع برحسب درجه برای ۱۰۰ گره، $k=6$, $p=0.5$ و متوسط درجه نیز ۶ می‌باشد.

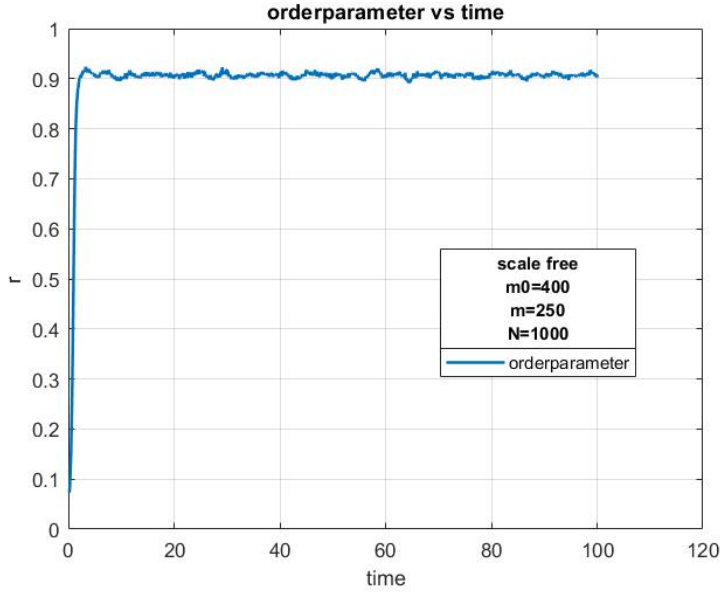
۴-پارامترنظم



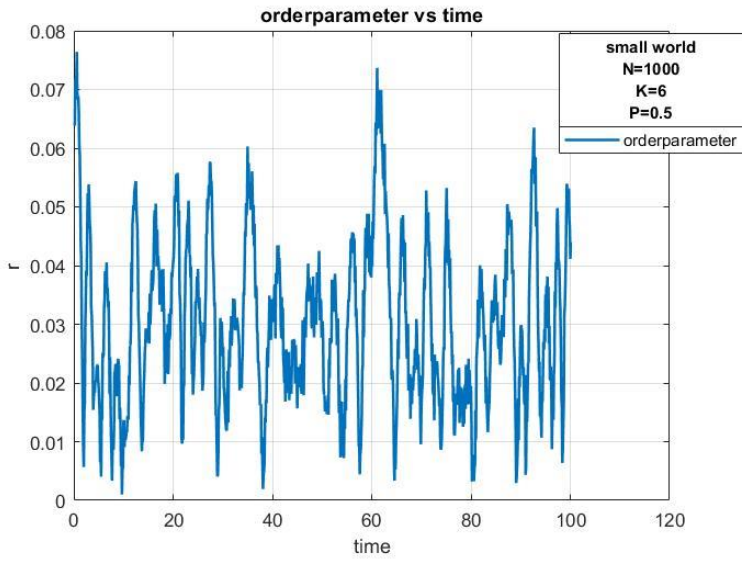
شکل۱۷: نمودار پارامتر نظم برحسب زمان برای شبکه‌ی اردوش رنیی $P=0.5$, $N=1000$



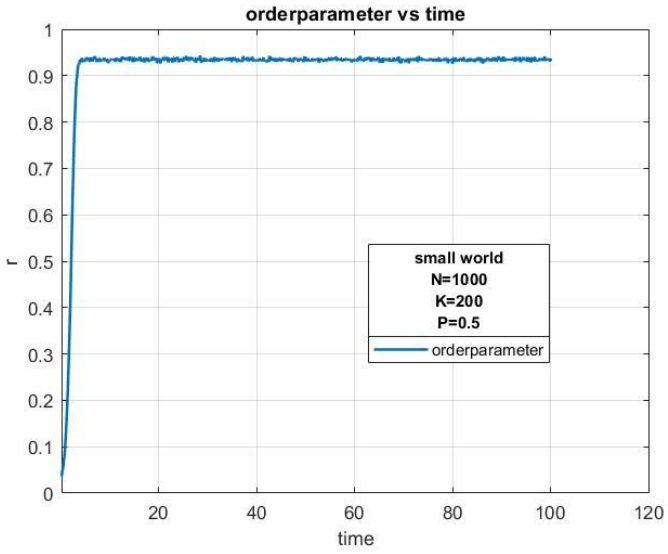
شکل۱۸: نمودار پارامتر نظم برحسب زمان برای شبکه‌ی بی مقیاس m=6 و m0=10 برای ۱۰۰۰ گره.



شکل۱۹: نمودار پارامتر نظم برحسب زمان برای شبکه‌ی بی مقیاس m=250 و m0=400 برای ۱۰۰۰ گره.



شکل۲۰: نمودار پارامتر نظم برحسب زمان برای شبکه‌ی جهان کوچک K=6 و P=0.5 برای ۱۰۰۰ گره.



شکل۲۱: نمودار پارامتر نظم برحسب زمان برای شبکه‌ی جهان کوچک K=200 و P=0.5 برای ۱۰۰۰ گره.

با توجه به شکل ۱۸ و ۱۹ برای شبکه بی مقیاس می‌بینیم با افزایش تعداد m و به تبع آن m_0 (زیرا $m<m_0$ می‌باشد) پارامتر نظم تقریبا برابر ۰.۹ است و در حول این مقدار ثابت می‌باشد (می‌دانیم هرچه پارامتر نظم به یک نزدیک تر باشد سیستم همگام تر و منظم ترمی‌باشد)

و با توجه به شکل ۲۰ و ۲۱ برای شبکه‌ی جهان کوچک مشاهده می‌شود که با افزایش تعداد نزدیک ترین همسایه‌ها (k) پارامتر نظم به یک نزدیک تر می‌شود و تقریبا حول ۰.۹۳ ثابت می‌باشد.

۵-پیوست

کد مربوط به شبکه‌ی اردوش رنیی:

```
/*********************************************************************************************/
/***                                                                                       ***/
/***  Template code for random graph(ER)                                                   ***/
/***                                                                                       ***/
/***  random graph(ER)                                                                     ***/
/***  by FatemeTavakkoli; Student ID#: 951301                                              ***/
/***                                                                                       ***/
/***  The following code is developed in C++ by DevC++                                     ***/
/*** Compile this code in DevC++                                                          ***/
/*** next produce  graphs fromrandomnetwork.txt and degreee.txt  by python                ***/
/*********************************************************************************************/

#include<iostream>
#include <cstdlib>
#include <stdio.h>
#include <iostream>
#include <fstream>
#include <math.h>
#include <ctime>
#include <complex>
using namespace std;
```

```
22.
23.
24. const int   NETWORK_SIZE=1000;
25. const int  nt = 1000;              // The number of time steps
26. const double dt = 0.1;             // time step in runge kutta algorithm
27. double K_copl =20;                 //coupling constant
28.
29. // Variables
30. double probobliyof_the_dge;
31. int** Adjacentmatrix;
32.
33. // Declarations
34. // ============
35. void initial();
36. void generateRandomNetwork();
37. void store_randomnetwork();
38. void calculateDegreeDistribution();
39. void store_degreedistribut(double*);
40. void store_degree(int*);
41. void initW();                      // Initialize
42. void initTeta();                   // init teta() sets the initial values
43. void executeSingleRKStep();        // Ececute single time step according to the Run
    geKutta method
44. void execute();                   // Ececute main part of algorithm
45. float t = 0;
46. float r();                      //  order parameter
47. float w [NETWORK_SIZE];
48. float teta[NETWORK_SIZE];
49. int main(int argc, char *argv[]) {      // Main routine
50.                                   // Rewrite following two lines
51.     clock_t start, finish;            //Used to record the algorithm start time
52. double duration;
53. cout << "Please enter the edge probability of ER network:";
54. cin >> probobliyof_the_dge;
55. start = clock();
56. srand((unsigned)time(NULL));       //used for rand() to generate random numbers
57. initial();
58.
59. generateRandomNetwork();
60. store_randomnetwork();
```

```
61. calculateDegreeDistribution();
62. initW();                    // Initialize
63.   initTeta();                 // init teta() sets the initial values
64. executeSingleRKStep();       // Ececute single time step according to the RungeKu
    tta method
65. execute();
66. finish = clock();
67.
68. duration = (double)(finish - start) / CLOCKS_PER_SEC;
69. system("pause");
70. return 0;
71. }
72.
73. void initial() {
74.
75. if (!(Adjacentmatrix = (int**)malloc(sizeof(int*) * (NETWORK_SIZE + 1))))
76. {
77.  cout << "Adjacency matrix memory allocation error" << endl;
78.      exit(0);
79. }
80.
81. for (int i = 1; i <= NETWORK_SIZE; i++)
82. {
83. if (!(Adjacentmatrix[i] = (int*)malloc(sizeof(int) * (NETWORK_SIZE + 1))))
84. {
85. cout << "adjacentMatrix[" << i << "]Memory allocation error" << endl;
86. exit(0);
87. }
88. }
89. }
90. void generateRandomNetwork()
91. {
92.  int i;
93.  int j;
94.  for ( i =1; i<=NETWORK_SIZE; i++)
95. for (j =i; j <=NETWORK_SIZE; j++)
96.  Adjacentmatrix[i][j] = Adjacentmatrix[j][i] =0;      //initialize the adjacency matrix (
    ER network)
97.          int count =0;                                 //to count the number of undirected e
    dges in the network
```

```
98.         double probability = 0.0;
99. for (int i =1; i<=NETWORK_SIZE; i++)
100.        {
101.        for (int j = i + 1; j <=NETWORK_SIZE; j++)
102.        {
103.         probability = (rand()% NETWORK_SIZE) / (double)NETWORK_SIZE;    //Gen
    erate a random number
104.         if (probability <proboblityof_the_dge)                    //for add an edge bet
    ween pair node
105.        {
106.        count++;
107.        Adjacentmatrix[i][j] = Adjacentmatrix[j][i] = 1;
108.        }
109.        }
110.         }
111.         //Repeat until all node pairs are selected once
112.         cout << "The number of edges in the ER network you construct is:" << co
    unt;
113.        }
114.
115.        void store_randomnetwork()
116.          {
117.           FILE* fout;
118.        if (NULL == (fout = fopen("ernetwork.txt", "w")))
119.        {
120.         cout << "Error opening file randomnetwork.txt!\n" << endl;
121.        exit(0);
122.         }//Create a new randomNetwork.txt file in the code storage path to store t
    he adjacency matrix of the ER network
123.        int i;
124.        int j;
125.        for (i = 1; i <=NETWORK_SIZE; i++)
126.        {
127.        for (j = 1; j <=NETWORK_SIZE; j++)
128.        fprintf(fout, "%d ", Adjacentmatrix[i][j]);
129.         fprintf(fout, "\n");
130.        }
131.        fclose(fout);
132.
133.          }
```

```
134.
135.        void calculateDegreeDistribution()
136.        {
137.            int* degree;
138.        double* statistic;
139.        int i;
140.        int j;
141.        double averageDegree = 0.0;
142.        ofstream degre("degreeER.txt");
143.        if (!(degree = (int*)malloc(sizeof(int) * ( NETWORK_SIZE+ 1))))
144.        {
145.         cout << "degree*malloc error" << endl;
146.        exit(0);
147.        }
148.        for (i = 1; i <= NETWORK_SIZE; i++)degree[i] = 0;
149.        if (!(statistic = (double*)malloc(sizeof(double) * NETWORK_SIZE)))
150.        {
151.         cout << "statistic*malloc error" << endl;
152.        exit(0);
153.        }
154.        for (i = 0; i < NETWORK_SIZE; i++)
155.        statistic[i] = 0.0;
156.        for (i = 1; i <=NETWORK_SIZE; i++)
157.        for (j = 1; j <=NETWORK_SIZE; j++)
158.        degree[i] = degree[i] + Adjacentmatrix[i][j];
159.        for (i = 1; i <= NETWORK_SIZE; i++)
160.        averageDegree += degree[i];
161.         cout << "\t Average Degree<k>=" << averageDegree / (double) NETWO
    RK_SIZE;
162.         for (i = 1; i <= NETWORK_SIZE; i++)
163.          degre<<degree[i]<<endl;
164.        for (i = 1; i <=NETWORK_SIZE; i++)
165.        statistic[degree[i]]++;
166.        double indentify = 0.0;
167.        for (i = 0; i <NETWORK_SIZE; i++)
168.        {
169.        statistic[i] = statistic[i] / (double)NETWORK_SIZE;
170.        indentify += statistic[i];
171.        }
```

```
172.          cout << endl << "If output is 1, the algorithm is correct \toutput=" << in
    dentify << endl;
173.         store_degreedistribut(statistic);
174.
175.         }
176.
177.         void store_degreedistribut(double* statistic)
178.         {
179.
180.         FILE* fwrite;
181.         if (NULL == (fwrite = fopen("ER_degree.txt", "w")))
182.         {
183.          cout << "Error opening file" << endl;
184.         exit(0);
185.         }
186.         int i;
187.         for (i = 0; i < NETWORK_SIZE; i++)
188.          fprintf(fwrite, "%d %f\n", i, statistic[i]);      // statistic[i] is its probability
189.         fclose(fwrite);
190.         }
191.         void initW()
192.         {
193.           int max_n=2;
194.           int min_n=-2;
195.           for (int i =0 ; i<NETWORK_SIZE ; i++)
196.            w[i]=(double)rand()/RAND_MAX * (max_n - min_n) + min_n;  // uniform
    distribution for frequency [-2,2]
197.
198.         }
199.
200.         void initTeta()
201.         {
202.           for(int i=0 ; i<NETWORK_SIZE ; i++)
203.            teta[i]=((rand()*1.0/RAND_MAX)*(acos(-1))*2)-(acos(-
    1)); // between pi and -pi
204.
205.         }
206.         // Function f() which returns the theta dots in the Kuramoto model
207.         void f_RK(const float teta[], float t, float thetaDot[]) {
208.           const float a = K_copl/NETWORK_SIZE;
```

```
209.
210.          for (int i=0; i<NETWORK_SIZE; i++) {
211.            double sum = 0.0;
212.
213.            for(int j=0; j<NETWORK_SIZE; j++)
214.              sum += Adjacentmatrix[i][j] * sin( teta[j] - teta[i] );
215.
216.            thetaDot[i] = w[i] + a * sum;     //Kuramoto equation
217.          }
218.        }
219.
220.
221.        void executeSingleRKStep()
222.         {
223.          static float k1[NETWORK_SIZE];
224.          f_RK(teta,t, k1);
225.
226.          static float temp[NETWORK_SIZE];
227.          for (int i=0 ; i<NETWORK_SIZE ; i++)
228.            temp[i] = teta[i] + 0.5 * dt * k1[i];
229.
230.          static float k2[NETWORK_SIZE];
231.          f_RK(temp, t + 0.5 * dt, k2);
232.
233.          for (int i=0 ; i<NETWORK_SIZE ; i++)
234.            temp[i] = teta[i] + 0.5 * dt * k2[i];
235.
236.          static float k3[NETWORK_SIZE];
237.          f_RK(temp, t + 0.5 * dt, k3);
238.
239.          for (int i=0 ; i<NETWORK_SIZE; i++)
240.            temp[i] = teta[i] + dt * k3[i];
241.
242.          static float  k4[NETWORK_SIZE];
243.          f_RK(temp, t + dt, k4);
244.
245.          const float a = dt / 6;
246.          for (int i=0 ; i<NETWORK_SIZE; i++)
247.            teta[i] += a * (k1[i] + 2.0 *  k2[i] + 2.0 *  k3[i] +  k4[i]);
248.
```

```
249.          t += dt;
250.        }
251.        //order parameter
252.        float r() {
253.          float x = 0.0,     // The real and imaginary parts of order parameter
254.                y = 0.0;
255.
256.          for(int i=0; i<NETWORK_SIZE; i++) {
257.            x += cos(teta[i]);
258.            y += sin(teta[i]);
259.          }
260.
261.          x /= NETWORK_SIZE;
262.          y /= NETWORK_SIZE;
263.
264.          return sqrt( x*x + y*y );
265.        }
266.        // Ececute main part of algorithm
267.        void execute() {
268.          ofstream outputP;
269.          outputP.open("orderparameterER.txt", ios::out|ios::trunc);;
270.          for(int c=0; c<nt; c++)
271.          {
272.            executeSingleRKStep();
273.            outputP << t << '\t' << r() << endl;
274.          }
275.
276.          outputP.close();
277.        }
278.
```

کد مربوط به شبکه‌ی بی مقیاس:

```
1.  /*************************************************************************************
    ******************/
2.  /***                                                                              ***/
```

```
3.  /***  Template code for scalefree graph(SF)                                    ***/
4.  /***                                                                          ***/
5.  /***  scale free(SF)                                                         ***/
6.  /***  by FatemeTavakkoli; Student ID#: 951301                                ***/
7.  /***                                                                          ***/
8.  /***  The following code is developed in C++ by DevC++ IDE                   ***/
9.  /*** Compile this code in DevC++                                             ***/
10. /*****************************************************************************
    ******************/
11.
12. #include <cstdlib>
13. #include <stdio.h>
14. #include <iostream>
15. #include <fstream>
16. #include <math.h>
17. #include <ctime>
18. #include <complex>
19. #include<time.h>
20. #include<string.h>
21. using namespace std;
22.
23. int  M, M_0;
24.
25.
26. struct Node;
27. typedef struct Node* NodePtr;
28. typedef struct Node {
29. int degree;
30. double weight;
31. double probabilityDistribution;
32. }Node;
33. const int   NETWORK_SIZE=1000;      //networksiza
34. const int  nt = 1000;               // The number of time steps
35. const double dt = 0.1;             // time step in runge kutta algorithm
36. double K_copl =20;                 //coupling constant
37. Node* decisionMaking;
38. int** adjacentMatrix;
39. int* initalNetwork;
40. void initial();
41. void initalNetwork_M0_connected();
```

```
42. void updateDecisionMakingData();
43. void generateFreeScaleNetwork();
44. void writeDataToFile();
45. void calculateDegreeDistribution();
46. void write2File_degreedistribut(double*statistic);
47. void initW();                          // Initialize
48. void initTeta();                       // init teta() sets the initial values
49. void executeSingleRKStep();       // Ececute single time step according to the Rung
    eKutta method
50. void execute();                       // Ececute main part of algorithm
51. float t = 0;
52. float r();                          //  order parameter
53. float w [NETWORK_SIZE];
54. float teta[NETWORK_SIZE];
55. int main(int argc, char** argv)
56. {
57.
58.  cout << "****************** You will build an ER random network, please follow th
    e prompts: ************** **" << endl;
59.
60.  cout << "Please enter the total number of BA network initialization network nod
    es:";
61. cin >> M_0;
62.  cout << "Please enter the number of edges between the added node and the pr
    evious network node:";
63. cin >> M;
64.
65. srand((unsigned)time(NULL));
66. initial();
67. initalNetwork_M0_connected();
68. generateFreeScaleNetwork();
69. writeDataToFile();
70. calculateDegreeDistribution();
71. initW();                          // Initialize
72.   initTeta();                     // init teta() sets the initial values
73. executeSingleRKStep();   // Ececute single time step according to the RungeKutta
    method
74. execute();
```

```
75. cout << "You have successfully constructed the BA network, and the adjacency m
    atrix has been successfully written into the file freeScale.txt; the network degree h
    as been written into the file freeScale_degree.txt" << endl;
76. system("pause");
77. return 0;
78. }
79.
80. void initial()
81. {
82. if (!(decisionMaking = (NodePtr)malloc(sizeof(Node) * (NETWORK_SIZE + 1))))
83. {
84. printf("decisionMaking* malloc error\n");
85. exit(0);
86. }
87. if (!(adjacentMatrix = (int**)malloc(sizeof(int*) * (NETWORK_SIZE + 1))))
88. {
89. printf("adjacentMatrix** malloc error\n");
90. exit(0);
91. }
92. int i;
93. for (i = 1; i <= NETWORK_SIZE; i++)
94. {
95. if (!(adjacentMatrix[i] = (int*)malloc(sizeof(int) * (NETWORK_SIZE + 1))))
96. {
97. printf("adjacentMatrix[%d]* malloc error\n", i);
98. exit(0);
99. }
100.        }
101.        if (!(initalNetwork = (int*)malloc(sizeof(int) * (M_0 + 1))))
102.        {
103.        printf("initalNetwork* malloc error\n");
104.        exit(0);
105.        }
106.        }
107.
108.        //Initialization: randomly select M_0 nodes in NETWORK_SIZE to form a co
    nnected network.
109.
110.        void initalNetwork_M0_connected() {
111.        int i, j, randomFirst, randomSecond;
```

```
112.        for (i = 1; i <= NETWORK_SIZE; i++)
113.        for (j = 1; j <= NETWORK_SIZE; j++)
114.        adjacentMatrix[i][j] = 0;
115.         // Randomly generate M_0 nodes
116.        for (i = 1; i <= M_0; i++)
117.        {
118.        initalNetwork[i] = rand() % NETWORK_SIZE + 1;
119.        for (j = 1; j < i; j++)
120.        if (initalNetwork[i] == initalNetwork[j])
121.        {
122.        i--;
123.        break;
124.        }
125.        }
126.        for (i = 1; i < M_0; i++)
127.        adjacentMatrix[initalNetwork[i]][initalNetwork[i + 1]] = adjacentMatrix[inita
    lNetwork[i + 1]][initalNetwork[i]] = 1;
128.        adjacentMatrix[initalNetwork[M_0]][initalNetwork[1]] = adjacentMatrix[init
    alNetwork[1]][initalNetwork[M_0]] = 1;
129.        updateDecisionMakingData();
130.        }
131.
132.
133.        //Update the decisionMaking array through advancedMatrix
134.
135.        void updateDecisionMakingData() {
136.        int i, j, totalDegree = 0;
137.
138.        for (i = 1; i <= NETWORK_SIZE; i++)
139.        decisionMaking[i].degree = 0;
140.        for (i = 1; i <= NETWORK_SIZE; i++)
141.        for (j = 1; j <= NETWORK_SIZE; j++)
142.        decisionMaking[i].degree += adjacentMatrix[i][j];
143.        for (i = 1; i <= NETWORK_SIZE; i++)
144.        totalDegree += decisionMaking[i].degree;
145.        for (i = 1; i <= NETWORK_SIZE; i++)
146.        decisionMaking[i].weight = decisionMaking[i].degree / (double)totalDegre
    e;
147.        decisionMaking[1].probabilityDistribution = decisionMaking[1].weight;
148.        for (i = 2; i <= NETWORK_SIZE; i++)
```

```
149.        decisionMaking[i].probabilityDistribution = decisionMaking[i - 1].probabilit
    yDistribution + decisionMaking[i].weight;
150.        }
151.
152.
153.        //Construct BA scale-free network model
154.        void generateFreeScaleNetwork() {
155.        int i, k, j = 1, length = 0;
156.        int *const random_auxiliary_old = (int*)malloc(sizeof(int)*(NETWORK_SIZE
    + 1));
157.        int *const random_auxiliary = (int*)malloc(sizeof(int)*(NETWORK_SIZE + 1 -
     M_0));
158.
159.
160.        //To ensure that a <new> node is introduced every time, so randomly sele
    ct non-duplicate nodes to join, and delete M_0 nodes in the initial network first
161.
162.        for (i = 1; i <= NETWORK_SIZE; i++)
163.        random_auxiliary_old[i] = i;
164.
165.        for (i = 1; i <= M_0; i++)
166.        random_auxiliary_old[initalNetwork[i]] = 0;
167.        for (i = 1; i <= NETWORK_SIZE; i++)
168.        if (random_auxiliary_old[i] != 0)
169.        random_auxiliary[j++] = random_auxiliary_old[i];
170.
171.        /* Add new nodes to construct a scale-free network*/
172.        int new_node_index, new_node_value;
173.        double random_decision = 0.0;
174.         int targetNode; //Indicates the node to be connected that has been found
     in the network
175.        length = NETWORK_SIZE - M_0;
176.        int flag;
177.        for (i = 1; i <= NETWORK_SIZE - M_0; i++)
178.        {
179.        new_node_index = rand() % length + 1;
180.        new_node_value = random_auxiliary[new_node_index];
181.        random_auxiliary[new_node_index] = random_auxiliary[length--];
182.         for (j = 1; j <= M; j++) //Connect to the M nodes in the existing network a
    ccording to the probability. It is not possible to re-edge or self-connect.
```

```
183.        {
184.        flag = 0;
185.        random_decision = (rand() % 1000) / (double)1000;
186.        for (k = 1; k <= NETWORK_SIZE; k++)
187.        {
188.
189.        if (decisionMaking[k].probabilityDistribution >= random_decision && deci
    sionMaking[k].degree != 0 && adjacentMatrix[new_node_value][k] != 1)
190.        {
191.
192.        targetNode = k;
193.        flag = 1;
194.        break;
195.        }
196.        }
197.        if (flag == 0)
198.        {
199.        for (k = 1; k <= NETWORK_SIZE; k++)
200.        {
201.        if (decisionMaking[k].degree != 0 && adjacentMatrix[new_node_value][k] !
    = 1)
202.        {
203.        targetNode = k;
204.        break;
205.        }
206.        }
207.        }
208.        //printf(" target node is %d\n", targetNode);
209.        adjacentMatrix[new_node_value][targetNode] = adjacentMatrix[targetNod
    e][new_node_value] = 1;
210.        }
211.         updateDecisionMakingData(); //else the newly selected joining node and t
    he M edges in the existing network are linked before updating
212.        }
213.        }
214.
215.
216.
217.
218.        void calculateDegreeDistribution()
```

```
219.          //Calculate the degree distribution, and call the write2File_degreedistribu
    t function to write the degree distribution into the file freeScale_degree.txt
220.        {
221.        int* degree;
222.        double* statistic;
223.        int i, j;
224.        double averageDegree = 0.0;
225.        ofstream degre("degreeSF.txt");
226.        if (!(degree = (int*)malloc(sizeof(int) * (NETWORK_SIZE + 1))))
227.        {
228.         cout << "degree*malloc error" << endl;
229.        exit(0);
230.        }
231.        for (i = 1; i <= NETWORK_SIZE; i++)degree[i] = 0;
232.        if (!(statistic = (double*)malloc(sizeof(double) * NETWORK_SIZE)))
233.        {
234.         cout << "statistic*malloc error" << endl;
235.        exit(0);
236.        }
237.        for (i = 0; i < NETWORK_SIZE; i++)statistic[i] = 0.0;
238.        for (i = 1; i <= NETWORK_SIZE; i++)
239.        for (j = 1; j <= NETWORK_SIZE; j++)
240.        degree[i] = degree[i] + adjacentMatrix[i][j];
241.        for (i = 1; i <= NETWORK_SIZE; i++)
242.        averageDegree += degree[i];
243.          cout << "\t Average Degree<k>=" << averageDegree / (double)NETWO
    RK_SIZE;
244.        for (i = 1; i <= NETWORK_SIZE; i++)
245.          degre<<degree[i]<<endl;
246.        for (i = 1; i <= NETWORK_SIZE; i++)
247.        statistic[degree[i]]++;
248.        double indentify = 0.0;
249.        for (i = 0; i < NETWORK_SIZE; i++)
250.        {
251.        statistic[i] = statistic[i] / (double)NETWORK_SIZE;
252.        indentify += statistic[i];
253.        }
254.         cout << endl << "If output is 1, the algorithm is correct \toutput=" << in
    dentify << endl;
255.        write2File_degreedistribut(statistic);
```

```
256.        }
257.
258.        void write2File_degreedistribut(double* statistic)
259.        {
260.        FILE* fwrite;
261.        if (NULL == (fwrite = fopen("freeScale_degree.txt", "w")))
262.        {
263.         cout << "Error opening file" << endl;
264.        exit(0);
265.        }
266.        int i;
267.        for (i = 0; i < NETWORK_SIZE; i++)
268.         fprintf(fwrite, "%d %f\n", i, statistic[i]);// statistic[i] is its probability
269.        fclose(fwrite);
270.        }
271.
272.         void writeDataToFile()//Write the adjacency matrix of the BA network to t
    he file freeScale.txt
273.         {
274.        FILE* fout;
275.        if (NULL == (fout = fopen("freeScale.txt", "w")))
276.        {
277.        //printf("open file error!\n");
278.                        cout << "Error opening file" << endl;
279.        exit(0);
280.        }
281.        int i;
282.        int j;
283.        for (i = 1; i <= NETWORK_SIZE; i++)
284.        {
285.        for (j = 1; j <= NETWORK_SIZE; j++)
286.        fprintf(fout, "%d ", adjacentMatrix[i][j]);
287.        fprintf(fout, "\n");
288.        }
289.        fclose(fout);
290.        }
291.        void initW()
292.        {
293.          int max_n=2;
294.          int min_n=-2;
```

```
295.          for (int i =0 ; i<NETWORK_SIZE ; i++)
296.           w[i]=(double)rand()/RAND_MAX * (max_n - min_n) + min_n;  // uniform
    distribution for frequency [-2,2]
297.
298.        }
299.
300.        void initTeta()
301.        {
302.          for(int i=0 ; i<NETWORK_SIZE ; i++)
303.           teta[i]=((rand()*1.0/RAND_MAX)*(acos(-1))*2)-(acos(-
    1)); // between pi and -pi
304.
305.        }
306.        // Function f() which returns the theta dots in the Kuramoto model
307.        void f_RK(const float teta[], float t, float thetaDot[]) {
308.          const float a = K_copl/NETWORK_SIZE;
309.
310.          for (int i=0; i<NETWORK_SIZE; i++) {
311.            double sum = 0.0;
312.
313.            for(int j=0; j<NETWORK_SIZE; j++)
314.              sum += adjacentMatrix[i][j] * sin( teta[j] - teta[i] );
315.
316.            thetaDot[i] = w[i] + a * sum;     //Kuramoto equation
317.          }
318.        }
319.
320.
321.        void executeSingleRKStep()
322.         {
323.          static float k1[NETWORK_SIZE];
324.          f_RK(teta,t, k1);
325.
326.          static float temp[NETWORK_SIZE];
327.          for (int i=0 ; i<NETWORK_SIZE ; i++)
328.            temp[i] = teta[i] + 0.5 * dt * k1[i];
329.
330.          static float k2[NETWORK_SIZE];
331.          f_RK(temp, t + 0.5 * dt, k2);
332.
```

```
333.        for (int i=0 ; i<NETWORK_SIZE ; i++)
334.          temp[i] = teta[i] + 0.5 * dt * k2[i];
335.
336.        static float k3[NETWORK_SIZE];
337.        f_RK(temp, t + 0.5 * dt, k3);
338.
339.        for (int i=0 ; i<NETWORK_SIZE; i++)
340.          temp[i] = teta[i] + dt * k3[i];
341.
342.        static float  k4[NETWORK_SIZE];
343.        f_RK(temp, t + dt, k4);
344.
345.        const float a = dt / 6;
346.        for (int i=0 ; i<NETWORK_SIZE; i++)
347.          teta[i] += a * (k1[i] + 2.0 *  k2[i] + 2.0 *  k3[i] +  k4[i]);
348.
349.        t += dt;
350.      }
351.      //order parameter
352.      float r() {
353.        float x = 0.0,     // The real and imaginary parts of order parameter
354.              y = 0.0;
355.
356.        for(int i=0; i<NETWORK_SIZE; i++) {
357.          x += cos(teta[i]);
358.          y += sin(teta[i]);
359.        }
360.
361.        x /= NETWORK_SIZE;
362.        y /= NETWORK_SIZE;
363.
364.        return sqrt( x*x + y*y );
365.      }
366.      // Ececute main part of algorithm
367.      void execute() {
368.        ofstream outputP;
369.        outputP.open("order parameter.txt", ios::out|ios::trunc);;
370.        for(int c=0; c<nt; c++)
371.        {
372.          executeSingleRKStep();
```

```
373.              outputP << t << '\t' << r() << endl;
374.            }
375.
376.            outputP.close();
377.          }
```

کد مربوط به شبکه‌ی جهان کوچک:

```
1.  /*************************************************************************************
    ******************/
2.  /***                                                                              ***/
3.  /***  Template code for small world graph(SW)                                    ***/
4.  /***                                                                              ***/
5.  /***   small world(SW)                                                            ***/
6.  /***  by FatemeTavakkoli; Student ID#: 951301                                     ***/
7.  /***                                                                              ***/
8.  /***  The following code is developed in C++ by DevC++                           ***/
9.  /*** Compile this code in DevC++                                                 ***/
10. /*************************************************************************************
    ******************/
11. #include <cstdlib>
12. #include <stdio.h>
13. #include <iostream>
14. #include <fstream>
15. #include <math.h>
16. #include <ctime>
17. #include <complex>
18. #include<time.h>
19. #include<string.h>
20. using namespace std;
21.
22.
23. int  K;
24. double P;
```

```
25. int** adjacentMatrix;
26. ////////////////////////////////////////
27. const int   NETWORK_SIZE=1000;
28. const double dt = 0.1;              // time step in runge kutta algorithm
29. double K_copl =20;                  //coupling constant
30. float t = 0;
31. ////////////////////////////////////////
32.
33. void initial();
34. void generate_NearestNeighborCoupledNetwork();
35. void generateSmallWorld();
36. void writeDataToFile();
37. void calculateDegreeDistribution();
38. void write2File_degreedistribut(double*statistic);
39. void initW();                       // Initialize
40. void initTeta();                    // init teta() sets the initial values
41. void executeSingleRKStep();        // Ececute single time step according to the Run
    geKutta method
42. void execute();                    // Ececute main part of algorithm
43. float r();                       //  order parameter
44. float w [NETWORK_SIZE];
45. float teta[NETWORK_SIZE];
46. int main(int argc, char** argv)
47. {
48.
49.  cout << "****************** You will build an small word network, please follow th
    e prompts: ************** **" << endl;
50.  cout << "NEIGHBER:";
51.  cin >>K ;
52.  cout << " probablity:";
53.  cin >>P  ;
54.
55. srand((unsigned)time(NULL));
56.    initial();
57.    generate_NearestNeighborCoupledNetwork();
58.    generateSmallWorld();
59.    writeDataToFile();
60.    calculateDegreeDistribution();
61.    initW();                     // Initialize
62.   initTeta();                  // init teta() sets the initial values
```

```
executeSingleRKStep();        // Ececute single time step according to the RungeKutta method
execute();
system("pause");
return 0;
}

void initial()
{
if( !( adjacentMatrix = (int**)malloc(sizeof(int*) * (NETWORK_SIZE + 1))) )
{
printf("adjacentMatrix** malloc error");
exit(0);
}
int i;
for( i = 1; i <= NETWORK_SIZE; i++ )
{
if( !(adjacentMatrix[i] = (int*)malloc(sizeof(int) * (NETWORK_SIZE + 1))) )
{
printf("adjacentMatrix[i]* malloc error");
exit(0);
}
}
}
/*
 * generate the nearest neighbor coupling network: each node is connected to its left and right K/2 nodes
 * */
void generate_NearestNeighborCoupledNetwork(){
int i;
int j;
for( i = 1; i <= NETWORK_SIZE; i++ )
for( j = 1; j <= NETWORK_SIZE; j++ )
adjacentMatrix[i][j] = 0;
for( i = 1; i <= NETWORK_SIZE; i++ )
for( j = 1; j <= K/2; j++ )
if( i - j >= 1 && i + j <= NETWORK_SIZE )
adjacentMatrix[i][i - j] = adjacentMatrix[i][i + j] = 1;
else if( i - j < 1 )
        adjacentMatrix[i][NETWORK_SIZE + i - j] = adjacentMatrix[i][i + j] = 1;
```

```
101.        else if( i + j > NETWORK_SIZE )
102.        adjacentMatrix[i][i + j - NETWORK_SIZE] = adjacentMatrix[i][i - j] = 1;
103.        //test
104.        /*
105.        for( i = 1; i <= NETWORK_SIZE; i++ )
106.        {
107.        for( j = 1; j <= NETWORK_SIZE; j++ )
108.        {
109.        printf("%d ", adjacentMatrix[i][j]);
110.        }
111.        printf("\n");
112.        }
113.        */
114.        //test END
115.        }
116.
117.        /*
118.         * Construct WS small world model
119.         * */
120.        void generateSmallWorld(){
121.        int i, j;
122.        double isChange = 0.0;
123.        int re_connectRandomNode;
124.        int hasEage[NETWORK_SIZE + 1];
125.        int number_changedEage = 0;
126.        for( i = 1; i <= NETWORK_SIZE; i++ )
127.        {
128.        for( j = 1; j <= K/2; j++ )
129.        {
130.
131.        isChange = (rand()%NETWORK_SIZE)/(double)NETWORK_SIZE;
132.
133.        if( isChange < P )
134.        {
135.        while( 1 )
136.        {
137.        re_connectRandomNode = (rand() % NETWORK_SIZE) + 1;
138.
139.        if( adjacentMatrix[i][re_connectRandomNode] == 0 && re_connectRando
    mNode != i )
```

```
140.        break;
141.        }
142.
143.        if( i + j <= NETWORK_SIZE )
144.        adjacentMatrix[i][i + j] = adjacentMatrix[i + j][i] = 0;
145.        else
146.        adjacentMatrix[i][i + j - NETWORK_SIZE] = adjacentMatrix[i + j - NETWORK
    _SIZE][i] = 0;
147.        adjacentMatrix[i][re_connectRandomNode] = adjacentMatrix[re_connectRa
    ndomNode][i] = 1;
148.        number_changedEage++;
149.        }
150.        else
151.        {
152.        //printf("(%d, %d) no change\n", i, i+j);
153.        }
154.        }
155.        }
156.        //test
157.        /*printf("Small World NetWork\n");
158.        for( i = 1; i <= NETWORK_SIZE; i++ )
159.        {
160.        for( j = 1; j <= NETWORK_SIZE; j++ )
161.        printf("%d", adjacentMatrix[i][j]);
162.        printf("\n");
163.        }*/
164.        printf("the number of changed eage is %d, ratio is %f\n", number_changed
    Eage, (double)number_changedEage/(NETWORK_SIZE * K / 2));
165.        }
166.
167.        void calculateDegreeDistribution()
168.          //Calculate the degree distribution, and call the write2File_degreedistribu
    t function to write the degree distribution into the file freeScale_degree.txt
169.        {
170.        int* degree;
171.        double* statistic;
172.        int i, j;
173.        double averageDegree = 0.0;
174.        ofstream degre("degreeWS.txt");
175.        if (!(degree = (int*)malloc(sizeof(int) * (NETWORK_SIZE + 1))))
```

```
176.        {
177.         cout << "degree*malloc error" << endl;
178.        exit(0);
179.        }
180.        for (i = 1; i <= NETWORK_SIZE; i++)degree[i] = 0;
181.        if (!(statistic = (double*)malloc(sizeof(double) * NETWORK_SIZE)))
182.        {
183.         cout << "statistic*malloc error" << endl;
184.        exit(0);
185.        }
186.        for (i = 0; i < NETWORK_SIZE; i++)statistic[i] = 0.0;
187.        for (i = 1; i <= NETWORK_SIZE; i++)
188.        for (j = 1; j <= NETWORK_SIZE; j++)
189.        degree[i] = degree[i] + adjacentMatrix[i][j];
190.        for (i = 1; i <= NETWORK_SIZE; i++)
191.        averageDegree += degree[i];
192.          cout << "\t Average Degree<k>=" << averageDegree / (double)NETWO
    RK_SIZE;
193.        for (i = 1; i <= NETWORK_SIZE; i++)
194.          degre<<degree[i]<<endl;
195.        for (i = 1; i <= NETWORK_SIZE; i++)
196.        statistic[degree[i]]++;
197.        double indentify = 0.0;
198.        for (i = 0; i < NETWORK_SIZE; i++)
199.        {
200.        statistic[i] = statistic[i] / (double)NETWORK_SIZE;
201.        indentify += statistic[i];
202.        }
203.         cout << endl << "If output is 1, the algorithm is correct \toutput=" << in
    dentify << endl;
204.        write2File_degreedistribut(statistic);
205.        }
206.
207.        void write2File_degreedistribut(double* statistic)
208.        {
209.        FILE* fwrite;
210.        if (NULL == (fwrite = fopen("smallworddegree.txt", "w")))
211.        {
212.         cout << "Error opening file" << endl;
213.        exit(0);
```

```
214.        }
215.        int i;
216.        for (i = 0; i < NETWORK_SIZE; i++)
217.         fprintf(fwrite, "%d %f\n", i, statistic[i]);//i means degree, statistic[i] is its pro
    bability
218.        fclose(fwrite);
219.        }
220.
221.
222.        void writeDataToFile(){
223.        FILE* fout;
224.        int i, j;
225.        if( NULL == (fout = fopen("smallWorldNetwork.txt", "w")))
226.        {
227.        printf("open file(smallWorldNetwork.txt) error!");
228.        exit(0);
229.        }
230.        for( i = 1; i <= NETWORK_SIZE; i++ )
231.        {
232.        for( j = 1; j <= NETWORK_SIZE; j++ )
233.        fprintf(fout, "%d ", adjacentMatrix[i][j]);
234.        fprintf(fout,"\n");
235.        }
236.        fclose(fout);
237.
238.        }
239.        void initW()
240.        {
241.          int max_n=2;
242.          int min_n=-2;
243.          for (int i =0 ; i<NETWORK_SIZE ; i++)
244.           w[i]=(double)rand()/RAND_MAX * (max_n - min_n) + min_n;  // uniform
    distribution for frequency [-2,2]
245.
246.        }
247.
248.        void initTeta()
249.        {
250.          for(int i=0 ; i<NETWORK_SIZE ; i++)
```

```
251.            teta[i]=((rand()*1.0/RAND_MAX)*(acos(-1))*2)-(acos(-
    1)); // between pi and -pi
252.
253.        }
254.        // Function f() which returns the theta dots in the Kuramoto model
255.        void f_RK(const float teta[], float t, float thetaDot[]) {
256.          const float a = K_copl/NETWORK_SIZE;
257.
258.          for (int i=0; i<NETWORK_SIZE; i++) {
259.            double sum = 0.0;
260.
261.            for(int j=0; j<NETWORK_SIZE; j++)
262.              sum += adjacentMatrix[i][j] * sin( teta[j] - teta[i] );
263.
264.            thetaDot[i] = w[i] + a * sum;     //Kuramoto equation
265.          }
266.        }
267.
268.
269.        void executeSingleRKStep()
270.         {
271.          static float k1[NETWORK_SIZE];
272.          f_RK(teta,t, k1);
273.
274.          static float temp[NETWORK_SIZE];
275.          for (int i=0 ; i<NETWORK_SIZE ; i++)
276.            temp[i] = teta[i] + 0.5 * dt * k1[i];
277.
278.          static float k2[NETWORK_SIZE];
279.          f_RK(temp, t + 0.5 * dt, k2);
280.
281.          for (int i=0 ; i<NETWORK_SIZE ; i++)
282.            temp[i] = teta[i] + 0.5 * dt * k2[i];
283.
284.          static float k3[NETWORK_SIZE];
285.          f_RK(temp, t + 0.5 * dt, k3);
286.
287.          for (int i=0 ; i<NETWORK_SIZE; i++)
288.            temp[i] = teta[i] + dt * k3[i];
289.
```

```
290.          static float  k4[NETWORK_SIZE];
291.          f_RK(temp, t + dt, k4);
292.
293.          const float a = dt / 6;
294.          for (int i=0 ; i<NETWORK_SIZE; i++)
295.            teta[i] += a * (k1[i] + 2.0 *  k2[i] + 2.0 *  k3[i] +  k4[i]);
296.
297.          t += dt;
298.        }
299.        //order parameter
300.        float r() {
301.          float x = 0.0,     // The real and imaginary parts of order parameter
302.                y = 0.0;
303.
304.          for(int i=0; i<NETWORK_SIZE; i++) {
305.            x += cos(teta[i]);
306.            y += sin(teta[i]);
307.          }
308.
309.          x /= NETWORK_SIZE;
310.          y /= NETWORK_SIZE;
311.
312.          return sqrt( x*x + y*y );
313.        }
314.        // Ececute main part of algorithm
315.        void execute() {
316.          ofstream outputP;
317.          outputP.open("orderparameterWS.txt", ios::out|ios::trunc);;
318.          for(int i=0; i<NETWORK_SIZE; i++)
319.          {
320.            executeSingleRKStep();
321.            outputP << t << '\t' << r() << endl;
322.          }
323.
324.          outputP.close();
325.        }
326.
```

کد مربوط به رسم شبکه‌های فوق در پایتون:

```
import numpy as np
import networkx as nx                                   #import networkx library
import matplotlib.pyplot as plt                         #import matplotlib library
G=nx.Graph()
adjacentMatrix=np.loadtxt('smallWorldNetwork.txt')      #put matrix in arrey 2d

for i in range(len(adjacentMatrix)):
        for j in range(len(adjacentMatrix)):
            if adjacentMatrix[i][j]==1:
                G.add_edge(i,j)

plt.figure(figsize = (12, 12))
nx.draw_circular(G)
plt.savefig("ws10.png")
plt.show()

degree=nx.degree_histogram(G)
x=range(len(degree))
y=[z/float(sum(degree))for z in degree]
plt.scatter(x, y, color="blue",alpha=0.4,s=25)
plt.xlabel('k')
plt.ylabel("p(k)")
plt.savefig("degreeWS10.png")
plt.show()
```