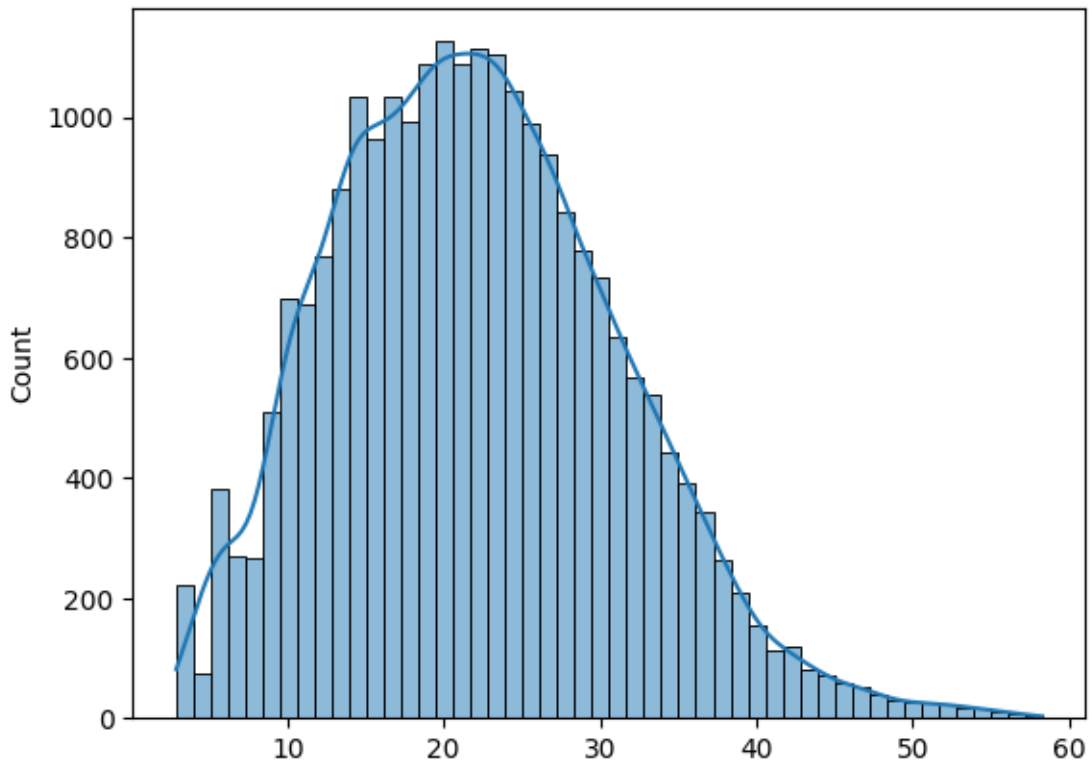
Notebook

April 17, 2023

[35]:
```
import scipy as sp
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
```

[36]:
```
data = sp.io.loadmat('coord1PHP.mat')
protdist = data['protdist'][:,0]
A = data['A']
coord = data['coord']
Nnode = A.shape[0]

sns.histplot(protdist, kde=True, bins=50)
```

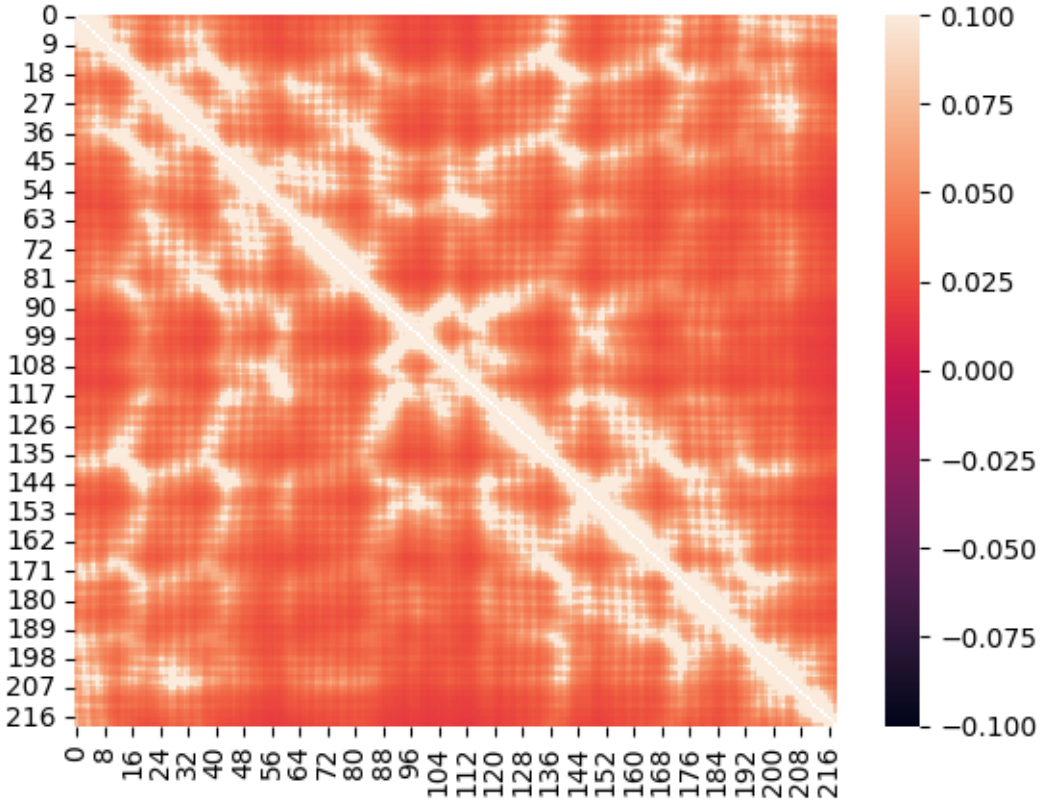
[36]:
```
<Axes: ylabel='Count'>
```



[37]:
```
D = sp.spatial.distance.squareform(protdist)
sns.heatmap(1./D)
```

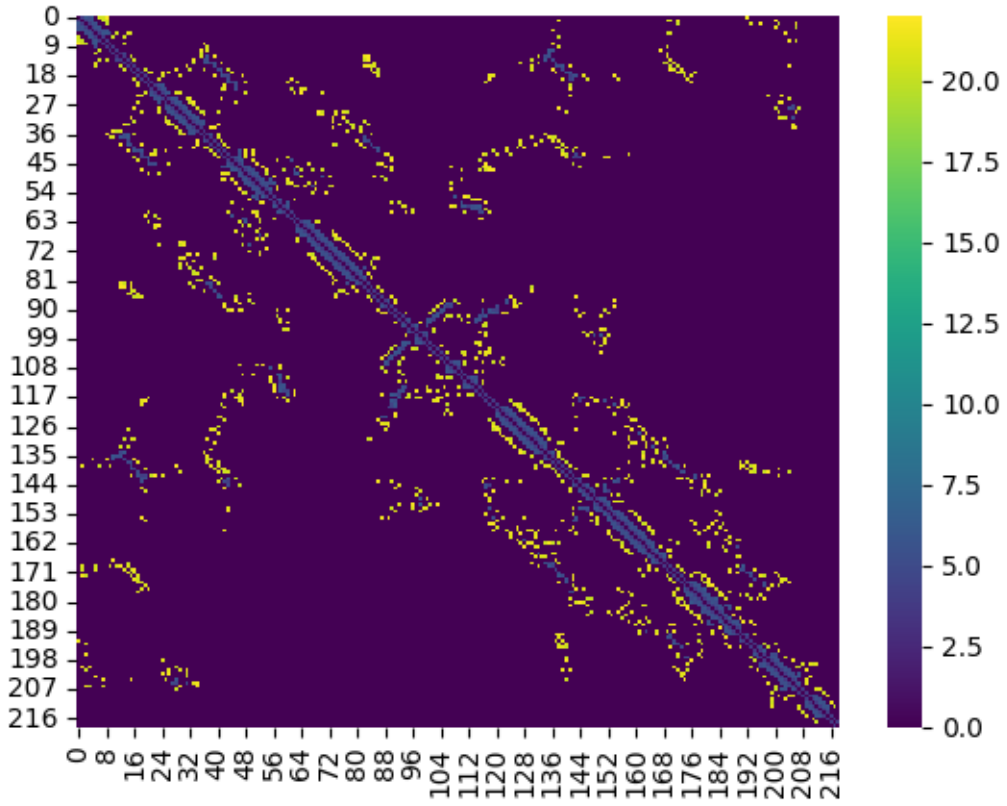
```
/tmp/ipykernel_450/2899854611.py:2: RuntimeWarning: divide by zero encountered
in divide
  sns.heatmap(1./D)
```

[37]: <Axes: >



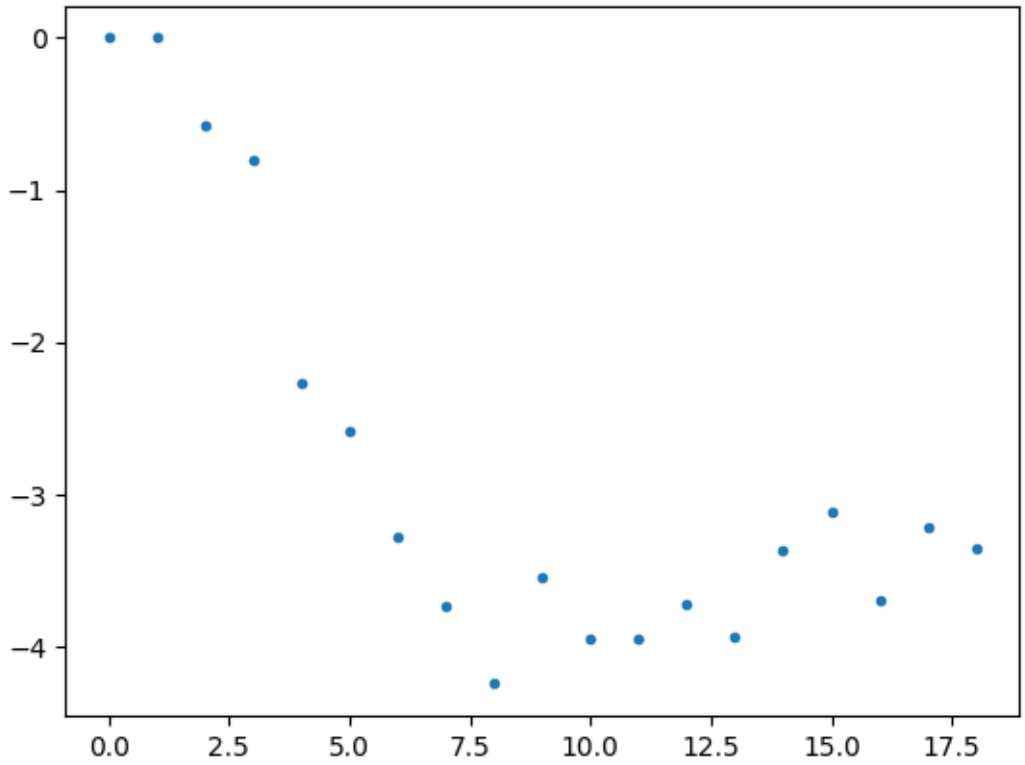
[38]:
```
Net6 = (D < 6) * D
Net10 = (D > 10) * (D < 11) * D
N1 = (D < 11) * D
sns.heatmap(Net6 + 2*Net10, cmap='viridis')
```

[38]: <Axes: >



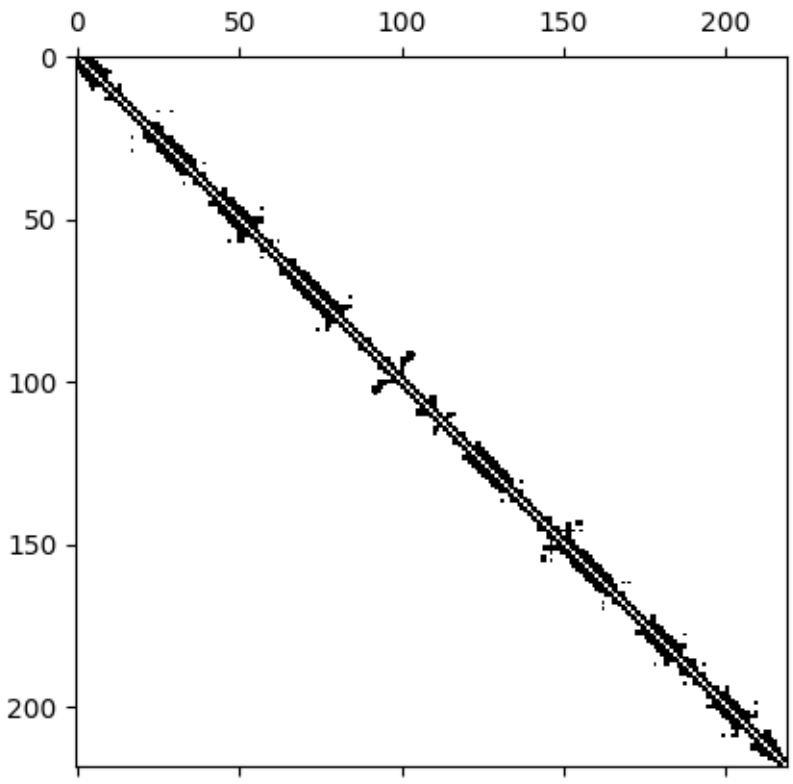
[39]:
```
Kdiag = []

for index in range(Nnode):
    Kdiag.append(np.sum(np.diag(A, index))/(Nnode - index))

plt.plot(np.log(Kdiag[1:20]), '.')
```

[39]: [<matplotlib.lines.Line2D at 0x7f42d241c3d0>]



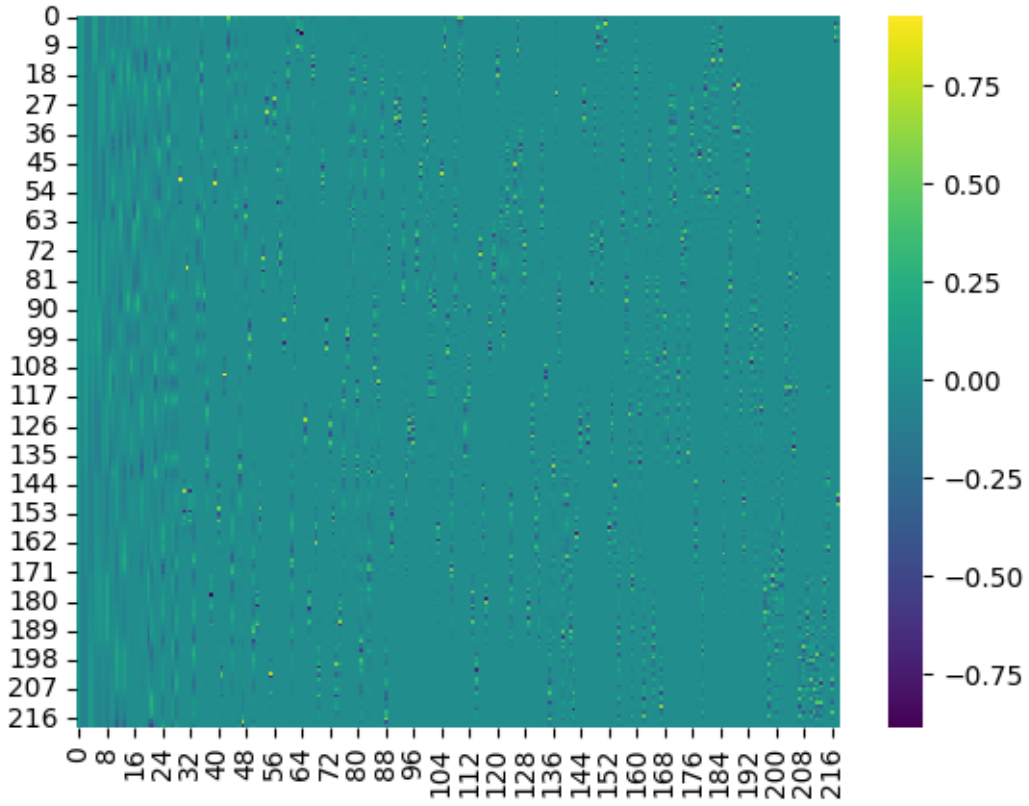
[51]:
```
Nkd = 12
Ad = np.zeros((Nnode, Nnode), dtype=int)

for ind in range(-Nkd, Nkd + 1):
    Ad += np.diagflat(A.diagonal(ind), ind)

plt.spy(Ad)
```

[51]: <matplotlib.image.AxesImage at 0x7f42d0900670>



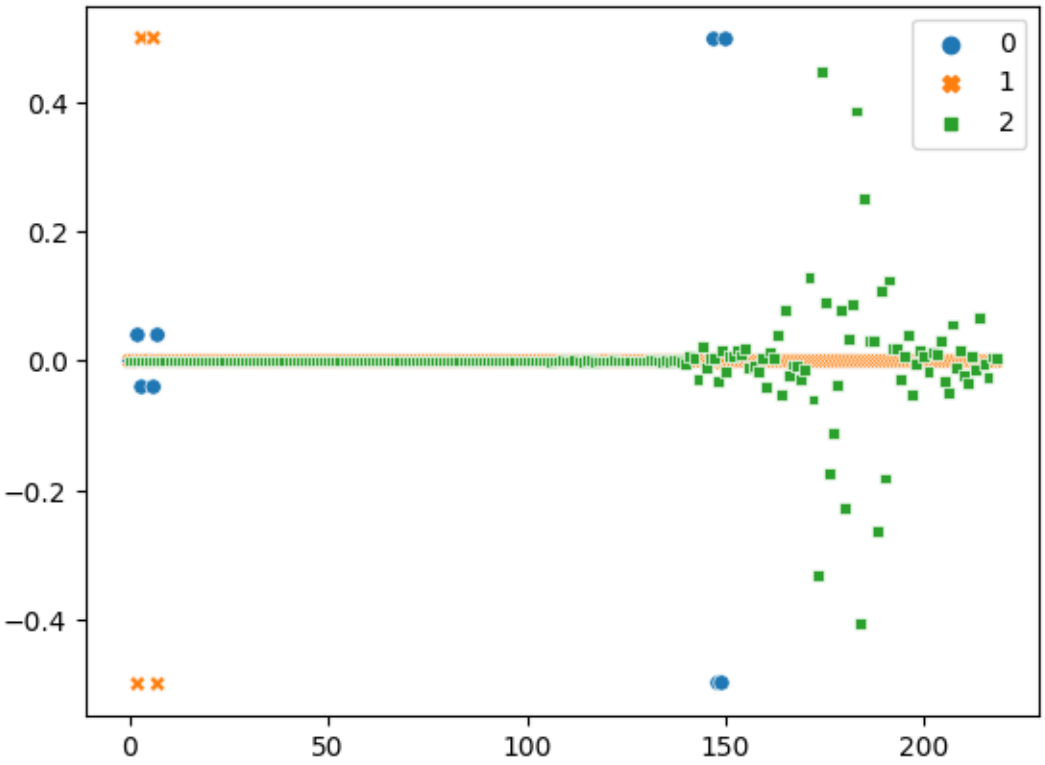
[52]:
```
ValAd, VecAd = np.linalg.eig(sp.sparse.csgraph.laplacian(Ad))
sns.heatmap(VecAd, cmap='viridis')
```

[52]: <Axes: >



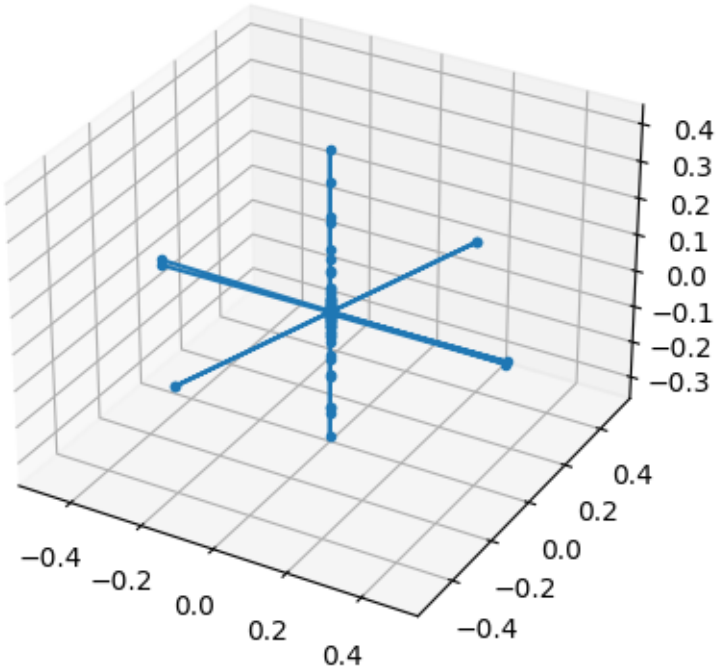
[50]:
```
i1 = Nnode - 1
i2 = Nnode - 2
i3 = Nnode - 3
sns.scatterplot(VecAd[:, [i1, i2, i3]])
```

[50]: <Axes: >



[68]:
```
fig = plt.figure()
ax = fig.add_subplot(projection='3d')

ax.plot3D(VecAd[:, i1], VecAd[:, i2], VecAd[:, i3], marker='.')
```

[68]: [<mpl_toolkits.mplot3d.art3d.Line3D at 0x7f42cfc2ed10>]



[58]:
```
import networkx as nx

D, P = sp.sparse.csgraph.dijkstra(Ad, directed=False, return_predecessors=True)
Cl = 1 / np.mean(D)
K = np.sum(A, axis=0)
BC = nx.betweenness_centrality(nx.from_numpy_array(A))

print(Cl)
print(np.mean(K))
print(BC)
```

```
0.057588259732525154
10.0
{0: 0.0, 1: 0.01204099367300543, 2: 0.01401334954164753, 3:
0.0006162970922864942, 4: 0.003207949937519427, 5: 0.004760302926528808, 6:
0.015745067857777278, 7: 0.00021252745606001165, 8: 0.00021175435907414559, 9:
0.018880116457933395, 10: 0.00888257472519769, 11: 0.0023694854983744664, 12:
0.0094880906468599, 13: 0.02933047382010951, 14: 0.03252025074991718, 15:
0.030695451217927962, 16: 0.038462364866308704, 17: 0.08435553124496131, 18:
0.024875777266538317, 19: 0.01898044341253277, 20: 0.012672017990833909, 21:
0.03203780430616507, 22: 0.020321735104592082, 23: 0.002593975616196208, 24:
0.006316049722693714, 25: 0.035867958406912044, 26: 0.00886974262256941, 27:
0.018539458858847696, 28: 0.029567403288930073, 29: 0.05102375663848864, 30:
```

0.013992145898055398, 31: 0.021094277586248687, 32: 0.022239208860012256, 33:
0.026374887834566138, 34: 0.0026377886461965614, 35: 0.01576424530824187, 36:
0.01584612667926236, 37: 0.013458710054482643, 38: 0.020007909512706208, 39:
0.04026872767018317, 40: 0.04116080043864233, 41: 0.031038824591402815, 42:
0.049506359273908354, 43: 0.07907563229687638, 44: 0.04581152688258064, 45:
0.025131182388419635, 46: 0.03463129057560673, 47: 0.006967532365888618, 48:
0.03873656519658367, 49: 0.01664604905974633, 50: 0.025236811190241043, 51:
0.008069711606234008, 52: 0.012008750054378977, 53: 4.4476800585852575e-05, 54:
0.00032554028818180245, 55: 0.00352953253426607, 56: 0.004601901943844797, 57:
0.021729101317803156, 58: 0.0021898518685196278, 59: 0.007972611015039441, 60:
0.013490805530604694, 61: 0.015217894877977626, 62: 0.014595108310520391, 63:
0.005991598655525304, 64: 0.000977136686718743, 65: 8.779572871367042e-05, 66:
0.00713082915552444, 67: 0.006780980038972298, 68: 0.0015920870216777734, 69:
0.0016156416676844354, 70: 0.017791848819956543, 71: 0.013775533674461092, 72:
0.0033992464645559864, 73: 0.029584864415429072, 74: 0.013202010924415211, 75:
0.0029463454953501124, 76: 0.004832367770598038, 77: 0.008853984296002388, 78:
0.0017218174595826365, 79: 0.0004899953839856271, 80: 0.001831193396308073, 81:
0.0017018359316171938, 82: 0.013174651761843273, 83: 0.008101155134373013, 84:
0.021480055121599707, 85: 0.020440890091434388, 86: 0.013040609677003335, 87:
0.04160345196893862, 88: 0.006992795703025573, 89: 0.008885548648901715, 90:
0.018733337255943523, 91: 0.01669154396358291, 92: 0.014229311136340066, 93:
0.01950489924079859, 94: 0.0019413816919582165, 95: 0.0021947242902580753, 96:
0.0024969387352375494, 97: 0.0021155962689326803, 98: 0.005304111672620396, 99:
0.0018397665359675323, 100: 0.0017114368241285135, 101: 0.0012072615043293763,
102: 0.001706180488012439, 103: 0.001195931596923099, 104:
0.0025517284381880636, 105: 0.006450860097221825, 106: 0.0028488804030948564,
107: 0.02045802291555756, 108: 0.0010351101405947209, 109:
0.0013741651130016568, 110: 0.009067461233343422, 111: 0.0006701891739726152,
112: 0.0005098619105737825, 113: 0.00033585664354907725, 114:
0.004770191645450268, 115: 0.022265054141984365, 116: 0.023376925553120237, 117:
0.05343715784061073, 118: 0.029600132593646702, 119: 0.042523522001271324, 120:
0.016777053826530084, 121: 0.0031890514396772777, 122: 0.0017688215704714278,
123: 0.014933951218284551, 124: 0.019715521257656438, 125: 0.002870722798816541,
126: 0.004360841163996621, 127: 0.030222649717141236, 128: 0.012389527629766302,
129: 0.0034695083944923633, 130: 0.004949597107106605, 131:
0.028225526376477492, 132: 0.004902191941911185, 133: 0.003176831905893899, 134:
0.012982396688985891, 135: 0.005350926055501822, 136: 0.0224118403074471, 137:
0.030644829376982832, 138: 0.018367872687382557, 139: 0.022133454411966274, 140:
0.033270056279473895, 141: 0.07154977878028065, 142: 0.041442192285288816, 143:
0.02974603933453213, 144: 0.029360846576278316, 145: 0.015275114748458208, 146:
0.027027311139692832, 147: 5.4070162780407754e-05, 148: 0.009264316050508576,
149: 0.0012838785151195941, 150: 0.008944705451739204, 151:
0.018733382155081198, 152: 0.009906271919185744, 153: 0.008832610192563775, 154:
0.030530566891168946, 155: 0.021868919339924627, 156: 0.01964725405928555, 157:
0.012867587764840717, 158: 0.025817656809759486, 159: 0.014807401235733852, 160:
0.006212465003977074, 161: 0.010365547118589714, 162: 0.026389643147896703, 163:
0.006702678621782567, 164: 0.0005089854157410398, 165: 0.002790989419934251,
166: 0.0005390301918236331, 167: 0.0011757513436919313, 168:

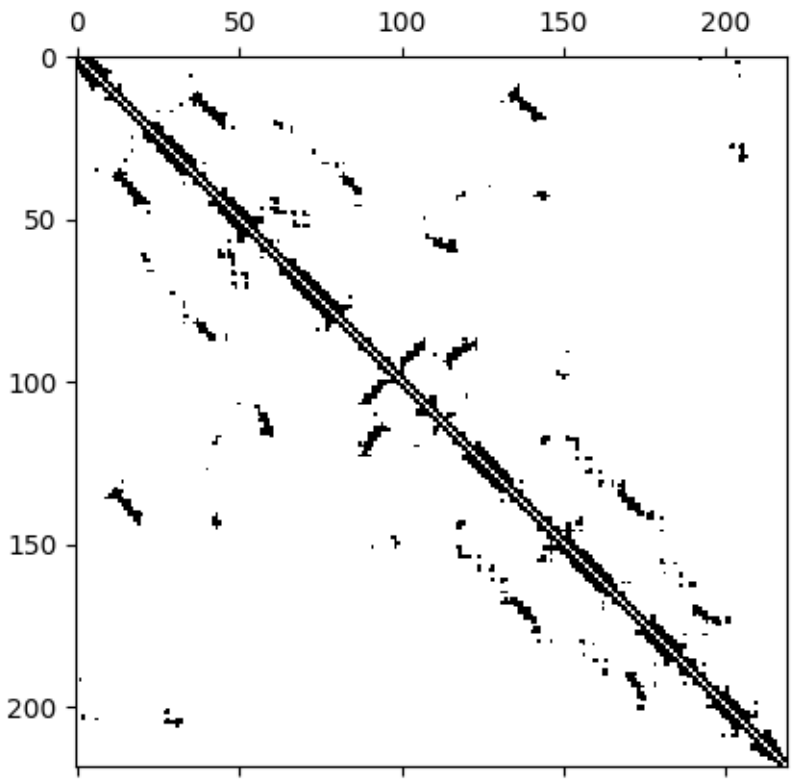
```
0.013708967377539785, 169: 0.0062230404399575, 170: 0.01860456955123589, 171:
0.02017392603687551, 172: 0.02489230843891768, 173: 0.06003089293203244, 174:
0.041275892955877636, 175: 0.005132034272732839, 176: 0.007231489848532819, 177:
0.02289808314820412, 178: 0.02065485568164289, 179: 0.002516859061860833, 180:
0.031175919810288166, 181: 0.015741357790581067, 182: 0.002312050114904961, 183:
0.0008180224023849952, 184: 0.002463079947627114, 185: 0.0005371761968987288,
186: 0.009818127251788943, 187: 0.004417996363606854, 188:
0.0007952555789802801, 189: 0.0013435470994491983, 190: 0.024106549846167332,
191: 0.0029870143027755256, 192: 0.0181704925816694, 193: 0.00871654427885206,
194: 0.010426663364159735, 195: 0.001104736513914226, 196: 0.004940120084855866,
197: 0.006607756060464395, 198: 0.006847534005938651, 199: 0.017862210749160765,
200: 0.029517026611465137, 201: 0.029967784477801136, 202: 0.039952256576713335,
203: 0.03000062376678563, 204: 0.032254913533162385, 205: 0.027821596797377886,
206: 0.014042953740099047, 207: 0.005862523318016057, 208: 0.021187174675247094,
209: 0.05822620705972127, 210: 0.002464162745641879, 211: 0.005387054758404398,
212: 0.008261391650002588, 213: 0.028543381424905416, 214:
0.00011626432165053056, 215: 0.0066772402029717185, 216: 0.011572735134899446,
217: 4.2277935145647484e-05, 218: 0.0}
```

1 Permutation effects

[20]:
```
import scipy as sp
import numpy as np
from matplotlib import pyplot as plt
```

[29]:
```
data = sp.io.loadmat('coord1PHP.mat')
A = data['A']
coord = data['coord']
Nnode = A.shape[0]
plt.spy(A)
```

[29]:
```
<matplotlib.image.AxesImage at 0x7fc8e8b04790>
```



[22]:
```
Index = np.arange(Nnode)
PermIndex = np.random.permutation(Nnode)
PermMat = np.zeros((Nnode,Nnode))
```

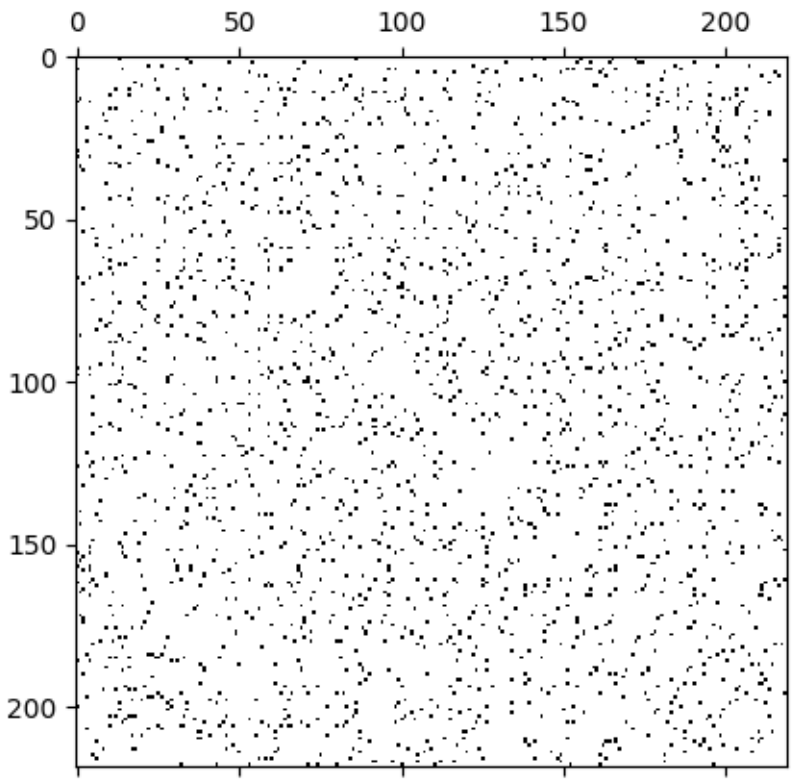
[23]:
```
for ind in Index:
    PermMat[Index[ind], PermIndex[ind]] = 1
```

A faster method could be

[24]:
```
PermMat = sp.sparse.coo_matrix((np.ones(Nnode), (Index, PermIndex)),␣
 ↪shape=(Nnode, Nnode)).toarray()
```

[25]:
```
A2 = PermMat.dot(A).dot(PermMat.T)
plt.spy(A2)
```

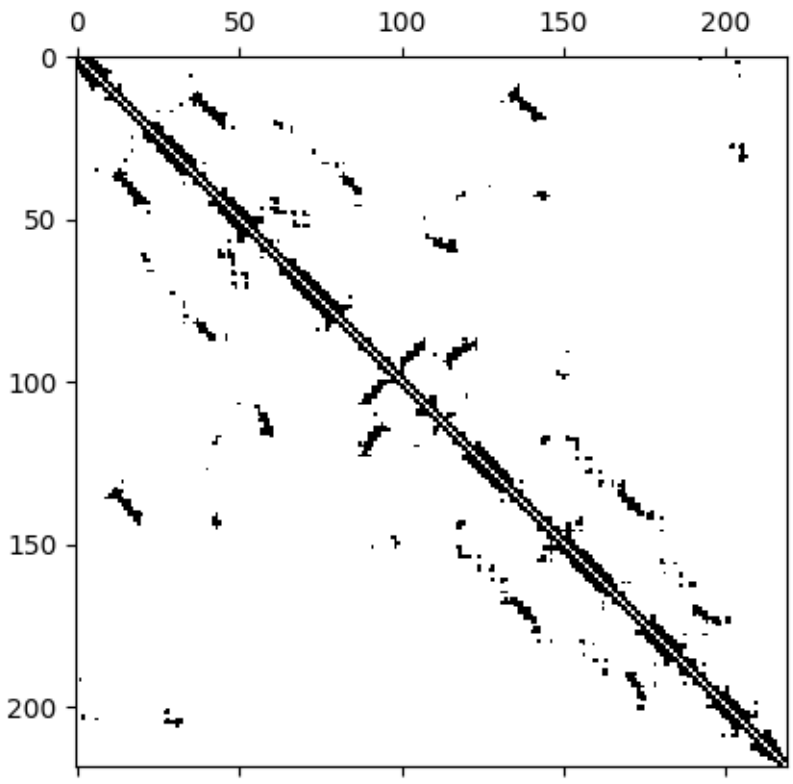
[25]:
```
<matplotlib.image.AxesImage at 0x7fc8e8d9c190>
```



We can invert the permutation effect

[26]:
```
A3 = (PermMat.T).dot(A2).dot(PermMat)
plt.spy(A3)
```

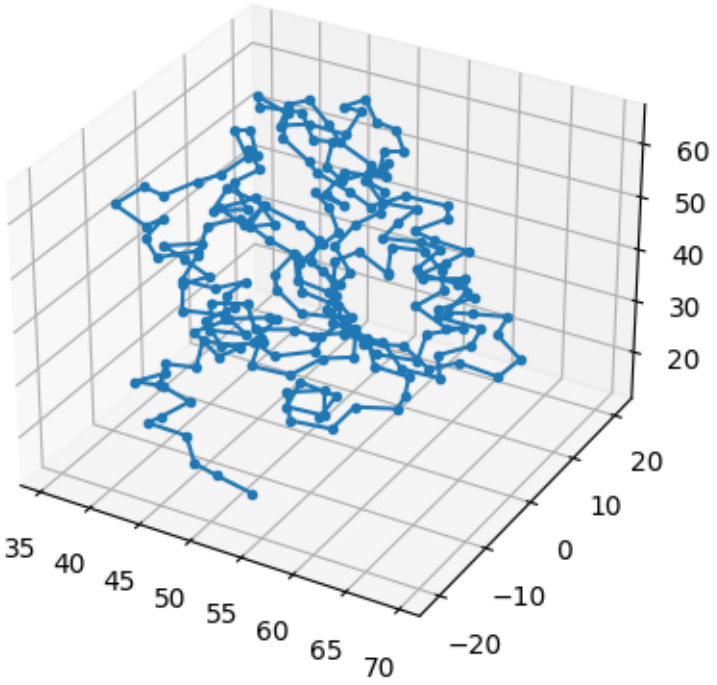
[26]:
```
<matplotlib.image.AxesImage at 0x7fc8e8c082b0>
```



Let's try to visualize the 3d structure of the protein

[38]:
```
fig = plt.figure()
ax = fig.add_subplot(projection='3d')

ax.plot(coord[:,0], coord[:,1], coord[:,2], marker='.')
```

[38]:
```
[<mpl_toolkits.mplot3d.art3d.Line3D at 0x7fc8e19db5b0>]
```



2 Code from ER class

[57]:
```
import scipy as sp
import numpy as np
import sys
import seaborn as sns
import matplotlib.pyplot as plt
import networkx as nx

def rand_canonical_f(Nnode, Plink):
    A = np.triu(np.random.rand(Nnode, Nnode), k=1)
    A = A > 1 - (2*Plink)
    A = A + A.T
    return A
```

Let's compare the memory usage between a sparse and a normal matrix. Moreover, we assume a microcanonical ensemble, fixing the number of nodes and links.

[58]:
```
Nnode = int(1e3)
Nlink = int(3e4)
Net1 = sp.sparse.lil_matrix((Nnode,Nnode))
Net2 = np.zeros((Nnode,Nnode))
```

```
print(sys.getsizeof(Net1))
print(sys.getsizeof(Net2))
```

```
48
8000128
```

2.1 Microcanonical ensemble

Now, let's try to generate a random graph without using Networkx functions.

[59]:
```
for index in range(100):
    # generate a random link
    ii = np.random.randint(Nnode, size=(Nlink, 2))
    # create a sparse matrix using these links
    NetM = sp.sparse.coo_matrix((np.ones(Nlink), (ii[:,0], ii[:,1])),␣
 ↪shape=(Nnode, Nnode))
    #remove loops
    NetM.setdiag(0)
    # symmetrize
    NetM = NetM + NetM.T
print(NetM)
```

```
  (0, 16)       1.0
  (0, 17)       1.0
  (0, 25)       1.0
  (0, 51)       1.0
  (0, 59)       1.0
  (0, 92)       1.0
  (0, 99)       1.0
  (0, 103)      1.0
  (0, 140)      1.0
  (0, 168)      1.0
  (0, 174)      1.0
  (0, 175)      1.0
  (0, 196)      1.0
  (0, 221)      3.0
  (0, 235)      1.0
  (0, 271)      1.0
  (0, 272)      1.0
  (0, 299)      1.0
  (0, 304)      1.0
  (0, 320)      1.0
  (0, 324)      1.0
  (0, 333)      1.0
  (0, 352)      1.0
  (0, 356)      1.0
  (0, 359)      1.0
```

```
:     :
(999, 675)    1.0
(999, 688)    1.0
(999, 695)    1.0
(999, 713)    1.0
(999, 716)    1.0
(999, 719)    1.0
(999, 723)    1.0
(999, 733)    1.0
(999, 742)    1.0
(999, 770)    1.0
(999, 788)    1.0
(999, 790)    1.0
(999, 807)    1.0
(999, 817)    1.0
(999, 821)    1.0
(999, 842)    1.0
(999, 849)    1.0
(999, 855)    1.0
(999, 868)    1.0
(999, 889)    1.0
(999, 891)    1.0
(999, 903)    2.0
(999, 911)    1.0
(999, 960)    1.0
(999, 961)    1.0
```

NOTE that reciprocal links and repeated links randomly created reduce symmetric links with a probability that scales as $\frac{1}{Nnode^2}$

Let's count the number of links in the matrix.

[60]:
```
Nl = int(NetM.count_nonzero()/2)
print('Randomly generated links: {:d}'.format(Nl))
print('Missing links for microcanonical ensemble: {:d}'.format(Nlink-Nl))
```

```
Randomly generated links: 29035
Missing links for microcanonical ensemble: 965
```

We notice that we miss some links, then we add one by one missing links until microcanonical constraint fulfilled.

[61]:
```
while Nl < Nlink:
    i1 = np.random.randint(Nnode)
    i2 = np.random.randint(Nnode)
    if i1 != i2 and NetM[i1,i2] == 0:
        NetM[i1,i2] = 1
        NetM[i2,i1] = 1 # symmetric link
        Nl += 1
```

Lastly, let's compute the degree vector, finding minimum and maximum.

[62]:
```
K = sum(NetM)
print('Average degree: {:.2f}'.format(np.mean(K)))
print('Minimum degree: {:d}'.format(int(np.min(K))))
print('Maximum degree: {:d}'.format(int(np.max(K))))
```

```
Average degree: 61.87
Minimum degree: 39
Maximum degree: 87
```

2.2 Canonical ensemble

Let's now try to build a canonical ensemble.

[63]:
```
Nnode = int(1e2)
Plink = 0.3
Nexp = int(1e4)
```

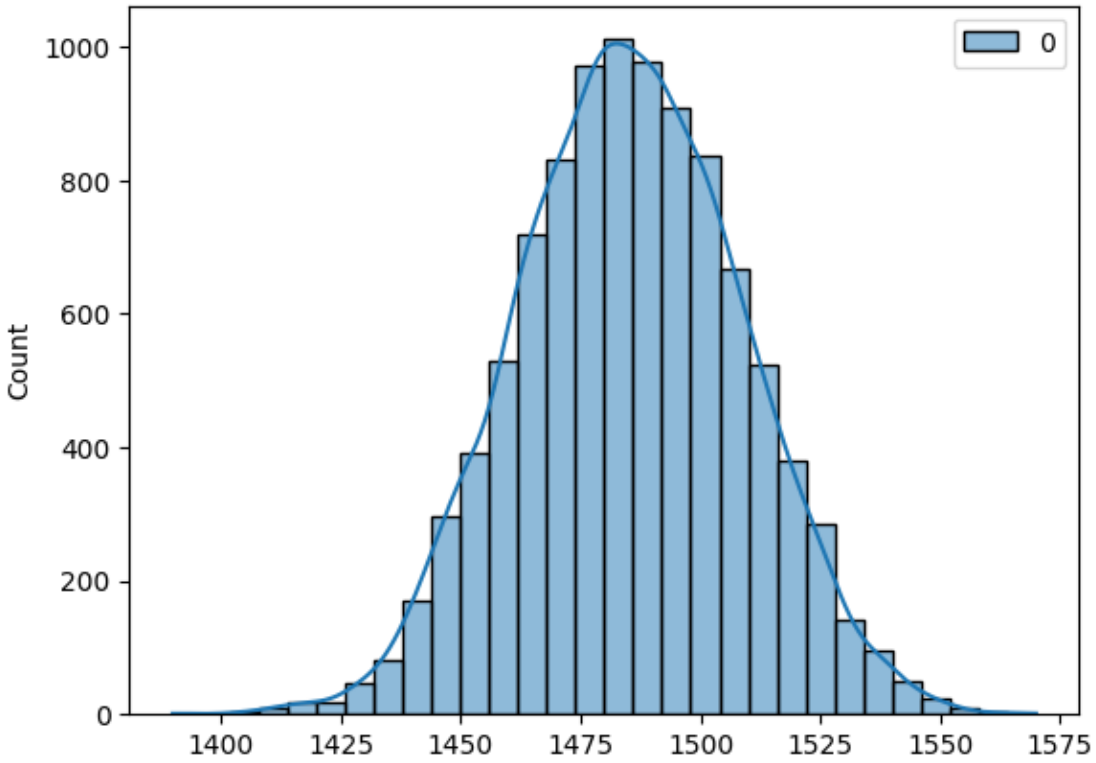
[64]:
```
LinkS = np.zeros((Nexp, 1))
for index in range(Nexp):
    NetC = sp.sparse.triu(sp.sparse.rand(Nnode, Nnode, density=Plink,␣
 ↪format='csr'), k=1)
    NetC = NetC + NetC.T
    LinkS[index] = NetC.count_nonzero()/2
```

Let's see some stats

[65]:
```
print('Average number of links: {:.2f}'.format(np.mean(LinkS)))
print('Standard deviation: {:.2f}'.format(np.std(LinkS)))

print('Expected number of links: {:.2f}'.format(Plink*Nnode*(Nnode-1)/2))
print('Expected standard deviation: {:.2f}'.format(np.
 ↪sqrt(Plink*(1-Plink)*Nnode*(Nnode-1)/2)))

sns.histplot(data=LinkS, bins=30, kde=True)
```

```
Average number of links: 1485.08
Standard deviation: 23.30
Expected number of links: 1485.00
Expected standard deviation: 32.24
```

[65]:
```
<Axes: ylabel='Count'>
```



The degree distribution for increasing size - CV decreases. Tends to the delta function - regular graph.

[66]:
```
Pow = 4.21 # max network size
Pl = 0.1
Cnt = 0

M = []
S = []
CV = []
Nnodes = []
```

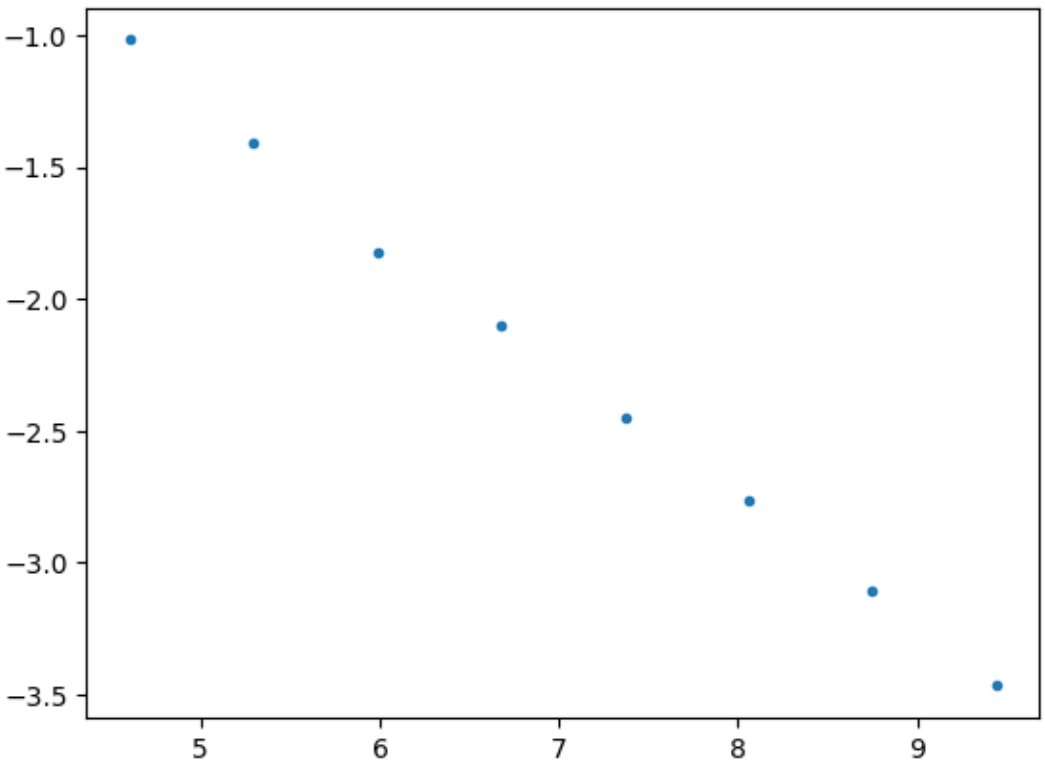
[67]:
```
for index in np.arange(2, Pow, step=0.3):
    Nnodes.append(int(10**index))
    NetC = sp.sparse.triu(sp.sparse.rand(Nnodes[Cnt], Nnodes[Cnt], density=Pl,␣
 ↪format='csr'), k=1)
    NetC = NetC + NetC.T
    K = sum(NetC).todense()
    M.append(np.mean(K))
    S.append(np.std(K))
    CV.append(np.std(K)/np.mean(K))
```

```
    # sns.histplot(data=K, bins=30, kde=True, label='N={:d}'.
↪format(Nnodes[Cnt]))
    print('N={:d} - Average degree: {:.2f}'.format(Nnodes[Cnt], M[Cnt]))
    print('N={:d} - Standard deviation: {:.2f}'.format(Nnodes[Cnt], S[Cnt]))
    print('N={:d} - CV: {:.2f}'.format(Nnodes[Cnt], CV[Cnt]))
    Cnt += 1
```

```
N=100 - Average degree: 4.50
N=100 - Standard deviation: 1.62
N=100 - CV: 0.36
N=199 - Average degree: 9.76
N=199 - Standard deviation: 2.39
N=199 - CV: 0.25
N=398 - Average degree: 19.72
N=398 - Standard deviation: 3.19
N=398 - CV: 0.16
N=794 - Average degree: 39.65
N=794 - Standard deviation: 4.83
N=794 - CV: 0.12
N=1584 - Average degree: 79.53
N=1584 - Standard deviation: 6.86
N=1584 - CV: 0.09
N=3162 - Average degree: 158.26
N=3162 - Standard deviation: 9.98
N=3162 - CV: 0.06
N=6309 - Average degree: 314.95
N=6309 - Standard deviation: 14.07
N=6309 - CV: 0.04
N=12589 - Average degree: 629.21
N=12589 - Standard deviation: 19.65
N=12589 - CV: 0.03
```

[68]:
```
plt.plot(np.log(Nnodes), np.log(CV), '.')
```

[68]:
```
[<matplotlib.lines.Line2D at 0x7fafb89a6860>]
```



2.3 Giant component transition

[69]:
```
Nnode = int(1e2)

rr = np.triu(np.random.rand(Nnode, Nnode), k=1)
rr = rr + rr.T
```

[70]:
```
Pmin = 1 / (20 * Nnode)
Pmax = 6 / Nnode
xP = np.arange(Pmin, Pmax, step=0.00002)
```

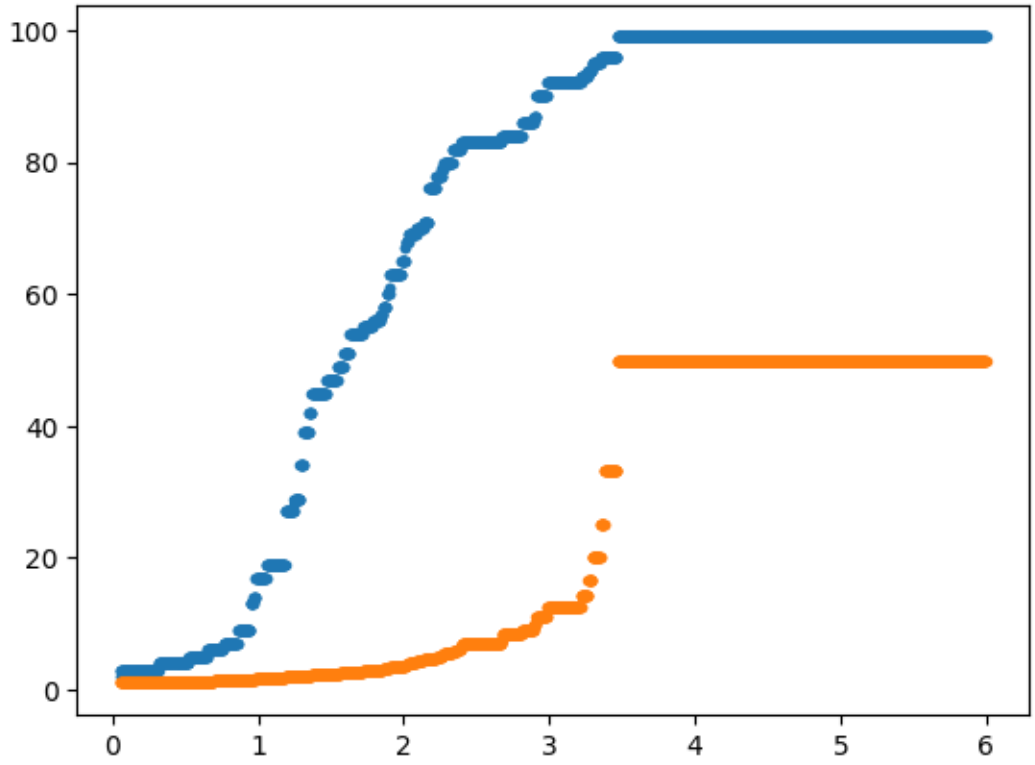
NOTE that $\lambda = xP * Nnode$

[71]:
```
Rc = np.zeros(len(xP))
Rm = np.zeros(len(xP))
Cnt = 0

for index in xP:
    Net = nx.from_numpy_array(rr > 1 - index)
    CompList = nx.connected_components(Net)
    Comps = [len(x) for x in CompList]
    Rc[Cnt] = max(Comps)
```

```
    Rm[Cnt] = np.mean(Comps)
    Cnt += 1
```

[72]:
```
plt.plot(xP*Nnode, Rc, '.')
plt.plot(xP*Nnode, Rm, '.')
```

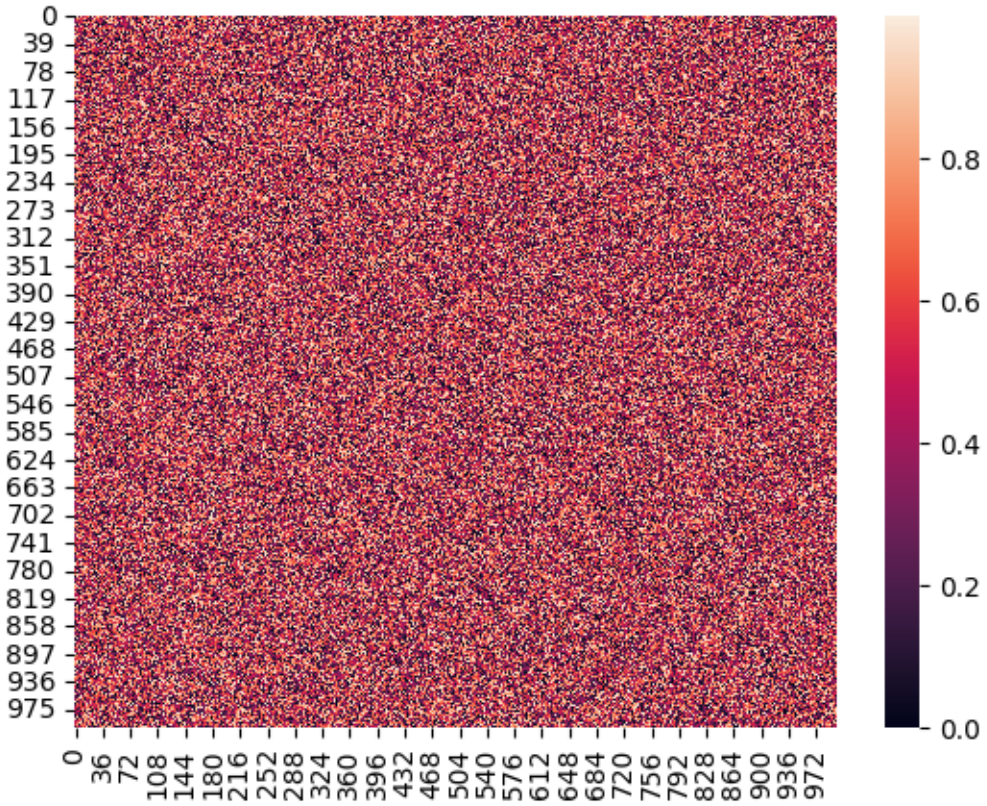
[72]: [<matplotlib.lines.Line2D at 0x7fafb882d990>]



2.4 Semicircle law for ER network spectrum vs RMT (Random Matrix Theory)

[73]:
```
Nnode = int(1e3)
Plink = 0.2 # link probability (canonical)

rr = np.triu(np.random.rand(Nnode, Nnode), k=1)
rr = rr + rr.T
np.fill_diagonal(rr, 0) # remove loops

# plot matrix with values
sns.heatmap(rr)
```

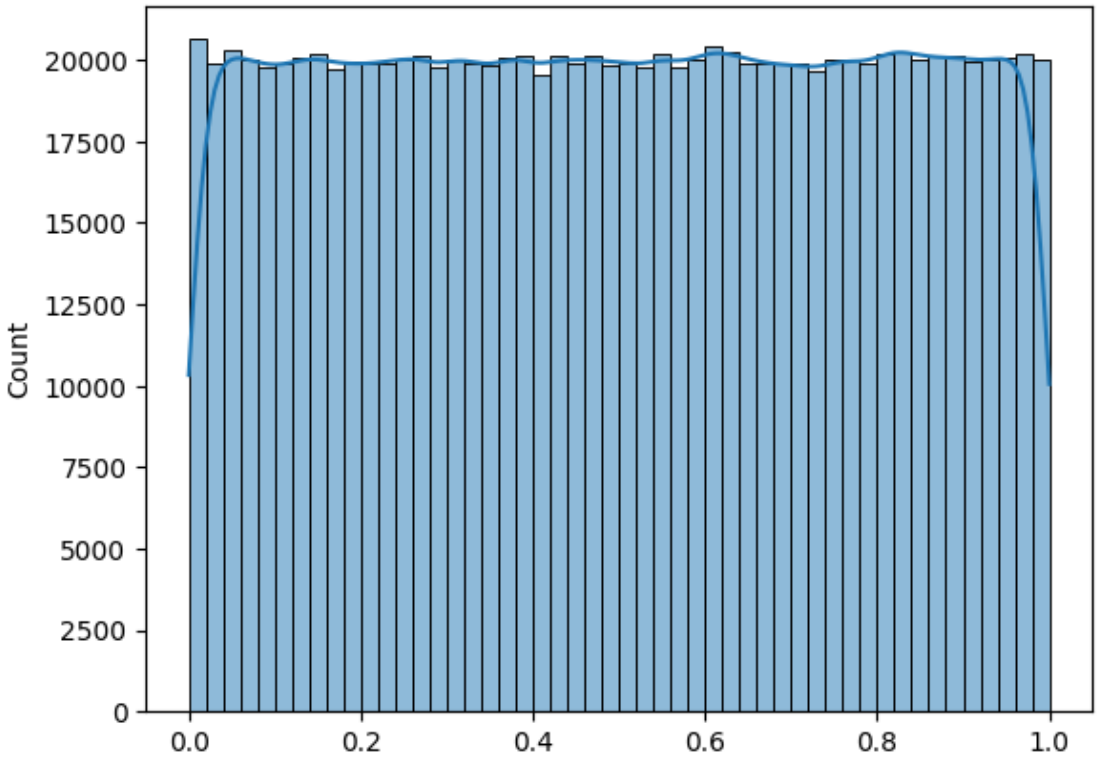
[73]: <Axes: >



Let's see also the histogram of coefficients

[74]:
```
sns.histplot(data=rr.ravel(), bins=50, kde=True)
```

[74]:
```
<Axes: ylabel='Count'>
```

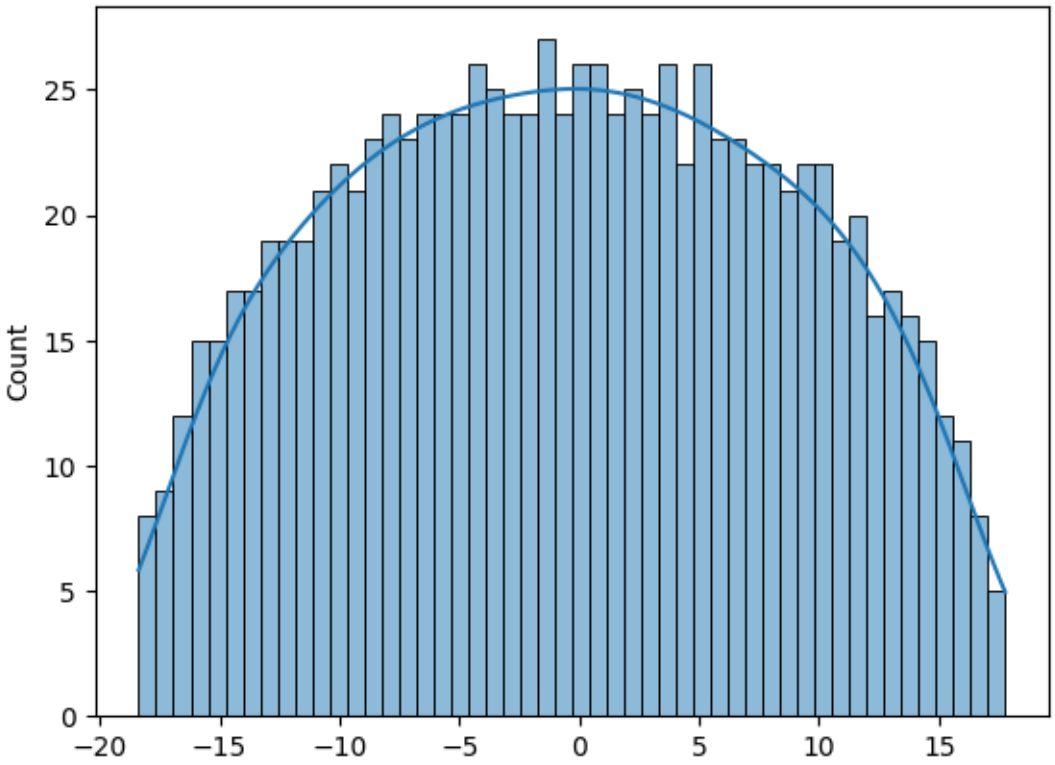


Now the eigenvalues and eigenvectors.

NOTE A symmetric ER network has a real semicircle spectrum

[75]:
```
vv = np.linalg.eigvals(rr)
sns.histplot(data=vv[vv < vv.max()], bins=50, kde=True)
```

[75]:
```
<Axes: ylabel='Count'>
```

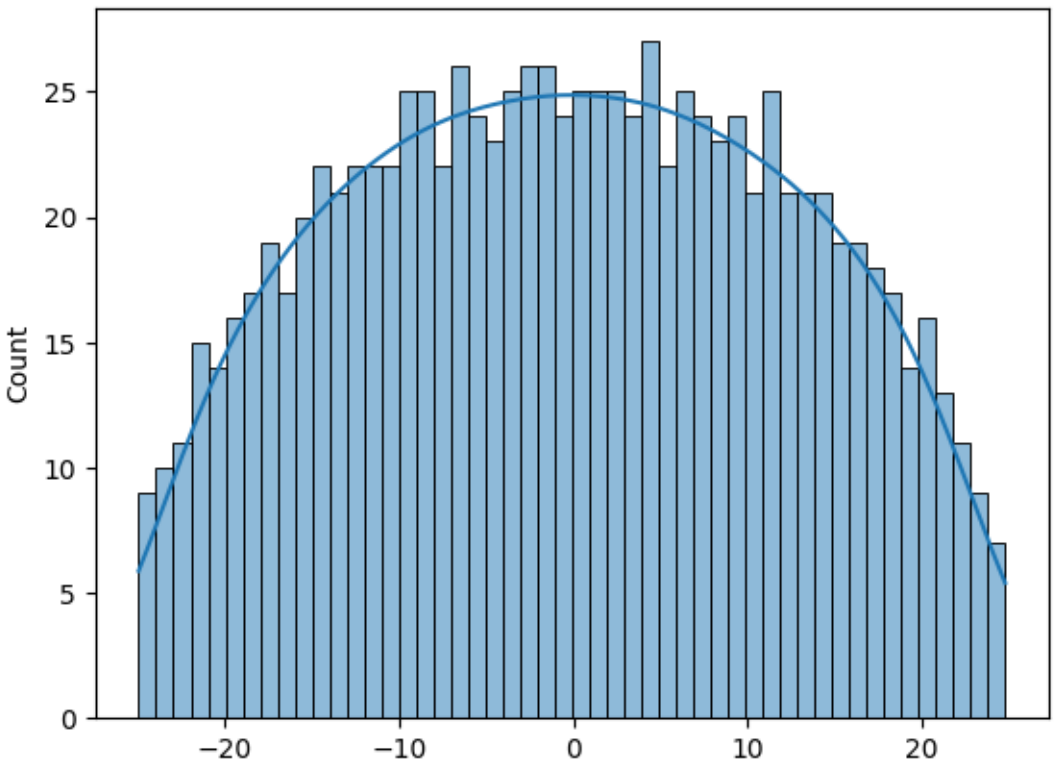


[76]:
```
rr = np.triu(np.random.rand(Nnode, Nnode), k=1) > 1 - Plink
rr = rr + rr.T
```

[77]:
```
vv, vec = np.linalg.eig(rr)
```

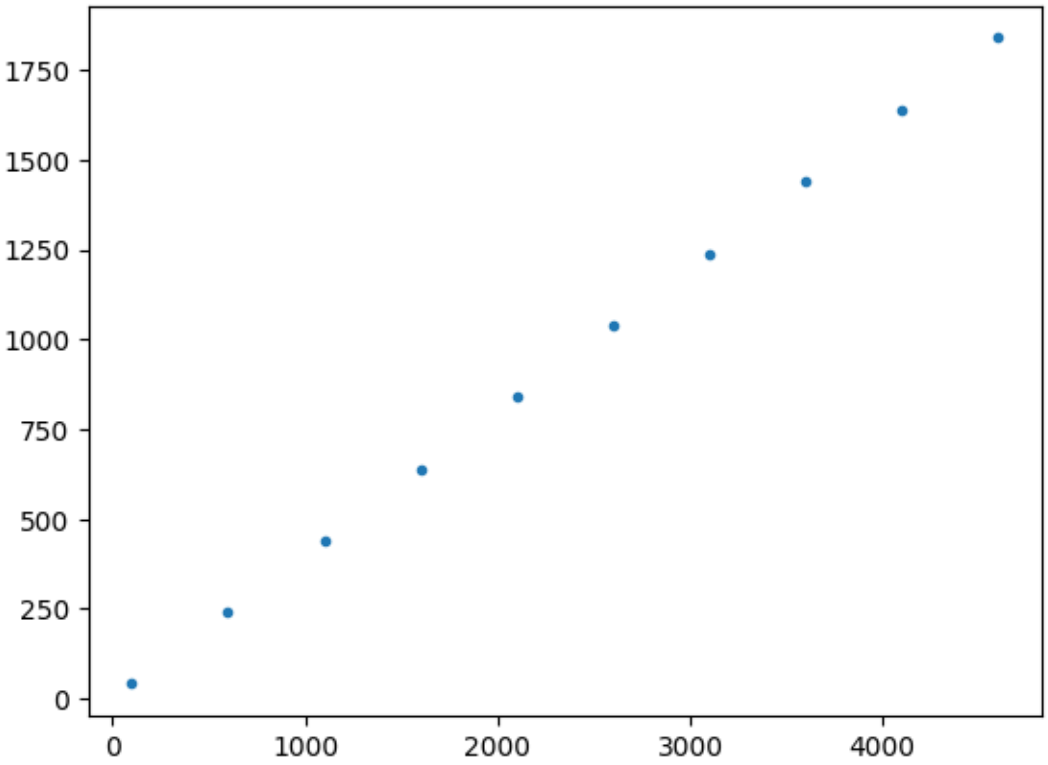
[78]:
```
sns.histplot(data=vv[vv < vv.max()], bins=50, kde=True)
```

[78]:
```
<Axes: ylabel='Count'>
```



[83]:
```
Cnt = 0
xx = np.arange(100, 5000, step=500)
vmax = []

for index in xx:
    rr = rand_canonical_f(index, Plink)
    vv = np.linalg.eigvals(rr)
    vmax.append(np.max(vv))
    Cnt += 1

plt.plot(xx, vmax, '.')
```

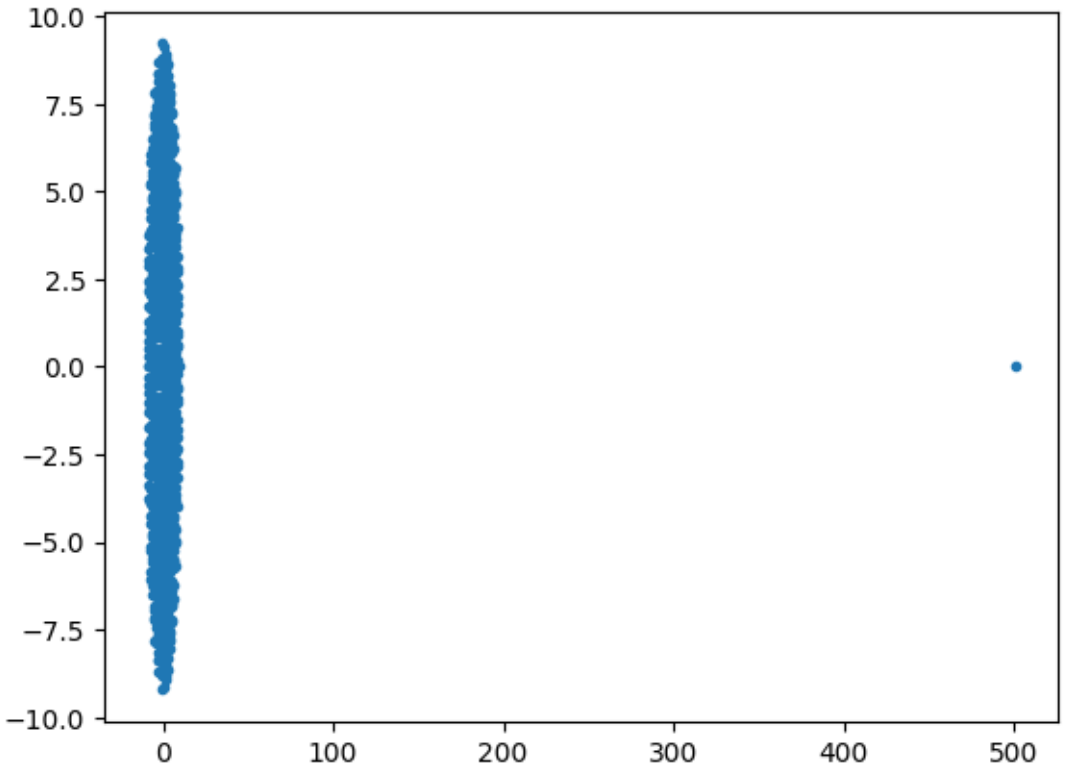
[83]: [<matplotlib.lines.Line2D at 0x7fafb8358f10>]



For a directed random matrix we'll have a complex circular spectrum

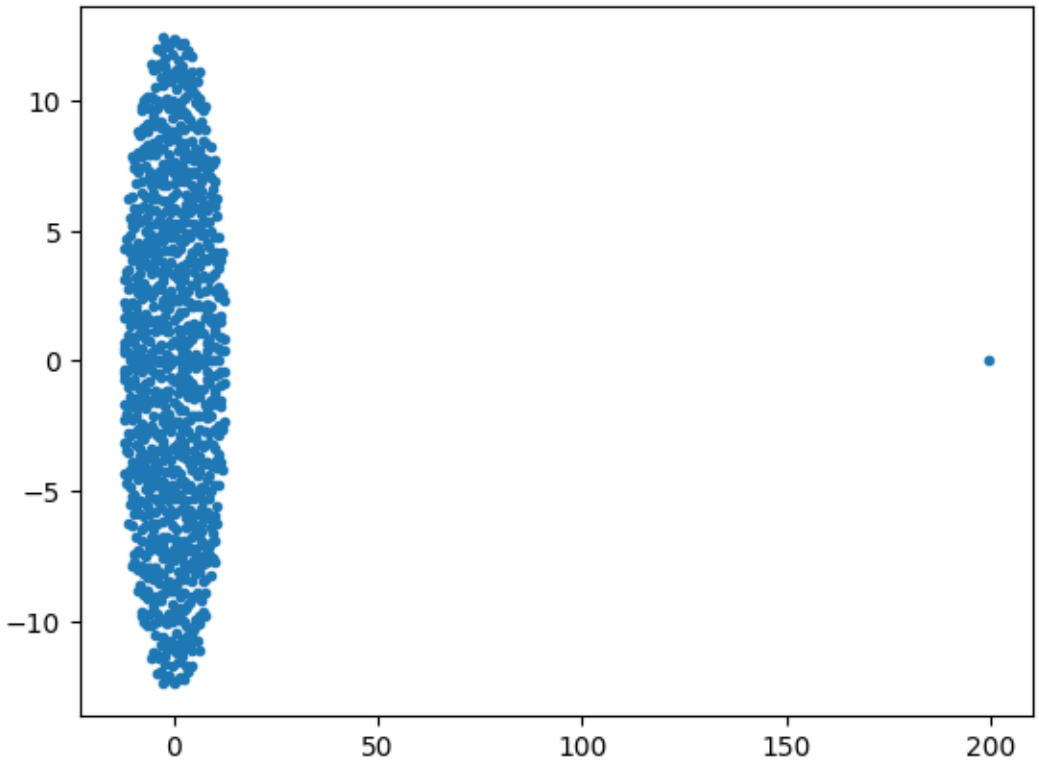
[80]:
```
rr = np.random.rand(Nnode, Nnode)
vv = np.linalg.eigvals(rr)
vvr = np.real(vv)
vvi = np.imag(vv)
plt.plot(vvr, vvi, '.')
```

[80]: [<matplotlib.lines.Line2D at 0x7fafb83ea710>]



[81]:
```
rr = np.random.rand(Nnode, Nnode) > 1 - Plink
np.fill_diagonal(rr, 0)
vv = np.linalg.eigvals(rr)
vvr = np.real(vv)
vvi = np.imag(vv)
plt.plot(vvr, vvi, '.')
```

[81]: [<matplotlib.lines.Line2D at 0x7fafb84633d0>]



[9]:
```
import scipy as sp
import numpy as np
import networkx as nx
from tqdm import tqdm # necessary because shuffling takes a while

def shuffle_net_sym_f(Net, Nsh):
    # Nsh = number of shuffles (* number of links)
    Shuf = Net
    I = list(nx.edges(Shuf))
    Nl = len(I)
    for ind in tqdm(range(Nl * Nsh)):
        L = np.random.randint(Nl, size = 2)
        while L[0] == L[1]:
            L[1] = np.random.randint(Nl)

        if Shuf.has_edge(I[L[0]][0], I[L[0]][1]) and Shuf.has_edge(I[L[1]][0],␣
↪I[L[1]][1]) and (Shuf.has_edge(I[L[0]][0], I[L[1]][1]) == False) and (Shuf.
↪has_edge(I[L[1]][0], I[L[0]][1]) == False) and (I[L[1]][0] != I[L[0]][1])␣
↪and (I[L[0]][0] != I[L[1]][1]) and (I[L[0]][0] != I[L[0]][1]) and␣
↪(I[L[1]][0] != I[L[1]][1]):
            # delete links
            Shuf.remove_edge(I[L[0]][0], I[L[0]][1])
```

```
            Shuf.remove_edge(I[L[1]][0], I[L[1]][1])
            if Shuf.has_edge(I[L[0]][1], I[L[0]][0]):
                Shuf.remove_edge(I[L[0]][1], I[L[0]][0])
            if Shuf.has_edge(I[L[1]][1], I[L[1]][0]):
                Shuf.remove_edge(I[L[1]][1], I[L[1]][0])
            # shuffle links
            Shuf.add_edge(I[L[0]][0], I[L[1]][1])
            Shuf.add_edge(I[L[1]][1], I[L[0]][0])
            Shuf.add_edge(I[L[1]][0], I[L[0]][1])
            Shuf.add_edge(I[L[0]][1], I[L[1]][0])
            # remap links
            I = list(nx.edges(Shuf))
            Nl = len(I)
    return Shuf
```

[10]:
```
data = sp.io.loadmat('Yeast_DIP.mat')
YSTnet = data['YSTnet']
```

Let's process the network, which is not symmetrized with loops

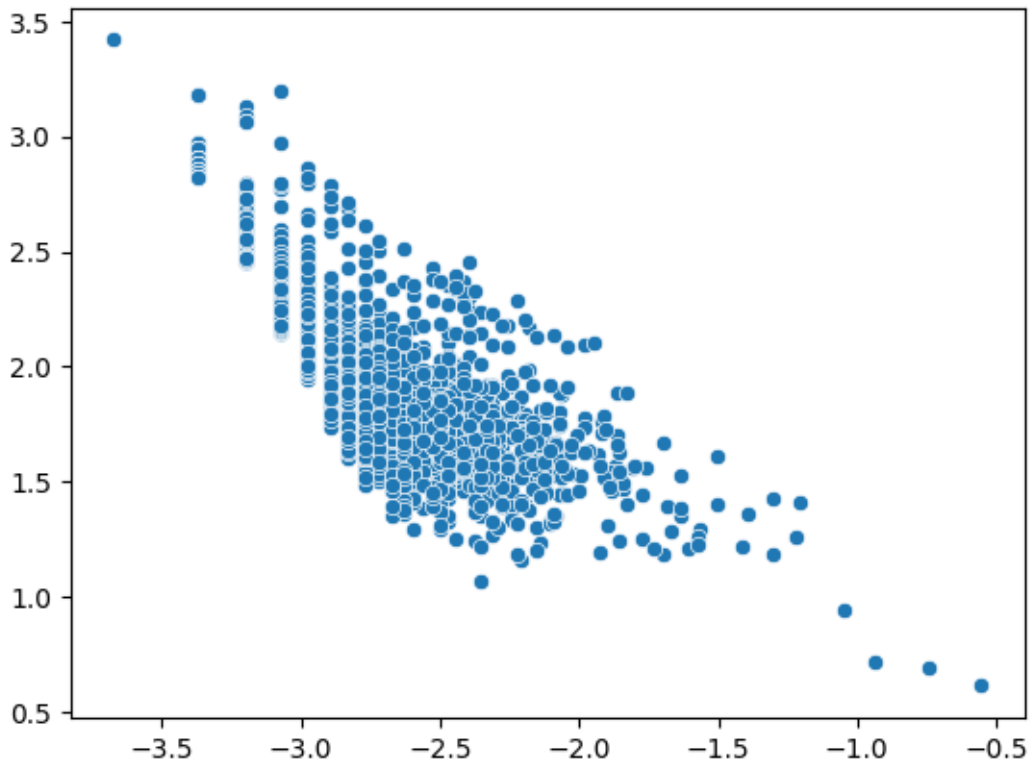
[11]:
```
Ndir = np.sum(YSTnet - YSTnet.T)
Nloop = YSTnet.trace()

print(Ndir)
print(Nloop)

YSTnet.setdiag(0)
YSTnetSym = YSTnet + YSTnet.T

GraphYST = nx.from_scipy_sparse_array(YSTnetSym)
```

```
0.0
4713.0
```

[12]:
```
Kyst = nx.degree_centrality(GraphYST)
Kyst = np.array(list(Kyst.values()))
Knn = np.divide(YSTnetSym * Kyst, Kyst)
```

[13]:
```
import seaborn as sns

sns.scatterplot(x=np.log10(Kyst), y=np.log10(Knn))
```

[13]: <Axes: >



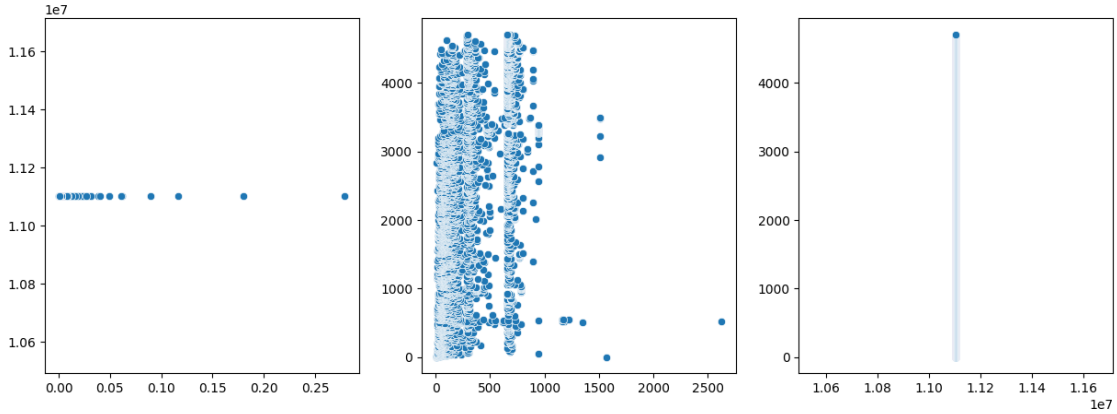
Let's shuffle links - microcanonical ensemble. **NOTE**: microcanonical shuffle does not work with few iterations.

[14]:
```
YSTshuf = shuffle_net_sym_f(GraphYST, 50) # maybe 50 is not enought but it␣
↪takes about 2 hours and half to run
```

```
100%|      | 976400/976400 [1:08:31<00:00, 237.49it/s]
```

[15]:
```
YSTshufK = np.sum(YSTshuf, axis = 0)
KnnS = np.divide(YSTshuf * YSTshufK, YSTshufK)

from matplotlib import pyplot as plt

fig, ax = plt.subplots(ncols=3, figsize=(15, 5))
sns.scatterplot(x=Kyst, y=YSTshufK, ax=ax[0])
sns.scatterplot(x=Knn, y=KnnS, ax=ax[1])
sns.scatterplot(x=YSTshufK, y=KnnS, ax=ax[2])
```

[15]:
```
<Axes: >
```


1e7
1.16
1.14
1.12
1.10
1.08
1.06
0.00
0.05
0.10
0.15
0.20
0.25
4000
3000
2000
1000
0
0
500
1000
1500
2000
2500
4000
3000
2000
1000
0
1.06
1.08
1.10
1.12
1.14
1.16
1e7