15.12.2023. Stoch. modelling HW3

① Voter model

$$c(\eta,\eta^i) = \sum_{j\neq i} q(j,i)\big(\eta(i)(1-\eta(j)) + \eta(j)(1-\eta(i))\big), \quad \forall i \in \Lambda$$

a) The process will not be ergodic, because (if Λ is finite and $q(i,j)$ is irreducible) there is not unique stationary distribution, because there are two absorbing states: $\eta \equiv 0$ and $\eta \equiv 1$.
→ any linear combination of these states is a stationary distribution.
If $q(i,j)$ is not irreducible, then not all people are communicating in one big group, but instead there are some small groups (connected components of $\mathbb{Z}$), each of which should be treated as a separate subgraph $\mathbb{N}_j$, $j = 1..$ # components.
So for each component we have $\eta_{(j)} \equiv 0$ and $\eta_{(j)} \equiv 1$, where $\eta_{(j)}$ denotes the coordinates of points in the j-th connected component. So any combination $\eta = \eta_{(1)} \dots \eta_{(j)}$, where $\eta_{(i)} \equiv 0$ or $\eta_{(i)} \equiv 1$ is a stationary distribution (this means that in each group people should move to consensus, and the consensus may be either 0 or 1).

b) $q(j,i) = 1$.

$$N_t := \sum_{i=1}^{L} \eta_t(i)$$

Derive the transition rates $g(n,m)$ for $n,m \in \{0,\dots,L\}$ for the process $(N_t : t \geq 0)$.
We know $c(\eta,\eta^i) = \sum_{\substack{i\in\Lambda \\ j\neq i}} \overbrace{q(j,i)}^{\equiv 1}\big(\eta(i)(1-\eta(j)) + \eta(j)(1-\eta(i))\big)$
And we know the formula for the generator of jump process:

$$\Rightarrow (Lf)(N(\eta)) = \sum_{i\in\Lambda} c(\eta,\eta^i)\big[f(N(\eta^i)) - f(N(\eta))\big] =$$

$$= \sum_{i\in\Lambda}\sum_{j\neq i} \big(\eta(i)(1-\eta(j)) + (1-\eta(i))\eta(j)\big)\big[f(N(\eta^i)) - f(N(\eta))\big] =$$

$$= \sum_{\substack{i\in\Lambda \\ \eta(i)=0}}\sum_{j\neq i} \eta(j)\big[f(N+1) - f(N)\big] + \sum_{\substack{i\in\Lambda \\ \eta(i)=1}}\sum_{j\neq i} (1-\eta(j))\big[f(N-1) - f(N)\big] =$$

$$= \sum_{\substack{i\in\Lambda \\ \eta(i)=0}} N\cdot\big[f(N+1) - f(N)\big] + \sum_{\substack{i\in\Lambda \\ \eta(i)=1}} \Big(\underbrace{\sum_{j\neq i} 1}_{=L-1} - \underbrace{\sum_{j\neq i}\eta(j)}_{=N-1}\Big)\big[f(N-1) - f(N)\big] =$$

$$= \boxed{(L-N)N\big[f(N+1) - f(N)\big] + N(L-N)\big[f(N-1) - f(N)\big]}$$

So both transition rates are symmetric and equal to $(L-N)N$.
And from this we see that only cases when both transition rates are zero – it is when $N=0$ or $N=L$, so all people have opinion 0 or 1.
⇒ there are two absorbing states $\eta \equiv 0$ and $\eta \equiv 1$ and any linear combination of them is a stationary distribution.

$$G = \begin{pmatrix} 0 & \cdots & 0 & & \\ (L-N)N & -2N(L-N) & N(L-N) & 0 & \\ & \ddots & \ddots & \ddots & \\ 0 & & N(L-N) & -2N(L-N) & N(L-N) \\ & & \pi_L = 0 & 0 & 0 \end{pmatrix}$$ — generator.

If we solve $[\pi_L]G = [0]$, $\begin{cases} -2\pi_1 + \pi_2 = 0 \\ \pi_{k-1} - 2\pi_k + \pi_{k+1} = 0, \quad k = 2 \dots N-1 \end{cases}$

we will have $\pi_{L-1} = 0, \ \pi_{L-2} - 2\pi_{L-1} = 0.$

$\Rightarrow \pi_k = C_1 + C_2 k.$

$\pi_1 = 0 \Rightarrow C_1 = 0. \Rightarrow \pi_k = C_2 k;$

$\pi_{L-1} = 0 \Rightarrow \pi_k = 0; \ k = 1 \dots L-1.$

$\Rightarrow$ stationary distributions are any $(a, 0 \dots 0, 1-a)$.

c) Give the state space S and the absorbing states of the process $(N_t; t \geq 0)$ and write master equation $p_t(i) := P(N_t = i)$ $\forall i \in S$.
Give a formula for all stationary distributions.
All stationary distributions we have found looking for π_k: $[\pi_k]G = 0$.
master equation: $\frac{d}{dt}[\pi_t] = [\pi_t]G$

$$\Rightarrow \begin{cases} \pi_0' = N(L-N)\pi_1 \\ \pi_1' = -2N(L-N)\pi_1 + N(L-N)\pi_2 \\ \pi_k' = N(L-N)\pi_{k-1} - 2N(L-N)\pi_k + N(L-N)\pi_{k+1} \\ \pi_{L-1}' = N(N-L)\pi_{L-2} - 2N(L-N)\pi_{L-1} \\ \pi_L' = N(N-L)\pi_{L-1}. \end{cases}$$

d) Use symmetry of rates $g(n,m)$ to argue that EN_t doesn't change in time.

$P(N_{t+1} = N) = \underbrace{P(N_{t+1} = N \mid N_t = N-1)}_{1/2} \cdot P(N_t = N-1) + \underbrace{P(N_{t+1} = N \mid N_t = N+1)}_{1/2} \cdot P(N_t = N+1)$

because going from N to $N-1$ is $N+1$ has equal probabilities,

then $EN_t \rightsquigarrow \frac{1}{2}E(N_t+1) + \frac{1}{2}E(N_t-1) = EN_t \Rightarrow EN_t$ doesn't change in time.

So if $N_0 = \frac{L}{2}$, then $EN_t = EN_0 = \frac{L}{2}$

$\Rightarrow$ if we look for a stationary distribution, it will be of the form $(a, 0 \dots 0, 1-a)$, but EN_t should be equal to $\frac{L}{2}$

$\Rightarrow (1-a) \cdot L = \frac{L}{2} \Rightarrow 1-a = \frac{1}{2} \Rightarrow a = \frac{1}{2}$

$\Rightarrow$ stationary distribution is $(\frac{1}{2}, 0 \dots 0, \frac{1}{2})$.

2 Erdos-Renyi graphs

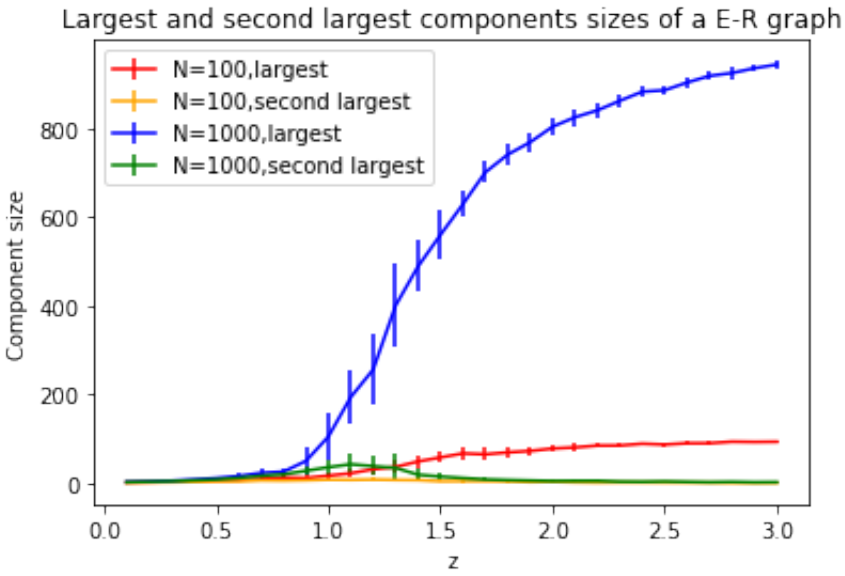
2a Plot the average size of the two largest components in each realisation divided by N, against z for both values of N in a single plot (4 data series in total, use different colours).

Use all 20 (or more) realisations and include error bars indicating the standard deviation.

In [36]:
```
import networkx as nx
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm

mas_N=np.arange(100,1100,step=100)
amt_samples=20
mas_z= np.linspace(0.1,3,30)
mas_graphs=[[] for i in range(len(mas_N))]
for i in range(len(mas_N)):
    N=mas_N[i]
    pN=mas_z/N
    for j in range(len(mas_z)):
        sample_of_graphs=[]
        for k in range(amt_samples):
            sample_of_graphs.append(nx.erdos_renyi_graph(N,pN[j
        mas_graphs[i].append(sample_of_graphs)

matr_largest_component_sizes_mean=np.zeros((len(mas_N),2,len(mas
matr_largest_component_sizes_std=np.zeros((len(mas_N),2,len(mas_
for i in tqdm(range(len(mas_N))):
    for j in range(len(mas_z)):
        #print(i,j)
        mas0=[]
        mas1=[]
        std0=[]
        std1=[]
        for k in range(amt_samples):
            G=mas_graphs[i][j][k]
            Gcc=sorted([G.subgraph(c) for c in nx.connected_comp
            #print(len(Gcc))
            mas0.append(Gcc[0].number_of_nodes())
```

```
            mas1.append(Gcc[1].number_of_nodes())
        #print(mas0,np.mean(mas0))
        #print(mas1,np.mean(mas1))
        matr_largest_component_sizes_mean[i][0][j]=np.mean(mas0
        matr_largest_component_sizes_mean[i][1][j]=np.mean(mas1
        matr_largest_component_sizes_std[i][0][j]=np.std(mas0)
        matr_largest_component_sizes_std[i][1][j]=np.std(mas1)


plt.errorbar(x=mas_z, y=matr_largest_component_sizes_mean[0][0]
plt.errorbar(x=mas_z, y=matr_largest_component_sizes_mean[0][1]
plt.errorbar(x=mas_z, y=matr_largest_component_sizes_mean[-1][0
plt.errorbar(x=mas_z, y=matr_largest_component_sizes_mean[-1][1
plt.legend()
plt.title('Largest and second largest components sizes of a E-R
plt.xlabel('z')
plt.ylabel('Component size')
plt.show()
```

```
100%|██████████| 10/10 [00:27<00:00,  2.73s/it]
```



We see two things:

1) the largest component size (denote it by $C_1(N)$) grows significantly as N grows, while the second component size (denote it by $C_2(N)$) does not show significant growth 2) Behaviour changes at $p = \frac{1}{N}$, that is, for $p = \frac{c}{N}$ with $c < 1$ we have less significant growth, while for $c > 1$ we have very fast growth of largest component size.

In fact, I found in the literature, that:

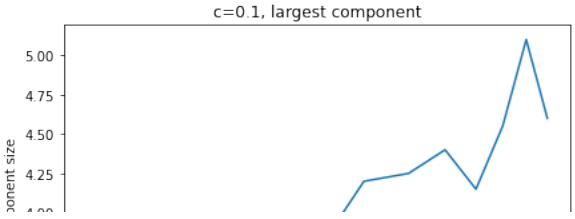
for $c < 1$: $C_1(N) \to \frac{\ln N}{c - \ln c - 1}$ as $N \to \infty$

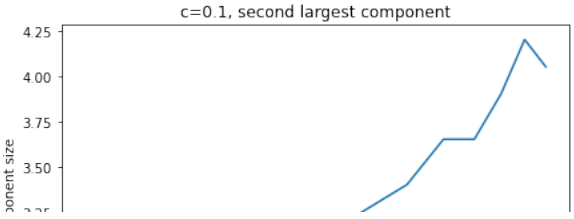
for $c > 1$: $C_1(N) \to n\beta(c)$, where β is the unique solution of equation $\beta + e^{-\beta c} = 1$ on interval $(0, 1)$,

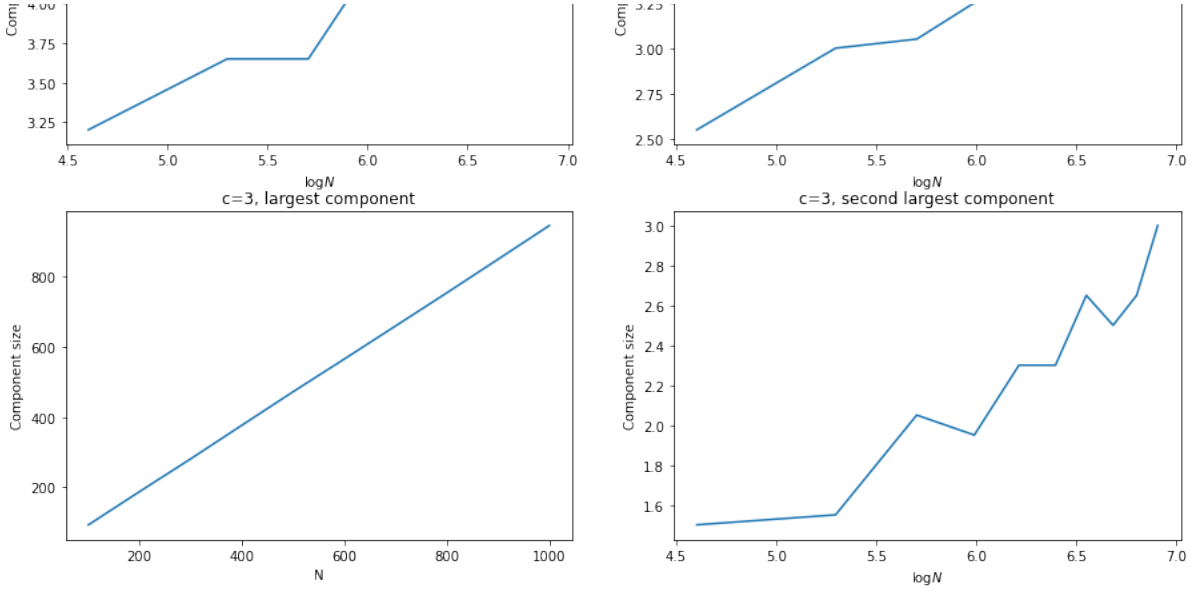
and $C_2(N) \to \frac{\ln N}{c - \ln c - 1}$ as $N \to \infty$. Lets check it for $c = 0.1$ and $c = 3$

We see that for $c = 3$ $C_1(N)$ really grows proportional to N; other three graphs are not looking like lines, may be we need more amt_samples to see the desired behaviour

In [37]:
```
m01_largest=[]
m01_second_largest=[]
m3_largest=[]
m3_second_largest=[]
for i in range(len(mas_N)):
    m01_largest.append(matr_largest_component_sizes_mean[i][0][
    m01_second_largest.append(matr_largest_component_sizes_mean
    m3_largest.append(matr_largest_component_sizes_mean[i][0][-
    m3_second_largest.append(matr_largest_component_sizes_mean[

fig,ax=plt.subplots(2,2)
fig.set_figheight(10)
fig.set_figwidth(15)
ax[0][0].plot(np.log(mas_N), m01_largest,label='m01_largest')
ax[0][0].set_xlabel(r'$\logN$')
ax[0][0].set_ylabel('Component size')
ax[0][0].set_title(r'c=0.1, largest component')

ax[0][1].plot(np.log(mas_N), m01_second_largest,label='m01_seco
ax[0][1].set_xlabel(r'$\logN$')
ax[0][1].set_ylabel('Component size')
ax[0][1].set_title(r'c=0.1, second largest component')

ax[1][0].plot(mas_N, m3_largest,label='m3_largest')
ax[1][0].set_xlabel('N')
ax[1][0].set_ylabel('Component size')
ax[1][0].set_title(r'c=3, largest component')

ax[1][1].plot(np.log(mas_N), m3_second_largest,label='m3_second_
ax[1][1].set_xlabel(r'$\logN$')
ax[1][1].set_ylabel('Component size')
ax[1][1].set_title(r'c=3, second largest component')
plt.show()
```

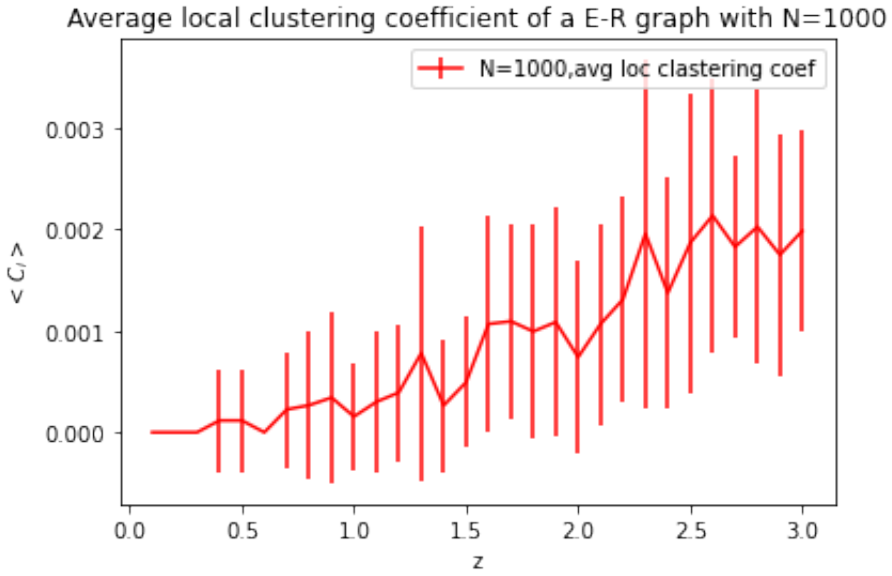






2b For N = 1000, plot the average local clustering coefficient ⟨Ci⟩ against z using all 20 re- alisations and i = 1, . . . , N for averaging, and including error bars indicating the standard deviation for all 20N data points.

In [48]:
```
local_clustering_coef_mean=np.zeros(len(mas_z))
local_clustering_coef_std=np.zeros(len(mas_z))

for j in range(len(mas_z)):
    mas_mean=[]
    mas_std=[]
    for k in range(amt_samples):
        G=mas_graphs[-1][j][k]
        clastering_coefs=nx.clustering(G)
        clastering_coefs=list(map(lambda x: clastering_coefs[x]
        mas_mean.append(np.mean(clastering_coefs))
        mas_std.append(np.std(clastering_coefs))
    local_clustering_coef_mean[j]=np.mean(mas_mean)
    local_clustering_coef_std[j]=np.std(mas_mean)

plt.errorbar(x=mas_z, y=local_clustering_coef_mean,yerr=local_c
plt.legend()
plt.title('Average local clustering coefficient of a E-R graph
plt.xlabel('z')
plt.ylabel(r'$<C_i>$')
plt.show()
```



We see that average local clustering coefficient grows, but very slowly with growth of z, so E-R graphs look more like trees than as clustering nodes, and local clustering coefficient has big standard deviation that does not seem to decrease with the growth of N.

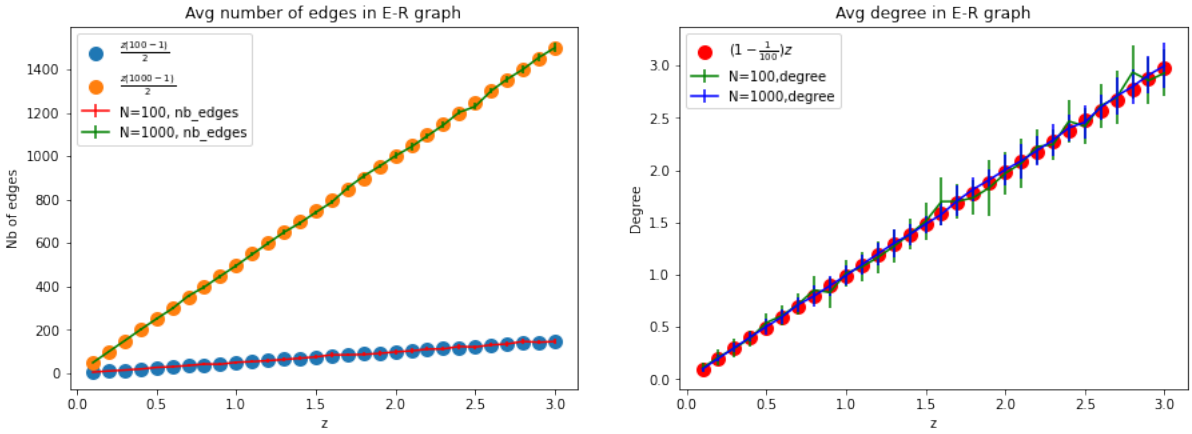
2c Use results in lectures to state what is the expected number of edges, as well as the ex- pected average degree as a function of z and N and plot these as a function of z for the two values of N, comparing your results with the expected ones. Include error bars to indicate the standard deviation for your data points.

In [92]:

```
matr_nb_edges_mean=np.zeros((len(mas_N),len(mas_z)))
matr_nb_edges_std=np.zeros((len(mas_N),len(mas_z)))
matr_degree_mean=np.zeros((len(mas_N),len(mas_z)))
matr_degree_std=np.zeros((len(mas_N),len(mas_z)))
for i in tqdm(range(len(mas_N))):
    for j in range(len(mas_z)):
        #print(i,j)
        mas_nb_edges=[]
        mas_degree=[]
        for k in range(amt_samples):
            G=mas_graphs[i][j][k]
            nb_edges=G.number_of_edges()
            degree=np.mean(list(map(lambda x: x[1],G.degree)))
            mas_nb_edges.append(nb_edges)
            mas_degree.append(degree)
        matr_nb_edges_mean[i][j]=np.mean(mas_nb_edges)
        matr_nb_edges_std[i][j]=np.std(mas_nb_edges)
        matr_degree_mean[i][j]=np.mean(mas_degree)
        matr_degree_std[i][j]=np.std(mas_degree)

fig,ax=plt.subplots(1,2)
fig.set_figheight(5)
fig.set_figwidth(15)

ax[0].errorbar(x=mas_z, y=matr_nb_edges_mean[0],yerr=matr_nb_ed
ax[0].errorbar(x=mas_z, y=matr_nb_edges_mean[-1],yerr=matr_nb_e
ax[0].scatter(mas_z,(100-1)/2*mas_z,label=r'$\frac{z(100-1)}{2}
ax[0].scatter(mas_z,(1000-1)/2*mas_z,label=r'$\frac{z(1000-1)}{
ax[0].set_xlabel('z')
ax[0].set_ylabel('Nb of edges')
ax[0].legend()
ax[0].set_title('Avg number of edges in E-R graph')

ax[1].errorbar(x=mas_z, y=matr_degree_mean[0],yerr=matr_degree_
ax[1].errorbar(x=mas_z, y=matr_degree_mean[-1],yerr=matr_degree
ax[1].scatter(mas_z,(1-1/100)*mas_z,linewidths=5,label=r'$(1-\f
ax[1].set_xlabel('z')
ax[1].set_ylabel('Degree')
ax[1].legend()
```

```
ax[1].set_title('Avg degree in E-R graph')

plt.show()
```

```
100%|██████████| 10/10 [00:02<00:00,  3.97it/s]
```



From lectures we know that expected number of edges is $E(K) = p\frac{N(N-1)}{2}$ and average degree is $E[< k >] = (N-1)p$; We have $P_N = \frac{z}{N}$, so expected number of edges is $E(K) = \frac{z(N-1)}{2}$ and and average degree is $E[< k >] = z(1 - \frac{1}{N})$;

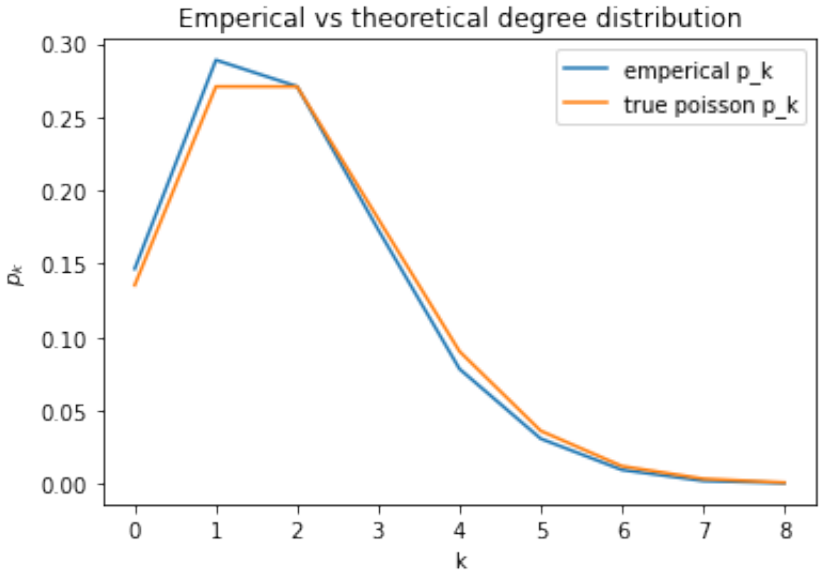
Avg degree is clearly right - the graph shows as a line with slope nearly 1; And number of edges also goes in line with theoretical (scattered) average number of edges.

2d For N = 1000 and your favourite value of z ∈ [0.5, 2], plot the degree distribution p(k) against k = 0, 1, . . . using all 20 realisations, and compare it to the mass function of the Poi(z) Poisson distribution in a single plot.

In [121]:

```
import scipy
from scipy import stats

def degree_distribution(G):
    vk = dict(G.degree())
    vk = list(vk.values())
    vk = np.array(vk)        # store degree values in array

    maxk = np.max(vk)
    k = np.arange(0,maxk+1) # possible values of k

    pk = np.zeros(maxk+1) # degree distribution p(k)
    for i in vk:
        pk[i] = pk[i] + 1
```

```
    pk = pk/sum(pk) # the sum of the elements of P(k) must to b

    return k,pk

mas_pks=[]
z=mas_z[19] #z=2
dict_ks=dict()
for s in range(amt_samples):
    G=mas_graphs[1][19][s]
    k,pk=degree_distribution(G)
    for a,b in zip(k,pk):
        if a not in dict_ks:
            dict_ks[a]=b
        else:
            dict_ks[a]+=b
mas_k=[]
mas_pk=[]
for k in dict_ks.keys():
    mas_k.append(k)
    mas_pk.append(dict_ks[k]/amt_samples)

mas_pk=np.array(mas_pk)
mas_poisson=[scipy.stats.poisson.pmf(k=k,mu=z) for k in mas_k]
plt.plot(mas_k,mas_pk,label='emperical p_k')
plt.plot(mas_k,mas_poisson, label='true poisson p_k')
plt.xlabel('k')
plt.ylabel(r'$p_k$')
plt.legend()
plt.title('Emperical vs theoretical degree distribution')
plt.show()
```

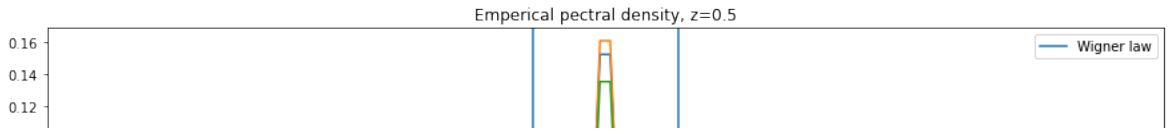


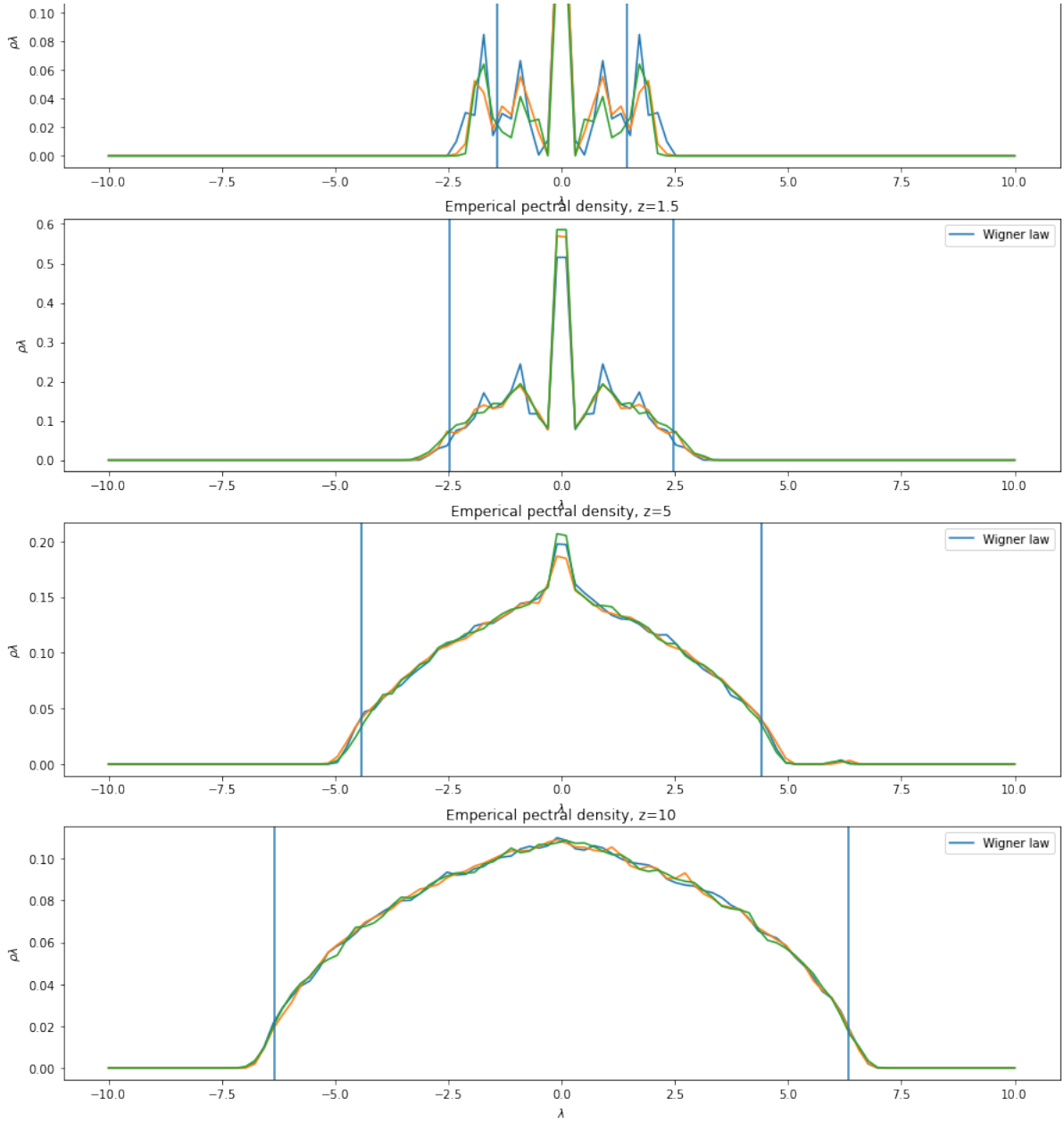
We see that for chosen $z = 2$ emperical and theoretical degree distributions are very close!

2e Consider z = 0.5, 1.5, 5 and 10. Plot the spectrum of the adjacency matrix A using all 20 realisations with a kernel density estimate, and compare it to the Wigner semi-circle law. Comment on your results based on what you expect from the lectures.

In [157]:
```
N=1000
amt_samples=3
mas_z= [0.5,1.5,5,10]
mas_kdes=[[] for i in range(len(mas_z))]

lambd = np.linspace(-10,10,100)
fig,ax=plt.subplots(4,1)
fig.set_figheight(18)
fig.set_figwidth(15)


for i in range(len(mas_z)):
    pN=mas_z[i]/N
    sample_of_graphs=[]
    for j in range(amt_samples):
        G=nx.erdos_renyi_graph(N,pN)
        A = nx.to_numpy_matrix(G)    #find adjacency matrix
        sigm=np.std(A.flatten())
        r=2*sigm*np.sqrt(N)
        if (j==0):
            print("Sigma=",sigm, "2*sigm*sqrt(N)=",r)
        evals, evecs = np.linalg.eig(A)  #calculate e'vales
        # find the spectral density using a kernel density esti
        evals=list(map(lambda x: x.real,evals))
        spectral_density = stats.gaussian_kde(evals, bw_method
        ax[i].plot(lambd, spectral_density(lambd))
    ax[i].axvline(x=r,label='Wigner law')
    ax[i].axvline(x=-r)
    ax[i].set_title('Emperical pectral density, z='+str(mas_z[i
    ax[i].set_xlabel(r'$\lambda$')
    ax[i].set_ylabel(r'$\rho\lambda$')
    ax[i].legend()
plt.show()
```

```
Sigma= 0.023058989049826093 2*sigm*sqrt(N)= 1.4583785187666467
Sigma= 0.037363693554037194 2*sigm*sqrt(N)= 2.363087468546182
Sigma= 0.07046346500137501 2*sigm*sqrt(N)= 4.456500824638093
Sigma= 0.1001563879340704 2*sigm*sqrt(N)= 6.334446161741373
```





We see that:

for $z > 1$ Wigner semi-circle law holds with support $4\sqrt{N}\sigma_N$, where σ_N is the standad deviation of adjacency matric coefficients.

for $z < 1$ the asymptotic spectra; density deviates from Wigner semi-circle spectral density.

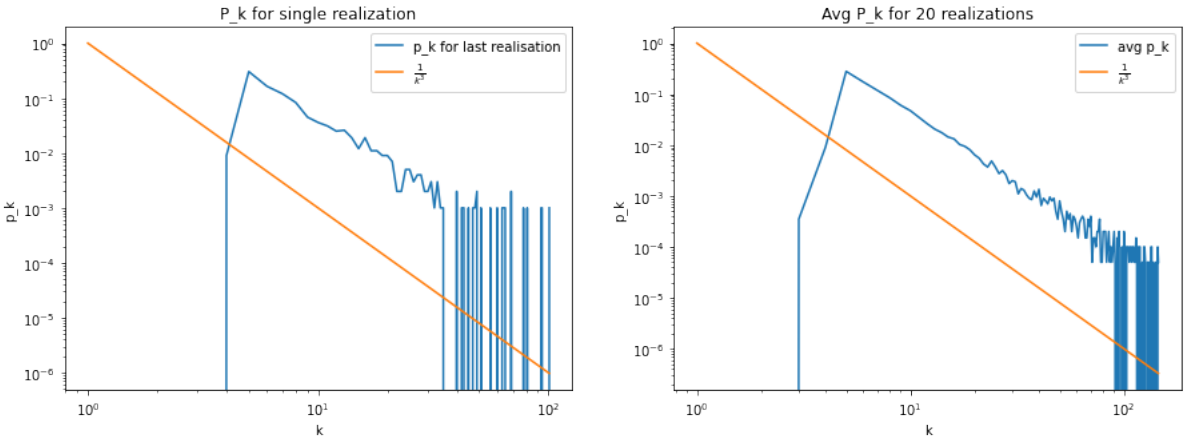
3 Baraba´si-Albert model

In [339]:
```
import numpy as np
from numpy.random import choice

ba_graphs=[]
```

```
def generate_barabasi_albert(m0,m,N):
    A=np.zeros((N,N))
    for i in range(m0):
        for j in range(m0):
            if i!=j:
                A[i][j]=1
    for t in range(1,N-m0+1):
        mas_ks=A[0:m0+t-1,0:m0+t-1].sum(axis=1)
        #print(mas_ks)
        mas_ks/=sum(mas_ks)
        possible_nodes=np.arange(len(mas_ks))
        realized_nodes = np.random.choice(possible_nodes, m, p=
        #print(realized_nodes)
        for i in realized_nodes:
            A[m0+t-1,i]=1
            A[i,m0+t-1]=1
    return nx.from_numpy_array(A)

m0=5
m=5
N=1000
amt_samples=20
for i in range(amt_samples):
    G=generate_barabasi_albert(m0,m,N)
    degree=np.mean(list(map(lambda x: x[1],G.degree)))
    #print("Avg degree=", degree) #is is 2m=10
    ba_graphs.append(G)

mas_pks=[]
dict_ks=dict()
for G in ba_graphs:
    k,pk=degree_distribution(G)
    for a,b in zip(k,pk):
        if a not in dict_ks:
            dict_ks[a]=b
        else:
            dict_ks[a]+=b
mas_k=[]
mas_pk=[]
for kk in dict_ks.keys():
    mas_k.append(kk)
    mas_pk.append(dict_ks[kk]/amt_samples)

fig,ax=plt.subplots(1,2)
fig.set_figheight(5)
fig.set_figwidth(15)

ax[0].loglog(k,pk,label='p_k for last realisation')
ax[0].loglog(k,1/k**3,label=r'$\frac{1}{k^3}$')
ax[0].set_xlabel('k')
ax[0].set_ylabel('p_k')
ax[0].legend()
ax[0].set_title('P_k for single realization')
```

```
ax[1].loglog(mas_k,mas_pk,label="avg p_k")
ax[1].loglog(mas_k,1/(np.array(mas_k)**3),label=r'$\frac{1}{k^3
ax[1].set_xlabel('k')
ax[1].set_ylabel('p_k')
ax[1].legend()
ax[1].set_title('Avg P_k for 20 realizations')

plt.show()
```

```
<ipython-input-339-6a5711e26d54>:54: RuntimeWarning: divide by zer
o encountered in true_divide
  ax[0].loglog(k,1/k**3,label=r'$\frac{1}{k^3}$')
<ipython-input-339-6a5711e26d54>:62: RuntimeWarning: divide by zer
o encountered in true_divide
  ax[1].loglog(mas_k,1/(np.array(mas_k)**3),label=r'$\frac{1}{k^3}
$')
```



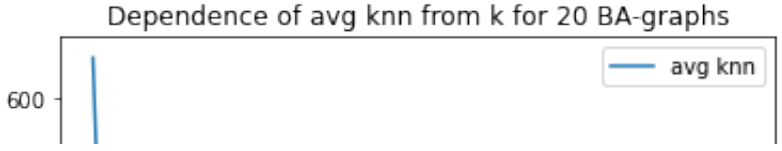
We see that in log-log plot the dependence of p_k on k is linear and the coefficient is -3, because the slope of the orange line (we expect p_k be proportional to k^{-3} from lectures) is the same as the slope of line that could approximate blue plot.

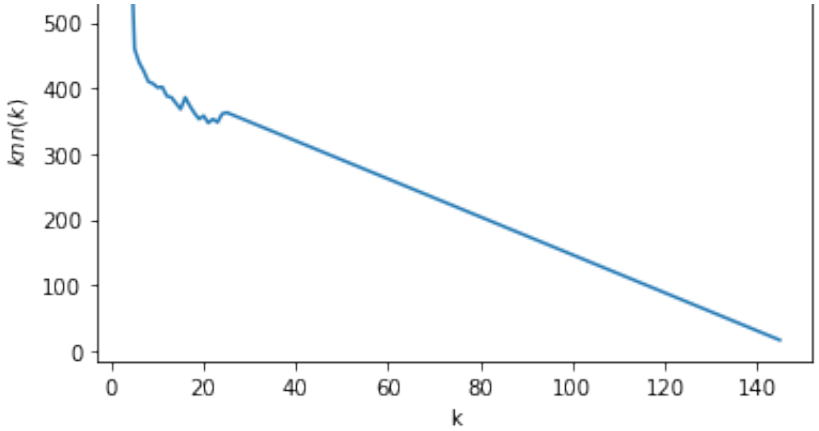
3b Compute the nearest neighbour and decide whether the graphs are typically uncorrelated or (dis-)assortative.

In [340]:

```
def calculate_knn(G,k):
    A = nx.to_numpy_matrix(G)
    ki_s=A.sum(axis=1).astype(int)
    max_k=int(max(ki_s))
#     if k==0:
#         print(max_k, min(ki_s),sum(ki_s), 2*G.number_of_edges
    knn_i_s=A.dot(ki_s.reshape((-1,1)))/ki_s
    num=0
    denom=0
```

```
    for i,ki in enumerate(ki_s):
        if ki==k:
            num+=knn_i_s[i]
            denom+=1
    if denom !=0:
        return max_k, num/denom
    else:
        return max_k, np.infty


dict_knns=dict()
for G in ba_graphs:
    mas_knn=[]
    k_max, knn_tek=calculate_knn(G,0)
    mas_knn.append(knn_tek)
    if 0 not in dict_knns:
        dict_knns[0]=knn_tek
    else:
        dict_knns[0]+=knn_tek
    #print("k_max=",k_max)
    for k in range(1,k_max+1):
        _,knn_tek=calculate_knn(G,k)
        if knn_tek <np.infty:
            knn_tek=int(knn_tek)
        mas_knn.append(knn_tek)
        if k not in dict_knns:
            #print(k,knn_tek)
            dict_knns[k]=knn_tek
        else:
            dict_knns[k]+=knn_tek


mas_k=[]
mas_knns=[]
for k in dict_knns.keys():
    mas_k.append(k)
    mas_knns.append(dict_knns[k]/amt_samples)
mas_k=[]
mas_knns=[]
for k in dict_knns.keys():
    if dict_knns[k]<np.inf:
        mas_k.append(k)
        mas_knns.append(dict_knns[k])
plt.plot(mas_k,mas_knns,label='avg knn')
plt.xlabel('k')
plt.ylabel(r'$knn(k)$')
plt.legend()
plt.title(r'Dependence of avg knn from k for 20 BA-graphs')
plt.show()
```





Note that we does not plot those $(k, knn(k))$ for which average $knn(k)$ in infinity; From the graph we see that $knn(k)$ depends on k - so the graphs are typycally correlated - this is strange because in the lecture it was written that BA graphs are uncorrelated - so may be we calculated not knn. And also $knn(k)$ is decreasing, so the graph is disaassertive.

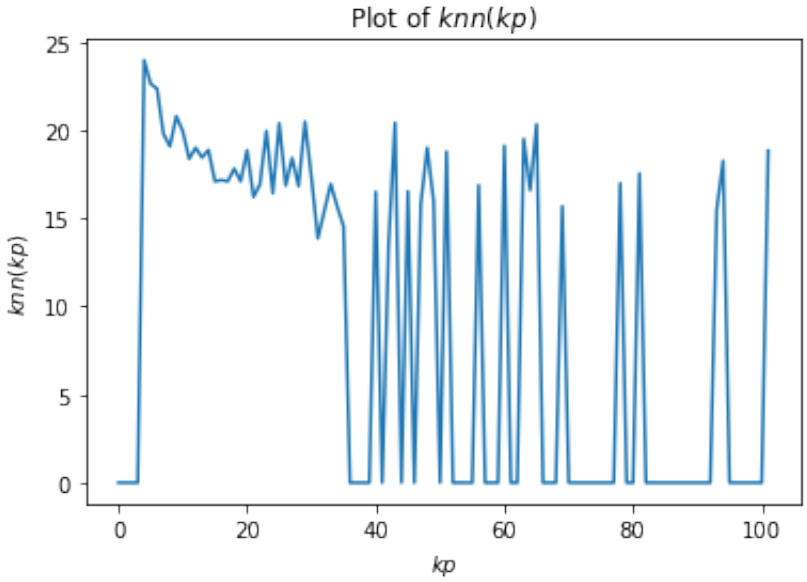
In [341]:
```
G=ba_graphs[-1]
A=nx.to_numpy_matrix(G)
ki_s=list(map(lambda x: x[1], G.degree))
k_max=max(ki_s)
matr_q=np.zeros((k_max+1,k_max+1))
for k in tqdm(range(k_max+1)):
    #print(k)
    for kp in range(k_max+1):
        q=0
        for i in range(A.shape[0]):
            for j in range(A.shape[1]):
                if (ki_s[i]==k and ki_s[j]==kp):
                    #print(A[i,j])
                    #print(k,kp,i,j,A[i,j])
                    q+=A[i,j]
        q/=A.sum()
        matr_q[k,kp]=q


```

```
100%|██████████| 102/102 [13:06<00:00,  7.71s/it]
```

In [356]:
```
matr_q.sum()
```

Out[356]:
```
1.0
```

In [381]:
```
# compute knn(kp) for different kp
mas_knn=[]
q_kp=matr_q.sum(axis=0)

for kp in range(k_max+1):
    s=0
    for k in range(k_max+1):
        s+=k*matr_q[k,kp]
    if q_kp[kp]!=0:
        s/=q_kp[kp]
    mas_knn.append(s)
plt.plot(np.arange(k_max+1),mas_knn)
plt.title(r'Plot of $knn(kp)$')
plt.xlabel(r'$kp$')
plt.ylabel(r'$knn(kp)$')
plt.show()
```



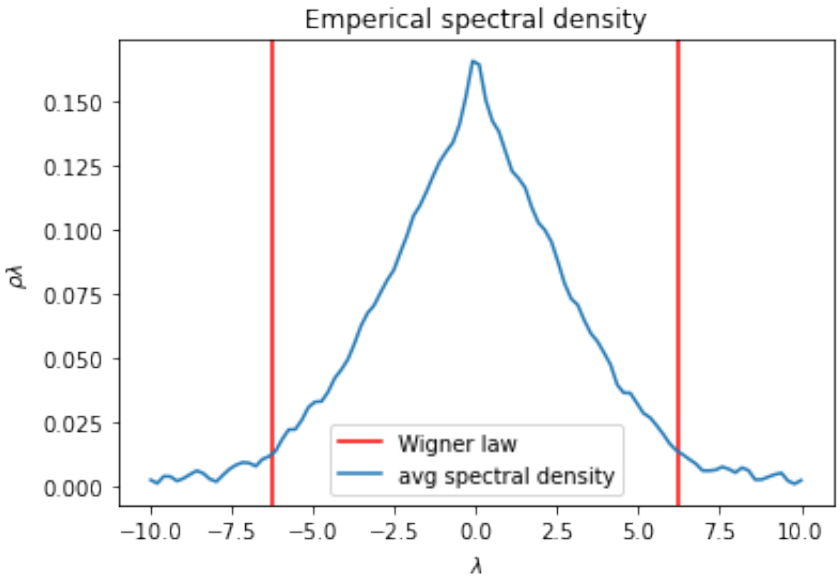
Now it is better, because knn looks like smth not depending on kp. If we averaged knn on 20 graphs, then the graph will become almost a line.

3c Plot the spectrum of the adjacency matrix A = (aij) using all realisations with a ker- nel density estimate, and compare it to the Wigner semi-circle law with σ2 = var[aij]. Comment on your results.

In [384]:
```
A_avg=np.zeros_like(A)
for G in ba_graphs:
    A=nx.to_numpy_matrix(G)
    for i in range(A.shape[0]):
            for j in range(A.shape[1]):
                A_avg[i,j]+=A[i,j]
A_avg/=amt_samples
```

In [397]:
```
sigm=np.std(A.flatten())
r=2*sigm*np.sqrt(N)
print("r=",r)
evals, evecs = np.linalg.eig(A)  #calculate e'vales
# find the spectral density using a kernel density estimate
evals=list(map(lambda x: x.real,evals))
spectral_density = stats.gaussian_kde(evals, bw_method = 0.05)
plt.title('Emperical spectral density')
plt.xlabel(r'$\lambda$')
plt.ylabel(r'$\rho\lambda$')
plt.axvline(x=r,label='Wigner law',color='red')
plt.axvline(x=-r,color='red')
plt.plot(lambd, spectral_density(lambd),label='avg spectral den
plt.legend()
plt.show()
```

```
r= 6.2340114391938695
```



We see that Wigner semi-circle law holds with support $4\sqrt{N}\sigma_N$, where σ_N is the standad deviation of adjacency matric coefficients.

In [ ]:
```

```