PY541 PS1

Raymond Kil

September 20, 2022

Problem 1 (Sethna 1.6: Random Matrix Theory)

(a)

Attached below as a pdf version of Jupyter notebook.

(b)

The eigenvalues of the matrix is given by:

$$\det \begin{pmatrix} a & b \\ b & c \end{pmatrix} = \det \begin{pmatrix} a-\lambda & b \\ b & c-\lambda \end{pmatrix} = (a-\lambda)(c-\lambda) - b^2 = 0 \Rightarrow \lambda_{\pm} = \frac{(a+c) \pm \sqrt{(c-a)^2 + 4b^2}}{2} \tag{1}$$

The eigenvalue difference is thus just two times one of the eigenvalues:

$$\Delta_\lambda = \sqrt{(c-a)^2 + 4b^2} = 2\sqrt{d^2 + b^2} \text{ , where } d = \frac{c-a}{2} \tag{2}$$

In (b, d) plane, the region of eigenvalue splitting falls within the following region:

$$\lambda \leq 2\sqrt{d^2 + b^2} \leq \lambda + \Delta_\lambda \Rightarrow \frac{\lambda^2}{4} \leq d^2 + b^2 \leq \frac{(\lambda + \Delta_\lambda)^2}{4} \tag{3}$$

The region of eigenvalue splitting has an area in (b, d) plane of

$$\pi \left(\frac{\lambda + \Delta_\lambda}{2}\right)^2 - \pi \left(\frac{\lambda}{2}\right)^2 = \pi \frac{\lambda^2 + 2\lambda\Delta + \Delta^2 - \lambda^2}{4} \approx \frac{\pi\lambda\Delta_\lambda}{2} \sim \lambda \tag{4}$$

where we considered Δ_λ to be small. Given that ρ_M is finite at $d = b = 0$, we can see from the result of Eq. 4 that the probability density $\rho(\lambda)$ of finding an eigenvalue splitting near $\lambda = 0$ vanishes. This is a manifestation of level repulsion.

(c)

Let's consider the following 2×2 matrix and a GOE matrix produced by adding the matrix to its transpose:

$$M = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \Rightarrow M_{GOE} = \begin{pmatrix} 2a & b+c \\ c+b & 2d \end{pmatrix} \tag{5}$$

Note that each entry of M has standard deviation of $\sigma_{a,d} = 1$. Thus, the diagonal elements will have standard deviation of $\sigma_{2a,2d} = 2\sigma_{a,d} = 2$. On the other hand, the standard deviation of off diagonal elements are added in quadrature. That is, $\sigma_{b+c} = \sqrt{\sigma_b^2 + \sigma_c^2} = \sqrt{1+1} = \sqrt{2}$. The fact that the standard deviation of diagonal elements are doubled and that of off-diagonal elements are added in quadrature can be generalized to matrices of size $N > 2$.

(d)

Probability distribution of eigenvalue spacing for $N = 2$ GOE is given by integrating over $\rho_M(d, b)$:

$$\rho(\lambda) = \int \rho_M(d,b)\delta(\lambda^2 - 4d^2 - 4b^2)dd\ db \tag{6}$$

The general expression for a bivariate Gaussian is given by:

$$f(x,y) = \frac{1}{2\pi\sigma_x\sigma_y\sqrt{1-\rho}} \exp\left[-\frac{1}{2(1-\rho^2)}\left[\left(\frac{x-\mu_x}{\sigma_x}\right)^2 - 2\rho\left(\frac{x-\mu_x}{\sigma_x}\frac{y-\mu_y}{\sigma_y}\right) + \left(\frac{y-\mu_y}{\sigma_y}\right)^2\right]\right] \tag{7}$$

where we have $\mu_{x,y} = \mu_{b,d} = 0$, $\sigma_{x,y} = \sigma_{b,d} = \sqrt{2}$, and $\rho = 0$. Eq. 7 reduces to:

$$\rho(b,d) = \frac{1}{4\pi} e^{-\frac{1}{4}(b^2+d^2)} \tag{8}$$

Using polar coordinates, we can substitute $r^2 = b^2 + d^2$ and $dd\ db = rdrd\theta$. Also, $\delta(\lambda^2 - 4d^2 - 4b^2) = \delta(\lambda^2 - 4r^2) = \frac{1}{|4|}\delta(\frac{\lambda^2}{4} - r^2)$. Then, the integral in Eq. 6 becomes:

$$\begin{aligned} \rho(\lambda) &= \int \rho_M(d,b)\delta(\lambda^2 - 4d^2 - 4b^2)dd\ db & (9)\\ &= \frac{1}{4\pi}\frac{1}{4}\int e^{-\frac{r^2}{4}}\delta\left(\frac{\lambda^2}{4} - r^2\right) rdrd\theta = \frac{\lambda}{8}e^{-\frac{\lambda^2}{16}} & (10) \end{aligned}$$

(e)

Attached below as a pdf version of Jupyter notebook.

(f)

Attached below as a pdf version of Jupyter notebook.

(g)

The trace can be expressed in terms of summation:

$$\mathrm{Tr}[H^T H] = \sum_i H_{ii}^T H_{ii} \tag{11}$$

We can also express the product of any two matrices as a sum:

$$H^T H = \sum_j H_{ij}^T H_{ji} \tag{12}$$

Combining Eq. 11 and 12 gives:

$$\mathrm{Tr}[H^T H] = \sum_i \sum_j H_{ij}^T H_{ji} = \sum_{ij} H_{ji} H_{ji} = \sum_{ij} (H_{ji})^2 \tag{13}$$

where we noted that the transpose of a matrix is equivalent to swapping the indices of a matrix. Thus, we showed that $\mathrm{Tr}[H^T H]$ is the sum of the squares of all elements of H. To show the invariance under orthogonal coordinate transformation, we can make a substitution $H \to R^T H R$ and $H^T \to \left(R^T H R\right)^T$, plug into Eq. 11, and see that the expression does not change.

$$\begin{aligned} \mathrm{Tr}[H^T H] &\Rightarrow \mathrm{Tr}\left[\left(R^T H R\right)\left(R^T H R\right)^T\right] = \mathrm{Tr}\left[R^T H^T R R^T H R\right] & (14)\\ &= \mathrm{Tr}\left[R^T H^T H R\right] = \mathrm{Tr}\left[R R^T H^T H\right] = \mathrm{Tr}\left[H^T H\right] \end{aligned}$$

where in the penultimate equality we used the cyclic invariance of the trace $\mathrm{Tr}[ABC] = \mathrm{Tr}[CAB]$. Therefore, the trace is invariant under orthogonal coordinate transformation.

(h)

Probability density $\rho(H)$ for finding GOE ensemble member H is equal to the probability density that each element in H matches with what we are finding. Such probability density can be expressed as a product of probabilities of each element having certain value. Note that for a $N \times N$ matrix, there are $\frac{n(n-1)}{2}$ off-diagonal elements and N diagonal elements. Thus, for a symmetric matrix, there are $\frac{N(N+1)}{2}$ independent elements. That is, for H_{ij}, only the values where $i \leq j$ have to match.

$$\rho(H) = \prod_{i \leq j} \rho(H_{ij}) = \left(\prod_i \rho(H_{ii}) \right) \left(\prod_{i<j} \rho(H_{ij}) \right) \tag{15}$$

Now we substitute an appropriate Gaussian for ρ:

$$\begin{aligned} \rho(H) &= \prod_i \frac{1}{\sqrt{8\pi}} e^{-\frac{H_{ii}^2}{8}} \times \sqrt{\prod_{i \neq j} \frac{1}{2\sqrt{\pi}} e^{-\frac{H_{ij}^2}{4}}} = \left(\frac{1}{\sqrt{8\pi}} \right)^N \left(\frac{1}{2\sqrt{\pi}} \right)^{\frac{N(N-1)}{2}} \prod_{i,j} e^{-\frac{H_{ij}^2}{8}} \\ &= \left(\frac{1}{2} \right)^{\frac{N(N+2)}{2}} \left(\frac{1}{\sqrt{\pi}} \right)^{\frac{N(N+1)}{2}} \exp\left[-\frac{1}{8} \mathrm{Tr}[H^T H] \right] \end{aligned} \tag{16}$$

where in the last equality, we substituted the product by the trace we derived at Eq. 13. We know from the result above that GOE ensemble is invariant under orthogonal transformation, since the probability distribution only depends on the trace, which is itself invariant under orthogonal transformation.

Problem 2 (Sethna 1.13: The Birthday Problem)

Attached below as a pdf version of Jupyter notebook.

Problem 3 (Sethna 1.15: Fisher Information)

(a)

$$P(x|\theta) = \frac{1}{(2\pi\sigma^2)^{N/2}} \exp\left[\frac{-\sum_i (y_i(\theta) - x_i)^2}{2\sigma^2} \right] \tag{17}$$

We are interested in the probability density that a least-squares model with true parameter θ gives a wrong prediction with false parameter ϕ. That is, we are interested in $P\big(y(\phi)|\theta\big)$. Plugging the expression into Eq. 17 gives:

$$P(x|\theta) \sim \exp\left[\frac{-(\vec{y}(\vec{\theta}) - \vec{y}(\vec{\phi}))^2}{2\sigma^2} \right] \tag{18}$$

We can notice in the argument for the exponential that the probability density only depends on the distance between the vectors $|\vec{y}(\vec{\theta}) - \vec{y}(\vec{\phi})|$.

(b)

The metric tensor $g_{\alpha\beta}$ gives the distance on the manifold between two nearby points $\vec{\theta}$ and $\vec{\theta} + \epsilon\vec{\Delta}$:

$$\epsilon^2 \sum_{\alpha\beta} g_{\alpha\beta} \Delta_\alpha \Delta_\beta \tag{19}$$

The general expression for the squared distance is given by:

$$\sum \frac{(y_i(\theta) - y_i(\phi))^2}{2\sigma^2} \text{ , where } \phi = \theta + \epsilon\Delta \tag{20}$$

Taylor expanding Eq. 20 to first order gives:

$$y_i^2(\phi) = y_i^2(\theta + \epsilon\Delta) \approx y_i(\theta) + \frac{\partial y_i}{\partial \theta_\alpha} \epsilon \Delta_\alpha \tag{21}$$

Then Eq. 20 gives:

$$\sum \frac{(y_i(\theta) - y_i(\theta) + \frac{\partial y_i}{\partial \theta_\alpha}\epsilon\Delta_\alpha)^2}{2\sigma^2} = \sum \frac{(\frac{\partial y_i}{\partial \theta_\alpha}\epsilon\Delta_\alpha)^2}{2\sigma^2} = \frac{1}{2\sigma^2}\sum \frac{\partial y_i}{\partial \theta_\alpha}\frac{\partial y_i}{\partial \theta_\beta}\epsilon^2\Delta_\alpha\Delta_\beta \tag{22}$$

Comparing the results from Eq. 22 to Eq. 22, we get the expression for the metric:

$$g_{\alpha\beta} = \frac{1}{\sigma^2}\frac{\partial y_i}{\partial \theta_\alpha}\frac{\partial y_i}{\partial \theta_\beta} = \frac{(J^T J)_{\alpha\beta}}{\sigma^2} \tag{23}$$

(c)

We are to show that for least-squares model,

$$g_{\alpha\beta}(\theta) = -\left\langle \frac{\partial^2 \log P(x|\theta)}{\partial\theta_\alpha \partial\theta_\beta} \right\rangle_{\vec{x}} = \frac{1}{\sigma^2}\left(J^T J\right)_{\alpha\beta} \tag{24}$$

Recall Eq. 17. Taking the log of it gives:

$$\log P(x|\theta) = -\frac{1}{2\sigma^2}\sum_i \left(y_i(\theta) - x_i\right)^2 = -\frac{1}{2\sigma^2}\sum_i \left(y_i^2(\theta) - 2y_i(\theta)x_i + x_i^2\right) \tag{25}$$

Plugging in Eq. 25 to 24 gives:

$$\begin{aligned}
-\left\langle \frac{\partial^2 \log P(x|\theta)}{\partial\theta_\alpha \partial\theta_\beta} \right\rangle_{\vec{x}} &= -\left\langle \frac{1}{2\sigma^2}\sum_i \frac{\partial^2}{\partial\theta_\alpha\partial\theta_\beta}\left(y_i^2(\theta) - 2y_i(\theta)x_i + x_i^2\right) \right\rangle_{\vec{x}} \qquad (26)\\
&= -\left\langle \frac{1}{2\sigma^2}\sum_i \frac{\partial}{\partial\theta_\alpha}\left[\frac{\partial}{\partial\theta_\beta}\left(y_i^2(\theta)\right) - 2\frac{\partial}{\partial\theta_\beta}\left(y_i(\theta)x_i\right) + \frac{\partial}{\partial\theta_\beta}\left(x_i^2\right)\right] \right\rangle_{\vec{x}}\\
&= -\left\langle \frac{1}{2\sigma^2}\sum_i \frac{\partial}{\partial\theta_\alpha}\left[2y_i(\theta)\frac{\partial y_i(\theta)}{\partial\theta_\beta} - 2x_i\frac{\partial y_i(\theta)}{\partial\theta_\beta}\right] \right\rangle_{\vec{x}}\\
&= -\left\langle \frac{1}{2\sigma^2}\sum_i \left[\frac{\partial y_i(\theta)}{\partial\theta_\alpha}\frac{\partial y_i(\theta)}{\partial\theta_\beta} + y_i(\theta)\frac{\partial^2 y_i(\theta)}{\partial\theta_\alpha\partial\theta_\beta} - 2x_i\frac{\partial^2 y_i(\theta)}{\partial\theta_\alpha\partial\theta_\beta}\right] \right\rangle_{\vec{x}}\\
&= -\left\langle \frac{1}{\sigma^2}\sum_i \frac{\partial y_i(\theta)}{\partial\theta_\alpha}\frac{\partial y_i(\theta)}{\partial\theta_\beta} \right\rangle_{\vec{x}} = -\frac{1}{\sigma^2}\left\langle \sum_i J_{i\alpha}J_{i\beta} \right\rangle_{\vec{x}}\\
&= -\frac{1}{\sigma^2}\left\langle \sum_i J^T_{\alpha i}J_{i\beta} \right\rangle_{\vec{x}} = \frac{1}{\sigma^2}(J^T J)_{\alpha\beta} \qquad (27)
\end{aligned}$$

Therefore we showed that Eq. 24 is true.

(d)

Taking the Taylor expansion of $\log P(\theta + \epsilon\Delta)$ to the second order in ϵ gives:

$$\begin{aligned}
\log\left(\frac{P(\theta+\epsilon\Delta)}{P(\theta)}\right) &\approx \log P(\theta) + \frac{\partial^2}{\partial\theta_\alpha\partial\theta_\beta}\log P(\theta)\epsilon^2\Delta_\alpha\Delta_\beta - \log(P(\theta)) \qquad (28)\\
&= \frac{\partial^2}{\partial\theta_\alpha\partial\theta_\beta}\log P(\theta)\epsilon^2\Delta_\alpha\Delta_\beta \qquad (29)
\end{aligned}$$

Where we expanded the fraction between $P(\theta+\epsilon\Delta)$ and $P(\theta)$ in order to see how much the probability of measuring values corresponding to the predictions at $\theta+\epsilon\Delta$ fall off compared to $P(\theta)$. Exponentiating the expression at Eq. 29 gives:

$$\frac{P(\theta+\epsilon\Delta)}{P(\theta)} = \exp\left[\frac{\partial^2}{\partial\theta_\alpha\partial\theta_\beta}\log P(\theta)\epsilon^2\Delta_\alpha\Delta_\beta\right] = \exp\left[g_{\alpha\beta}(\theta)\epsilon^2\Delta_\alpha\Delta_\beta\right] \tag{30}$$

Eq. 30 is a Gaussian with the center at $\epsilon = 0$ and with standard deviation of $\sigma = \frac{1}{\sqrt{2}g_{\alpha\beta}(\theta)}$. Therefore, to linear order, the FIM $g_{\alpha\beta}$ gives the range of likely measured parameters around the true parameters of the model.

Problem 4 (Sethna 2.21: Levy Flight)

(a)

The Fourier transform of $S_\alpha(x)$ is given by:

$$\tilde{S_\alpha}(k) = \int_{-\infty}^{\infty} e^{-ikx} S_\alpha(x) dx = e^{-|k|^\alpha} \tag{31}$$

When we set $k = 0$, then we recover the normalization condition:

$$\int_{-\infty}^{\infty} \exp^{-ikx} S_\alpha(x) dx_{|k=0} = \int_{-\infty}^{\infty} 1 \times S_\alpha(x) dx = e^{-|0|^\alpha} = 1 \tag{32}$$

Therefore, $S_\alpha(x)$ is normalized.

(b)

The distribution $P(z)$ is given by:

$$P(z) = \int S_\alpha(x) S_\alpha(x) dx = \int S_\alpha(x) S_\alpha(z-x) dx \text{ , where } z = x + y \tag{33}$$

Taking the Fourier transform, we can write $P(z)$ as a product of $S_\alpha(x)$ and $S_\alpha(y)$:

$$\tilde{P}(z) = \tilde{S_\alpha}(k) \tilde{S_\alpha}(k) = e^{-2|k|^\alpha} \tag{34}$$

We can notice that since $P(z)$ has an extra scaling factor at the exponent, its general shape will be same as that of $P(x)$ and $P(y)$, but just rescaled by the scaling factor λ_α. At the instance of Eq. 34, $\lambda_\alpha = 2^{1/\alpha}$. For $\alpha = 2$, we can see that $\lambda_2 = 2^{1/2} = \sqrt{2}$.

(c)

Every time we repeat part (b), we get an extra factor of λ_α. Therefore, if we repeat m times, we will have m scaling factors. That is, $\lambda_\alpha^m = \left(2^{1/\alpha}\right)^m$. Expressing m in terms of N, we can see at what power the distribution vary with N:

$$m = \log_2 N \Rightarrow \left(2^{\log_2 N}\right)^{1/\alpha} = N^{1/\alpha} \tag{35}$$

PY541_PS1

September 21, 2022

1 PY541 Problem Set 1

1.1 Raymond Kil

1.1.1 September 20, 2022

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import math
```

1.1.2 Problem 1: Random Matrix Theory (Sethna 1.6)

(a) First, I will be making $M = 1000$ Gaussian Orthogonal Ensemble (GOE) matrices.

[2]:
```
# The following function creates a GOE matrix with a given size.

def GOEmatrixGenerator(size):
    matrix = np.zeros((size, size))
    for row in range(size):
        for col in range(size):
            if col > row:
                matrix[row][col] = np.random.normal(0,1)
    matrixT = np.matrix.transpose(matrix)
    matrix = matrix + matrixT
    for diag in range(size):
        matrix[diag][diag]=np.random.normal(0,1)
    return matrix
```

[3]:
```
# Here I am creating 1000 GOE matrices of size 2x2, another 1000 of size 4x4,␣
↪and another 1000 of size 10x10.

M     = 1000
GOE2  = [0] * M
GOE4  = [0] * M
GOE10 = [0] * M

for i in range(M):
    GOE2[i]  = GOEmatrixGenerator(2)
```

```
    GOE4[i]  = GOEmatrixGenerator(4)
    GOE10[i] = GOEmatrixGenerator(10)
```

[4]:
```
# Now I am calculating the eigenvalues of the matrices, sroted to increasing␣
↪order

eigenvalues2  = np.sort(np.real(np.linalg.eigvals(GOE2)))
eigenvalues4  = np.sort(np.real(np.linalg.eigvals(GOE4)))
eigenvalues10 = np.sort(np.real(np.linalg.eigvals(GOE10)))
```

[5]:
```
# I am finding the split in neighboring eigenvalues.

split2  = np.zeros((M,1))
split4  = np.zeros((M,3))
split10 = np.zeros((M,9))

for i in range(M):
    for j in range(2-1):
        split2[i][j]  = eigenvalues2[i][j+1] - eigenvalues2[i][j]
    for j in range(4-1):
        split4[i][j]  = eigenvalues4[i][j+1] - eigenvalues4[i][j]
    for j in range(10-1):
        split10[i][j] = eigenvalues10[i][j+1] - eigenvalues10[i][j]


# Now the mean split.
mean_split2  = np.mean(np.ndarray.flatten(split2))
mean_split4  = np.mean(np.ndarray.flatten(split4))
mean_split10 = np.mean(np.ndarray.flatten(split10))

print("A GOE matrix: \n", GOE4[0],"\n")
print("Eigenvalues: \n", eigenvalues4[0],"\n")
print("Split between eigenvalues: \n", split4[0],"\n")
print("Mean splits are:", mean_split2,mean_split4,mean_split10)
```

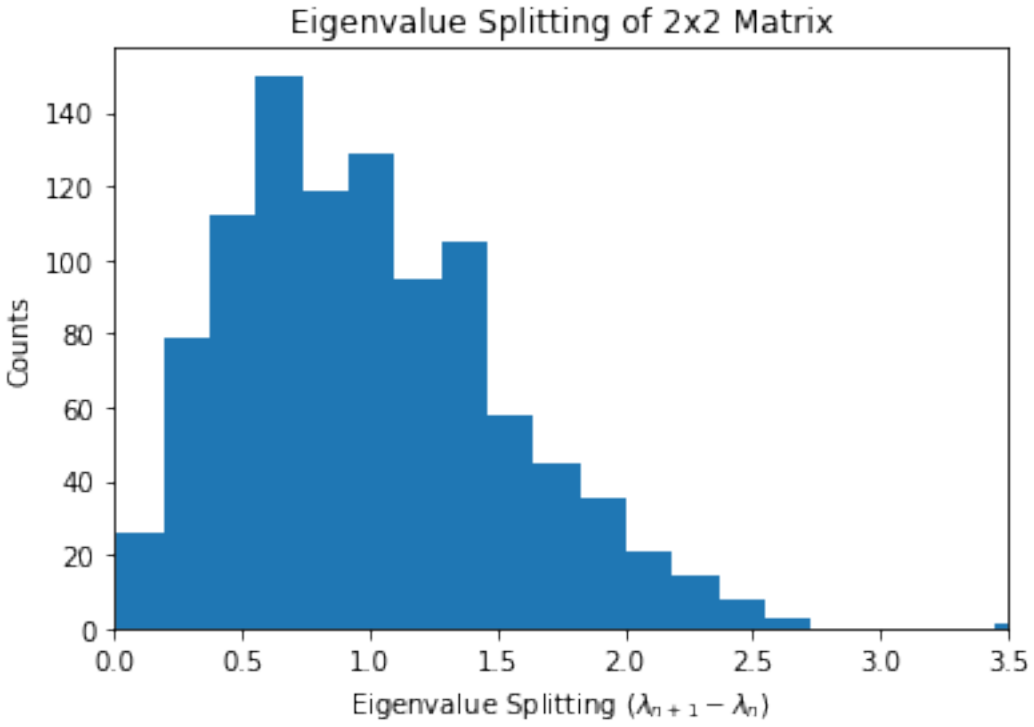
```
A GOE matrix:
 [[-0.86333527 -0.93542869  0.13370433 -0.30875281]
 [-0.93542869  1.19205748 -0.64750338  0.30437778]
 [ 0.13370433 -0.64750338  0.2179386   1.05256659]
 [-0.30875281  0.30437778  1.05256659  0.34373643]]

Eigenvalues:
 [-1.24720506 -0.98751342  1.30352776  1.82158795]

Split between eigenvalues:
 [0.25969165 2.29104118 0.51806019]
```

```
Mean splits are: 2.2480846947341475 1.6681005622628278 1.135006798075654
```

[6]:
```
# We are interested in the splitting of two MIDDLE eigenvalues!
# Let's plot a histogram of the eigenvalue splitting divided by the mean␣
↪splitting.

eigval_split2  = np.zeros(M) #This split will be the split divided by the mean␣
↪splitting.
eigval_split4  = np.zeros(M)
eigval_split10 = np.zeros(M)

for i in range(M):
    eigval_split2[i]  = (split2[i]/mean_split2)[0]
    eigval_split4[i]  = (split4[i]/mean_split4)[1]
    eigval_split10[i] = (split10[i]/mean_split10)[4] #index 4 means the middle␣
↪(N/2)th difference
```

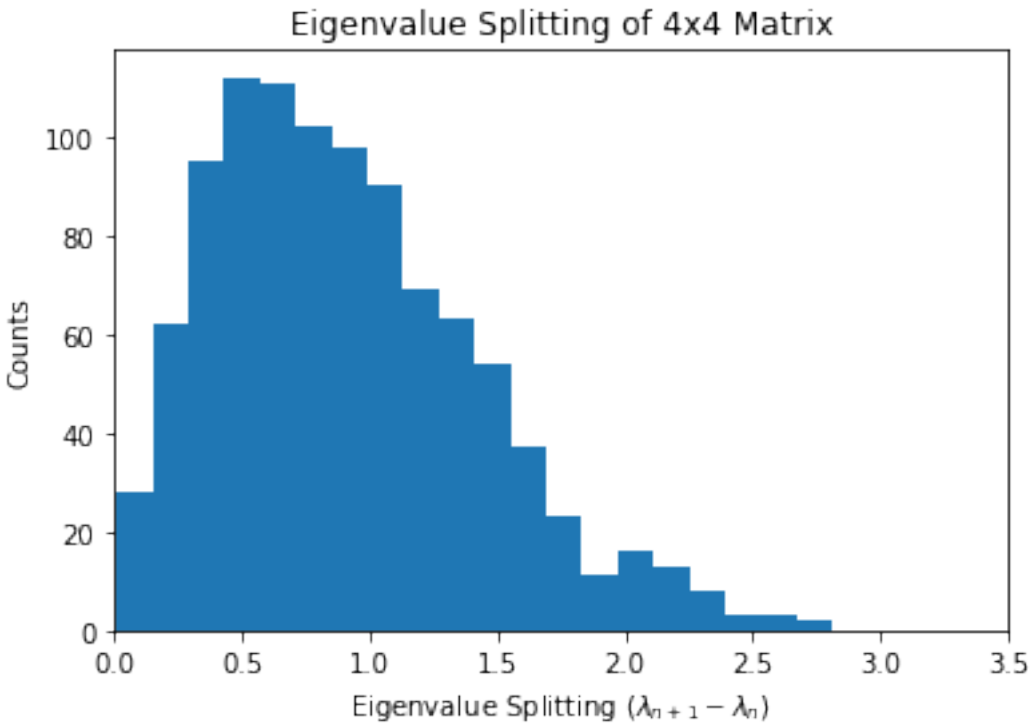
[7]:
```
plt.figure()
plt.hist(eigval_split2,bins=20)
plt.title("Eigenvalue Splitting of 2x2 Matrix")
plt.xlabel("Eigenvalue Splitting ($\lambda_{n+1}-\lambda_{n}$)")
plt.ylabel("Counts")
plt.xlim((0,3.5))
```

[7]:
```
(0.0, 3.5)
```



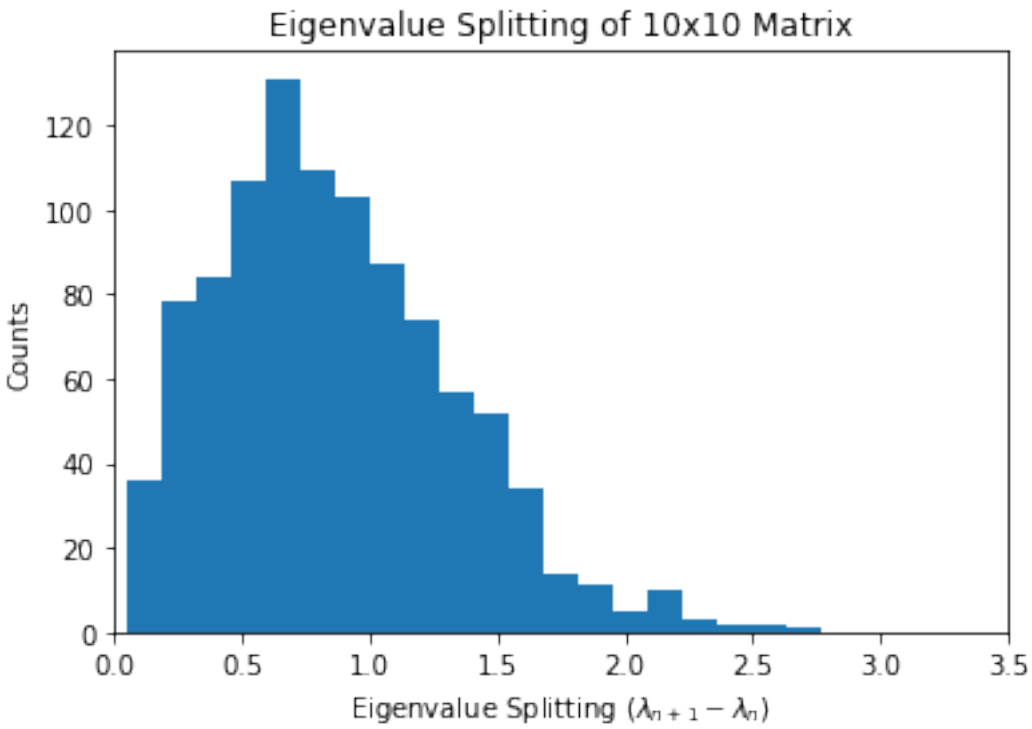
[8]:
```
plt.figure()
plt.hist(eigval_split4,bins=20)
plt.title("Eigenvalue Splitting of 4x4 Matrix")
plt.xlabel("Eigenvalue Splitting ($\lambda_{n+1}-\lambda_{n}$)")
plt.ylabel("Counts")
plt.xlim((0,3.5))
```

[8]: (0.0, 3.5)



[9]:
```
plt.figure()
plt.hist(eigval_split10,bins=20)
plt.title("Eigenvalue Splitting of 10x10 Matrix")
plt.xlabel("Eigenvalue Splitting ($\lambda_{n+1}-\lambda_{n}$)")
plt.ylabel("Counts")
plt.xlim((0,3.5))
```

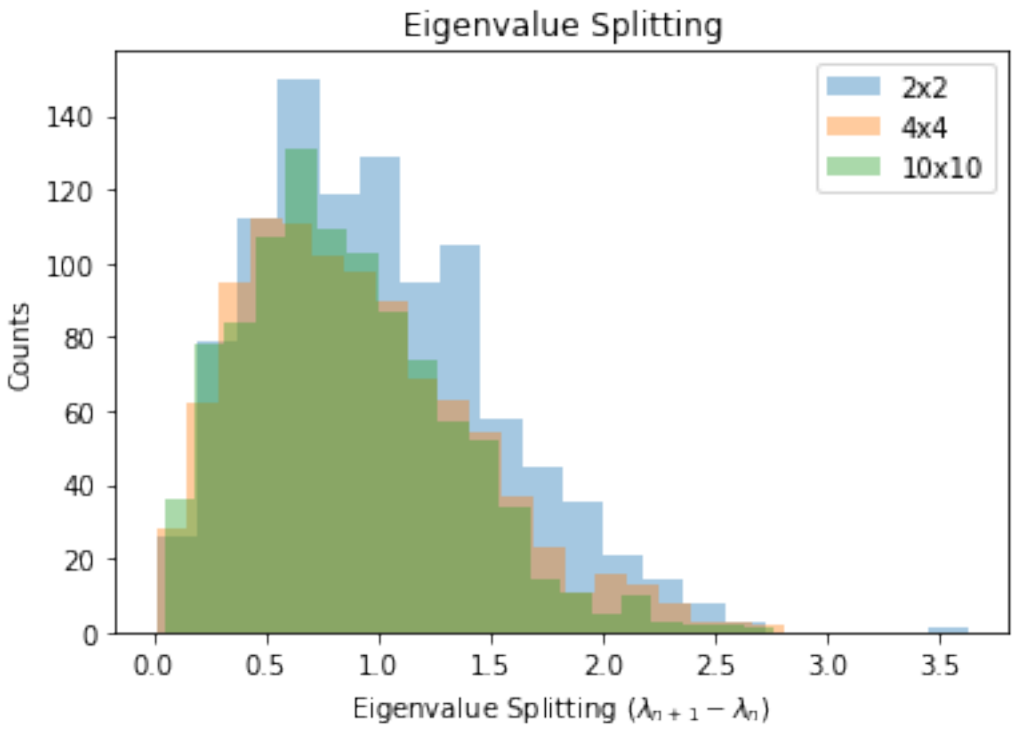
[9]: (0.0, 3.5)



Putting into perspective, let's try plotting the three histograms in one figure.

[10]:
```
plt.hist(eigval_split2,bins=20,alpha=0.4) #Notice that I used split2 instead of␣
↪meansplit, b/c it returns 1.
plt.hist(eigval_split4,bins=20,alpha=0.4)
plt.hist(eigval_split10,bins=20,alpha=0.4)
plt.title("Eigenvalue Splitting")
plt.xlabel("Eigenvalue Splitting ($\lambda_{n+1}-\lambda_{n}$)")
plt.ylabel("Counts")
plt.legend(["2x2","4x4","10x10"])
```

[10]:
```
<matplotlib.legend.Legend at 0x7f098ef36e10>
```



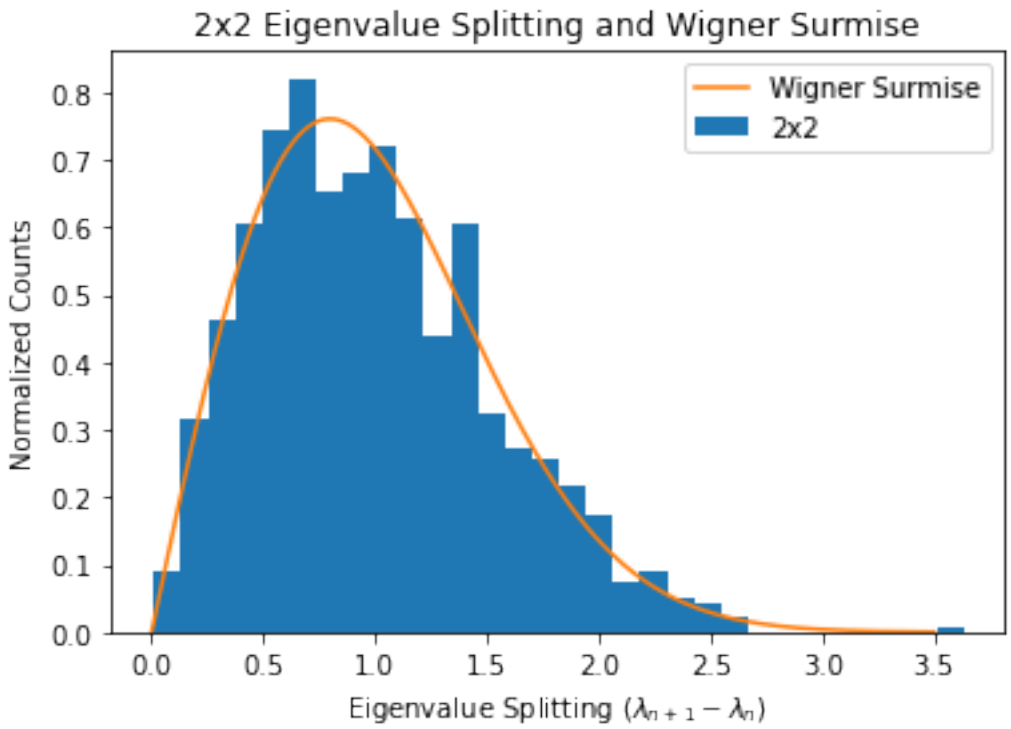
(e) I am plotting the Wigner's surmise along with $N = 2, 4, 10$ results from part (a).

[11]:
```
def WignerSurmise(S):
    rho = (np.pi*S)/2*np.exp((-np.pi*S**2)/4)
    return rho
```

[12]:
```
s = np.linspace(0,3.5,1000)
wigner = np.zeros(len(s))

for i in range(len(s)):
    wigner[i] = WignerSurmise(s[i])
```

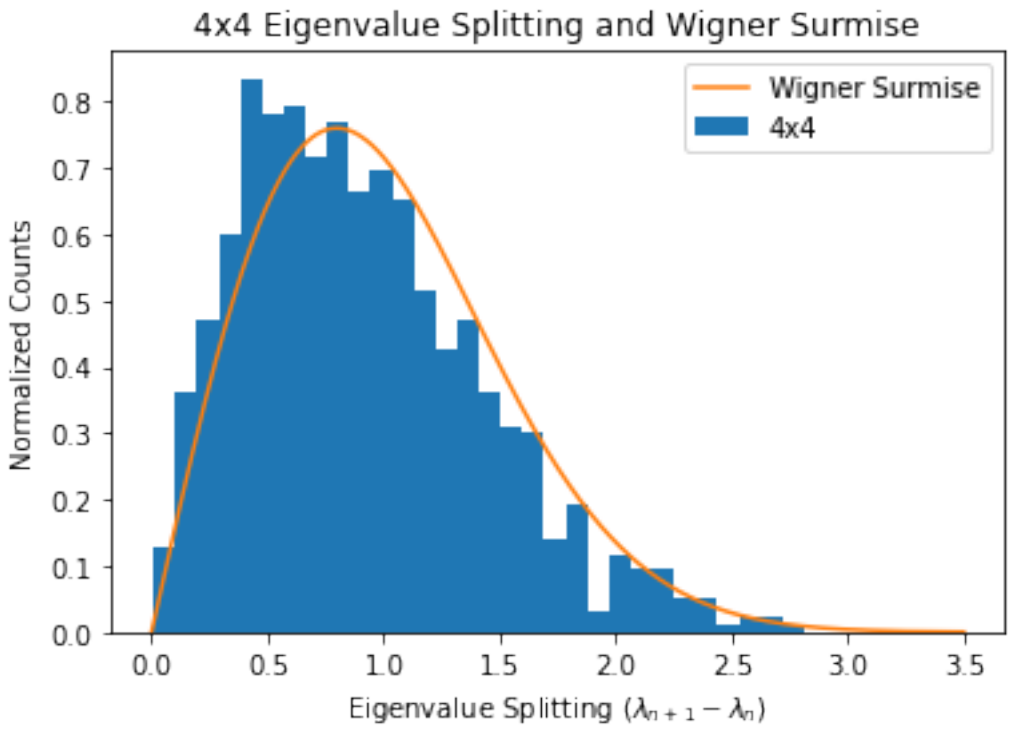
[13]:
```
plt.figure()
plt.hist(eigval_split2,bins=30,density=True)
plt.plot(s,wigner)
plt.title("2x2 Eigenvalue Splitting and Wigner Surmise")
plt.xlabel("Eigenvalue Splitting ($\lambda_{n+1}-\lambda_{n}$)")
plt.ylabel("Normalized Counts")
plt.legend(["Wigner Surmise","2x2"])
```

[13]:
```
<matplotlib.legend.Legend at 0x7f098edfe510>
```



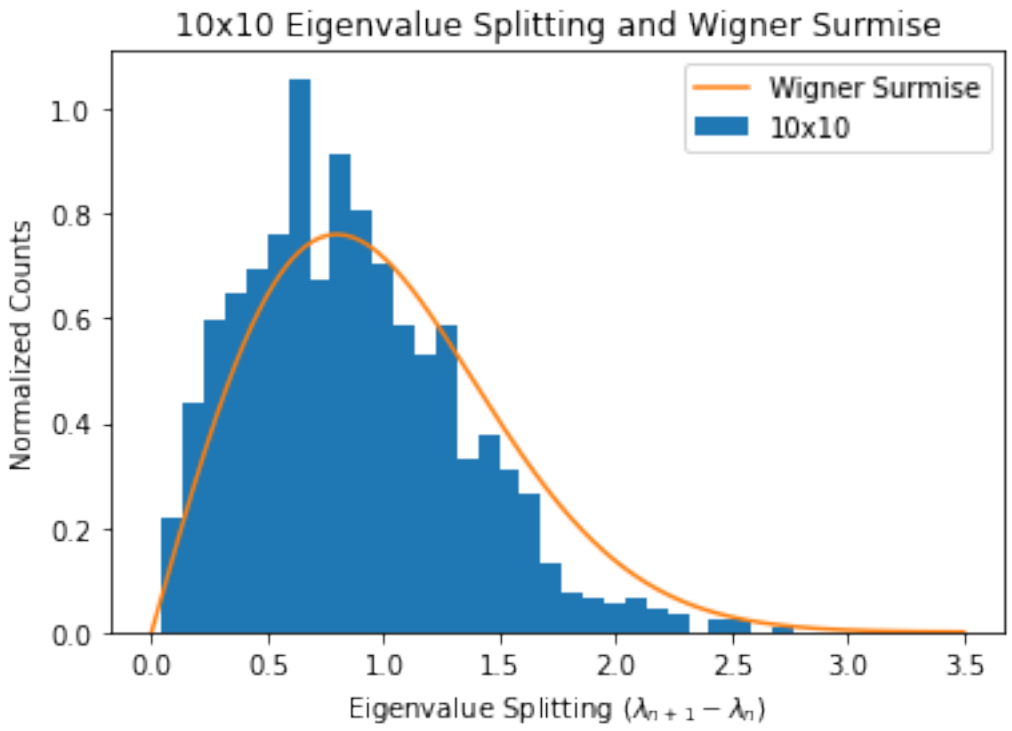
[14]:
```
plt.figure()
plt.hist(eigval_split4,bins=30,density=True)
plt.plot(s,wigner)
plt.title("4x4 Eigenvalue Splitting and Wigner Surmise")
plt.xlabel("Eigenvalue Splitting ($\lambda_{n+1}-\lambda_{n}$)")
plt.ylabel("Normalized Counts")
plt.legend(["Wigner Surmise","4x4"])
```

[14]:
```
<matplotlib.legend.Legend at 0x7f098edc2290>
```



[15]:
```
plt.figure()
plt.hist(eigval_split10,bins=30,density=True)
plt.plot(s,wigner)
plt.title("10x10 Eigenvalue Splitting and Wigner Surmise")
plt.xlabel("Eigenvalue Splitting ($\lambda_{n+1}-\lambda_{n}$)")
plt.ylabel("Normalized Counts")
plt.legend(["Wigner Surmise","10x10"])
```

[15]:
```
<matplotlib.legend.Legend at 0x7f098f071490>
```



(f) I am generating 1000 2x2 matrices, 1000 4x4 matrices, and 1000 10x10 matrices with elements filled with ± 1.

[16]:
```
# Let's first securely import the GOE matrices for this part.
# pm stands for plus minus.

pm2  = np.copy(GOE2)
pm4  = np.copy(GOE4)
pm10 = np.copy(GOE10)
```

[17]:
```
def pmMatrixGenerator(M):
    # This function takes in a GOE matrix, and turns it into matrix where␣
 ↪elements are pm1.
    for i in range(len(M)):
        for j in range(len(M)):
            if M[i][j] > 0:
                M[i][j] = 1
            elif M[i][j] <= 0:
                M[i][j] = -1
    return M
```

[18]:
```
for m in range(len(pm2)):
    pm2[m] = pmMatrixGenerator(pm2[m])
```

```
    pm4[m] = pmMatrixGenerator(pm4[m])
    pm10[m] = pmMatrixGenerator(pm10[m])

print(pm2[0])
print(pm4[0])
print(pm10[0])
```

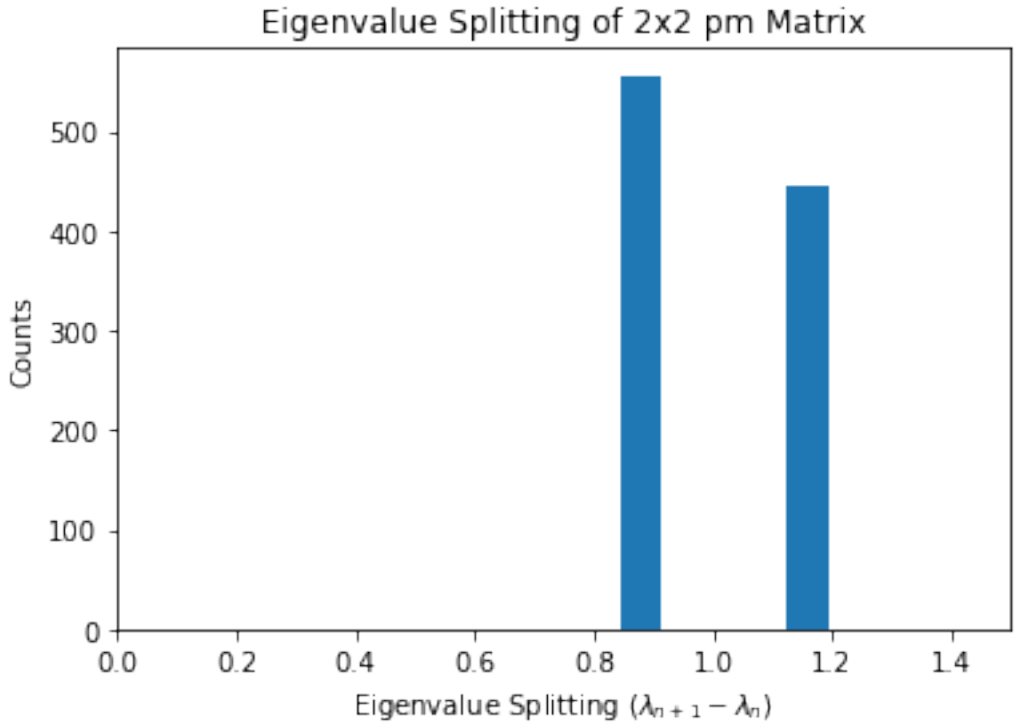
```
[[-1.  1.]
 [ 1. -1.]]
[[-1. -1.  1. -1.]
 [-1.  1. -1.  1.]
 [ 1. -1.  1.  1.]
 [-1.  1.  1.  1.]]
[[ 1.  1.  1.  1. -1. -1.  1. -1.  1. -1.]
 [ 1.  1.  1. -1.  1.  1. -1. -1.  1. -1.]
 [ 1.  1. -1. -1.  1.  1. -1. -1.  1.  1.]
 [ 1. -1. -1. -1.  1.  1. -1. -1.  1.  1.]
 [-1.  1.  1.  1. -1. -1. -1.  1. -1.  1.]
 [-1.  1.  1.  1. -1.  1. -1.  1. -1. -1.]
 [ 1. -1. -1. -1. -1. -1. -1. -1.  1. -1.]
 [-1. -1. -1. -1.  1.  1. -1.  1. -1. -1.]
 [ 1.  1.  1.  1. -1. -1.  1. -1.  1. -1.]
 [-1. -1.  1.  1.  1. -1. -1. -1. -1. -1.]]
```

[19]:
```
# Here I am getting the eigenvalue splittings for pm1 matrices.
# Basically reproducing part (a) with pm matrices.

pm_eigenvalues2  = np.sort(np.real(np.linalg.eigvals(pm2)))
pm_eigenvalues4  = np.sort(np.real(np.linalg.eigvals(pm4)))
pm_eigenvalues10 = np.sort(np.real(np.linalg.eigvals(pm10)))

pm_split2  = np.zeros((1000,1))
pm_split4  = np.zeros((1000,3))
pm_split10 = np.zeros((1000,9))

for i in range(M):
    for j in range(2-1):
        pm_split2[i][j]  = pm_eigenvalues2[i][j+1] - pm_eigenvalues2[i][j]
    for j in range(4-1):
        pm_split4[i][j]  = pm_eigenvalues4[i][j+1] - pm_eigenvalues4[i][j]
    for j in range(10-1):
        pm_split10[i][j] = pm_eigenvalues10[i][j+1] - pm_eigenvalues10[i][j]

pm_mean_split2  = np.mean(np.ndarray.flatten(pm_split2))
pm_mean_split4  = np.mean(np.ndarray.flatten(pm_split4))
pm_mean_split10 = np.mean(np.ndarray.flatten(pm_split10))
```

```
pm_eigval_split2  = np.zeros(1000)
pm_eigval_split4  = np.zeros(1000)
pm_eigval_split10 = np.zeros(1000)

for i in range(M):
    pm_eigval_split2[i]  = (pm_split2[i]/pm_mean_split2)[0]
    pm_eigval_split4[i]  = (pm_split4[i]/pm_mean_split4)[1]
    pm_eigval_split10[i] = (pm_split10[i]/pm_mean_split10)[4]
```

[20]:
```
plt.figure()
plt.hist(pm_eigval_split2,bins=5)
plt.title("Eigenvalue Splitting of 2x2 pm Matrix")
plt.xlabel("Eigenvalue Splitting ($\lambda_{n+1}-\lambda_{n}$)")
plt.ylabel("Counts")
plt.xlim((0,1.5))
```

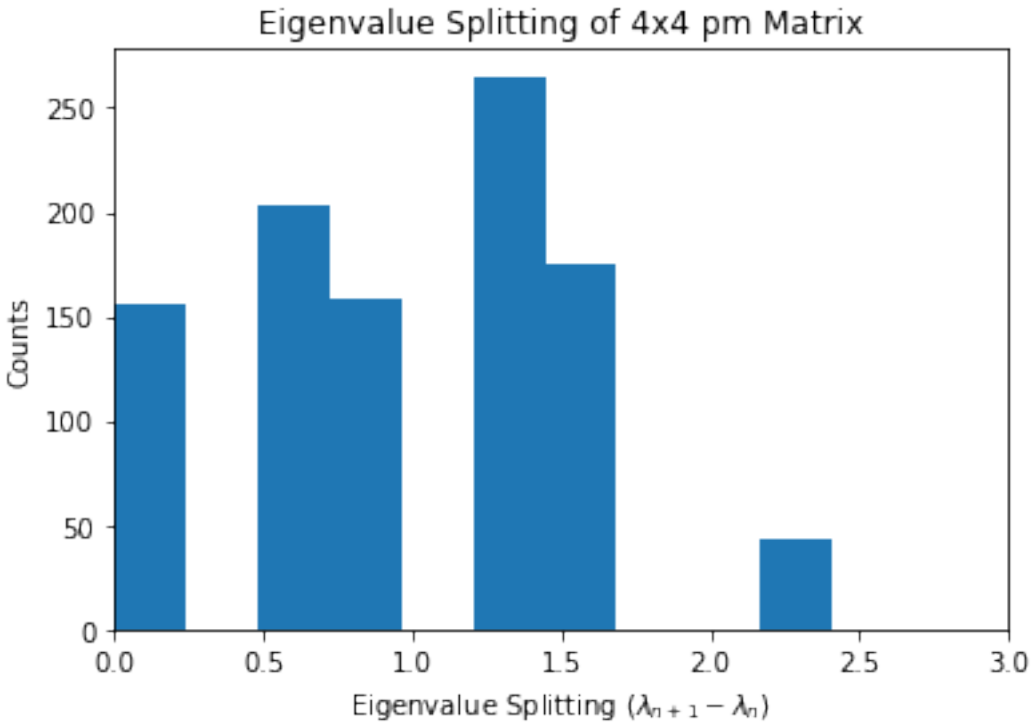
[20]: (0.0, 1.5)



[21]:
```
plt.figure()
plt.hist(pm_eigval_split4,bins=10)
plt.title("Eigenvalue Splitting of 4x4 pm Matrix")
plt.xlabel("Eigenvalue Splitting ($\lambda_{n+1}-\lambda_{n}$)")
plt.ylabel("Counts")
```

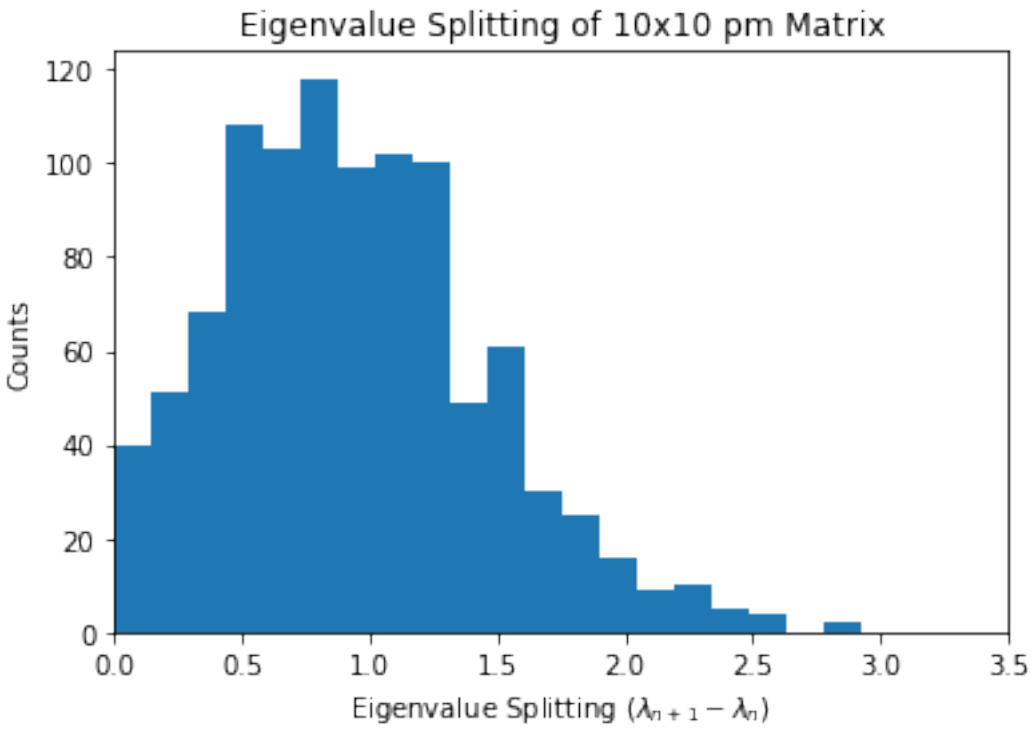
```
plt.xlim((0,3))
```

[21]: (0.0, 3.0)



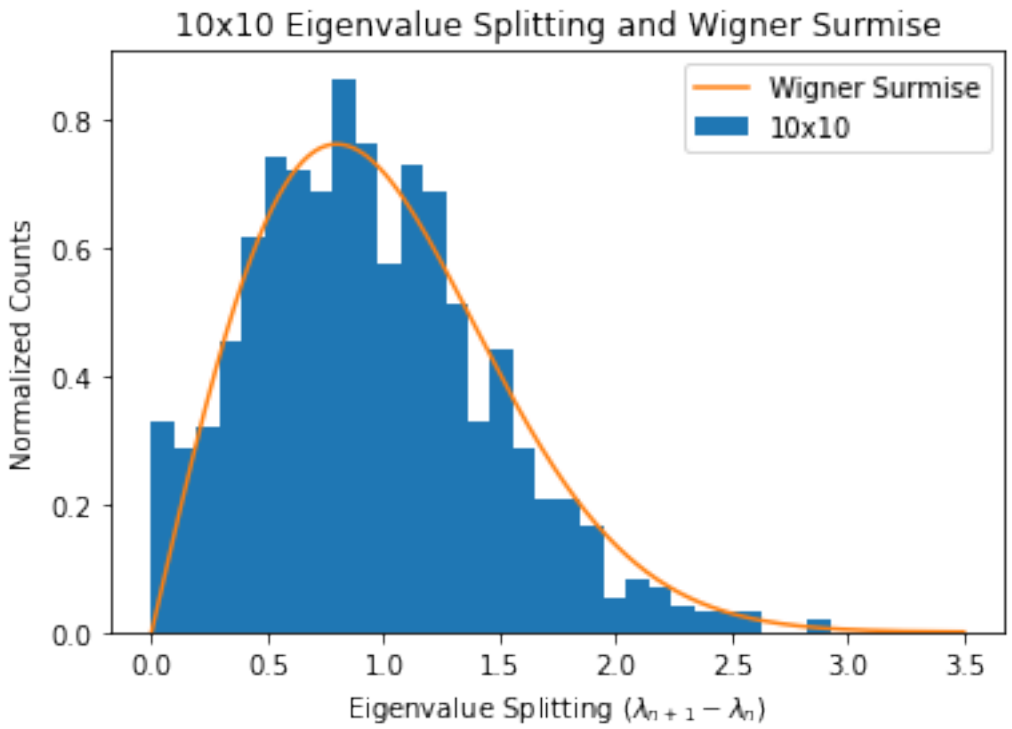
[22]:
```
plt.figure()
plt.hist(pm_eigval_split10,bins=20)
plt.title("Eigenvalue Splitting of 10x10 pm Matrix")
plt.xlabel("Eigenvalue Splitting ($\lambda_{n+1}-\lambda_{n}$)")
plt.ylabel("Counts")
plt.xlim((0,3.5))
```

[22]: (0.0, 3.5)



[23]:
```
# Let's plot the Wigner Surmise along with the eigenvalue split of 10x10 pm␣
↪matrices.

plt.figure()
plt.hist(pm_eigval_split10,bins=30,density=True)
plt.plot(s,wigner)
plt.title("10x10 Eigenvalue Splitting and Wigner Surmise")
plt.xlabel("Eigenvalue Splitting ($\lambda_{n+1}-\lambda_{n}$)")
plt.ylabel("Normalized Counts")
plt.legend(["Wigner Surmise","10x10"])
```

[23]:
```
<matplotlib.legend.Legend at 0x7f098f1a8f10>
```



We can see that for 2×2 and 4×4 matrices, the eigenvalue distributions are NOT universal. However, for 10×10 matrices, the eigenvalue distributions are universal.

1.1.3 Problem 2: The Birthday Problem (Sethna 1.13)

(a) I am constructing a function that returns the fraction of number of classes that have at least two kids with same birthday out of total number of classes. Here, C = number of classes, and K = number of students per class. Therefore, there will be $C \times K$ students in total.

[24]:
```
import random as rand
```

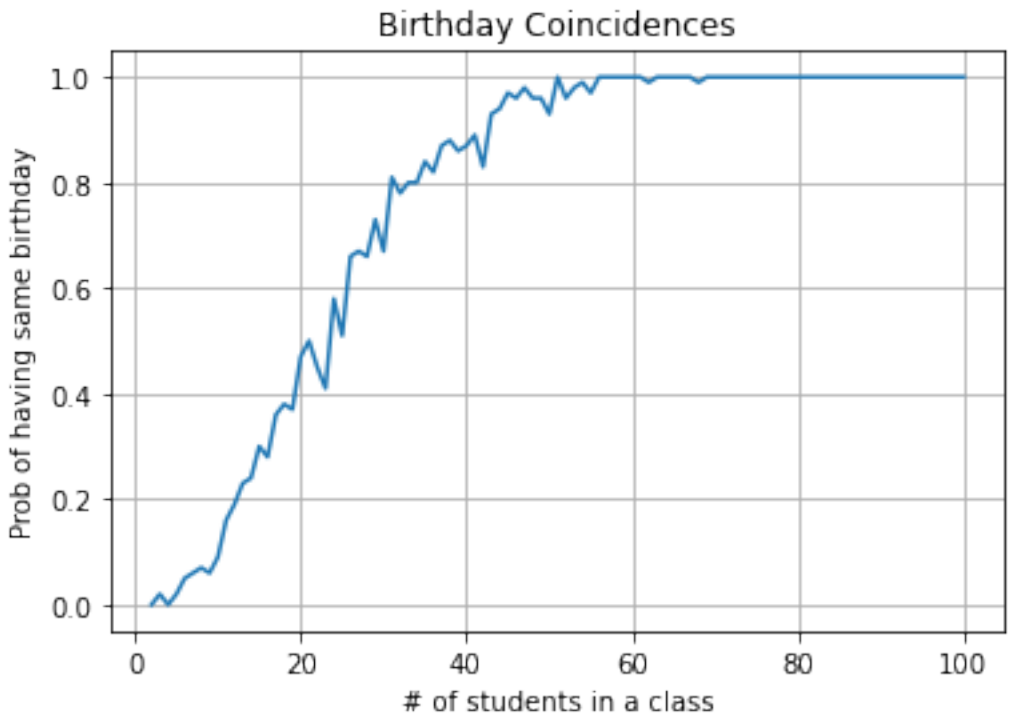
[25]:
```
def BirthdayCoincidences(K,C):
    sameBdayCount = 0
    bdays = np.zeros((C,K))
    for c in range(C):
        for k in range(K):
            bdays[c][k] = rand.randint(1,365)

    sortedBdays = np.sort(bdays,axis=1)

    for c in range(C):
        for k in range(K-1):
            if sortedBdays[c][k] == sortedBdays[c][k+1]:
                sameBdayCount += 1
```

```
                break

    fracSameBday = sameBdayCount/C

    return fracSameBday
```

[26]:
```
nClass = 100
nStudents = 100
K = np.arange(2,nStudents+1)
sameBdayFrac = np.zeros(len(K))

for i in range(len(K)):
    sameBdayFrac[i] = BirthdayCoincidences(K[i],nClass)
```

[27]:
```
plt.plot(K,sameBdayFrac)
plt.grid()
plt.xlabel("# of students in a class")
plt.ylabel("Prob of having same birthday")
plt.title("Birthday Coincidences")
```

[27]:
```
Text(0.5, 1.0, 'Birthday Coincidences')
```



(b) I am writing the formula that returns the probability of K students among D students NOT

sharing a birthday. The exact formula is given by:

$$P(n) = 1 \times \left(1 - \frac{1}{365}\right) \times \left(1 - \frac{2}{365}\right) \times \cdots \times \left(1 - \frac{n-1}{365}\right) = \prod_{n=1}^{K-1} \left(1 - \frac{n}{d}\right) \tag{1}$$

The approximate formula is given by:

$$P(K, D) \approx e^{-\frac{K^2}{2D}} \tag{2}$$

Below is the same fomulae in encoded in Python.

[28]:
```
def noSameBday(K,D):
    prob = 1
    for n in range(K):
        prob = prob * (1-n/D)
    return prob

def approxNoSameBday(K,D):
    return np.exp(-K**2/(2*D))
```

Plugging in $K = 31$ and $D = 30$ (so that $K > D$), we can show that the formula gives zero.

[29]:
```
K = 31
D = 30
print(noSameBday(K,D))
```

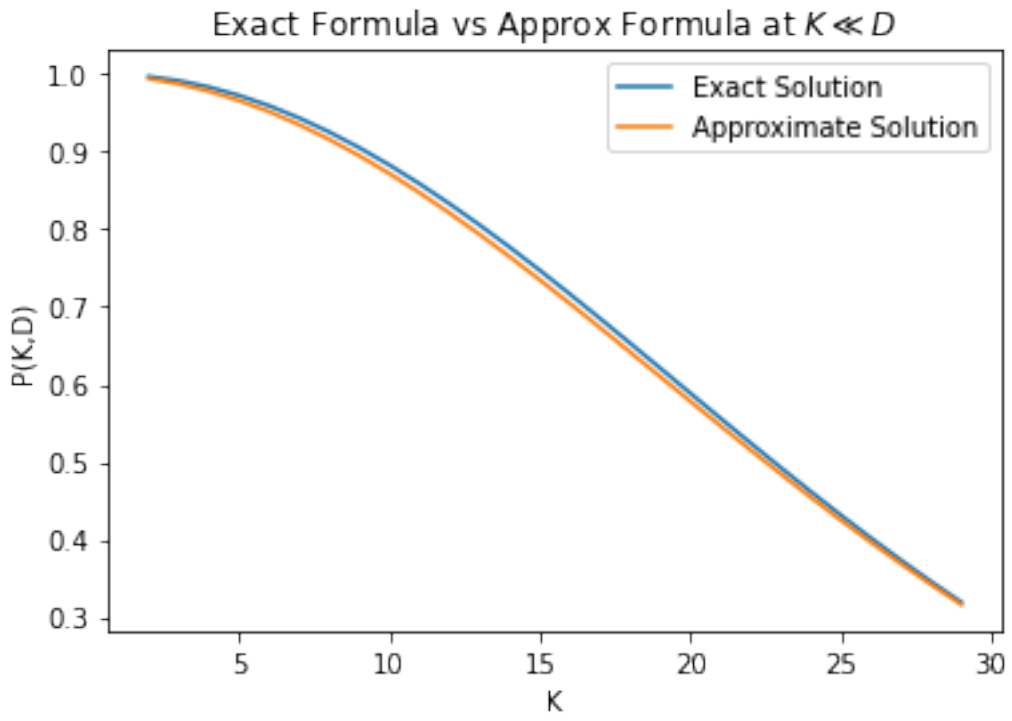
```
0.0
```

Now I am showing that the exact formula and the approximate formula converge in the limit $K \ll D$.

[30]:
```
D = 365
K_array = np.arange(2,30)

prob_array = np.zeros(len(K_array))
approxProb_array = np.zeros(len(K_array))

for i in range(len(K_array)):
    prob_array[i] = noSameBday(K_array[i],D)
    approxProb_array[i] = approxNoSameBday(K_array[i],D)
```

[31]:
```
plt.plot(K_array,prob_array)
plt.plot(K_array,approxProb_array)
plt.title("Exact Formula vs Approx Formula at $K\ll D$")
plt.xlabel("K")
plt.ylabel("P(K,D)")
plt.legend(["Exact Solution","Approximate Solution"])
```

[31]:
```
<matplotlib.legend.Legend at 0x7f09ba8fa210>
```



As shown in the figure above, in the limit $K \ll D$, the approximate solution and exact solution converge.

Inverting the approximate solution gives:

$$P(K, D) \approx e^{-\frac{K^2}{2D}} \Rightarrow K = \sqrt{-2D \ln P} \tag{3}$$

In order to get the number of students required to have 50% chance of sharing a birthday, we set $D = 365$ and $P = 0.5$.

[32]:
```
D = 365
P = 0.5
print("The number of students required to have 50% chance of sharing a birthday␣
 ↪is given by:",
      np.sqrt(-2*D*np.log(P)))
```

```
The number of students required to have 50% chance of sharing a birthday is
given by: 22.49438689559598
```

(c) This is the birthday problem with $D = 2^{32}$. Plugging in the numbers, we get:

[33]:
```
P = 0.5
D = 2**32
print(np.sqrt(-2*D*np.log(P)))
```

```
77162.74323557415
```

One should generate ≈ 77163 random numbers to expect coincidences half the time. We should be worried about coincidences with an old-style random number generator!

To determine the size of the lattice whose elements are generated by a modern random number generator ($D = 2^{52}$), we use the same formula as above with $K = L^3$, $D = 2^{52}$, and $P = 0.5$.

[34]:
```
D = 2**52
P = 0.5

print("The size of the lattice is given by: L = {0:.0f}".format((np.
 ↪sqrt(-2*D*np.log(P)))**(1/3)))
```

```
The size of the lattice is given by: L = 429
```