1(a)

We can use Box Muller Transformation to transform uniform distribution to normal distribution.

The algorithm is as follows. We first start with two random samples of equal length, u1 and u2, drawn from the uniform distribution Uniform(0,1). Then, we generate from them two normally-distributed random variables z1 and z2. Their values are:

$$z_1 = \sqrt{-2\ln(u_1)}\cos(2\pi u_2)$$
$$z_2 = \sqrt{-2\ln(u_1)}\sin(2\pi u_2)$$

1(b)

I use inverse method to transform uniform to exponential distribution.

CDF of exponential distribution: $F(x) = 1 - e^{-\lambda x} = u$

$F^{-1}(x) = -\ln(1-u)/\lambda$, as (1-u) ~ Uniform (0,1), inverse method suggests that

$X = (-\ln(U)/\lambda)$ leads to the random variable following exponential distribution.

1(c)

I use inverse method to transform uniform to Poisson distribution.

CDF of Poisson distribution: $e^{-\lambda}\sum_{i=0}^{\lfloor k\rfloor}\frac{\lambda^i}{i!}$

$X \equiv \min\{p = 0,1,2, \dots | U \leqslant \exp(-\lambda)\sum_{i=0}^{p}\frac{\lambda^p}{p!}\}$

leads to the random variable following Poisson distribution.

1(d)

We can use Gibbs sampler to sample Chi-Square distribution since it is k dimension, which are suitable to Gibbs sampler. With appropriate sampler and parameters, we can sample from Chi-Square distribution from uniform distribution.

1(e)

I can't find the corresponding distribution. The name Fk,m can means a lot of distributions since F is a single letter.

1(f)

We can first use Box Muller Transformation transform uniform random variable to normal random variable, since the PMF of Binomial distribution is quite similar to pdf of normal distribution. Then, use metropolis hasting algorithm with appropriate k to sample Binomial distribution.

1(g)

We can first use Box Muller Transformation transform uniform random variable to normal random variable, since the PMF of Negative Binomial distribution when r is big is quite similar to pdf of normal distribution. Then, use metropolis hasting algorithm with appropriate k to sample Negative Binomial distribution.
If the r is small, we can use metropolis hasting algorithm where q is uniform distribution.

2(a)

```
import numpy as np
import scipy.stats as stats

def p_pdf(x):
    return np.exp(-(np.abs(x) ** 3) / 3)

def f_x(x):
    return x ** 2

n = 1000
mu_approximate = 0
sigma_approximate = 1
q_x = stats.norm(mu_approximate, sigma_approximate)
value_sum = 0
norm_term = 0

for i in range(n):
    # sample from different distribution
    x_i = np.random.normal(mu_approximate, sigma_approximate)
    value = f_x(x_i) * (p_pdf(x_i) / q_x.pdf(x_i))
    value_sum += value
    norm_term += p_pdf(x_i) / q_x.pdf(x_i)
print(f"Importance Sample: {value_sum / norm_term}")
```

Result:

```
Importance Sample: 0.8045488450792782
```

2(b)

```python
import numpy as np
import scipy.stats as stats

def p_pdf(x):
    return np.exp(-(np.abs(x) ** 3) / 3)

def f_x(x):
    return x ** 2

n = 1000
q_x = stats.uniform(-3, 6)
value_sum = 0
count = 0
for i in range(n):
    ui = np.random.uniform(0, 1)
    xi = np.random.uniform(-3, 3)
    if (ui <= p_pdf(xi) / (q_x.pdf(x_i) * 6)):
        count += 1
        value_sum += f_x(xi)
print(f"Rejection Sample: {value_sum / count}")
```

Result:

```
Rejection Sample: 0.7307572492626692
```

3(a)

```python
import random

x = 0
y = 0
n = 20000
h = 1
samples = []

# Generate n samples
for i in range(n):
  e_x = random.uniform(-h, h)
```

```
  e_y = random.uniform(-h, h)
  while abs(x + e_x) > 1 or abs(y + e_y) > 1:
    e_x = random.uniform(-h, h)
    e_y = random.uniform(-h, h)
  x = x + e_x
  y = y + e_y
  samples.append((x, y))

estimator = (4/n) * sum(1 for x, y in samples if x**2 + y**2 <= 1
)
print("PI: ",estimator)
```

Result:

```
PI:  3.4398
```

Increasing the value of n will increase the accuracy of the estimator, since the bigger n means there are more samples, which are obviously can increase the accuracy. In contrast, decreasing the value of n will decrease the accuracy, since there are fewer samples. Increasing the value of h will also increase the accuracy, as the candidates will be generated from a larger range and will have bigger possibility to fall within the square. Decreasing the value of h will decrease the accuracy. The candidates will be generated from a smaller range and will have smaller possibility to fall within the square.

3(b)

The major flaw of this method is that the produced samples are not from the desired distribution (uniformly on the square) with the correct probability. We can use metropolis-Hasting algorithm to produce correct samples.
Metropolis Ratio: (p(x*) * effective_area(x*)) / (p(x)*effective_area(x))
The effective area means the overlapping area of original square and the square center by sample point with area of h^2. By this mean, the more area the sample's rectangle is overlapped with origin square, the more likely it will be kept. Since more overlapping area means the further samples sampled from this area can have higher possibility to stay in the origin square.

3(c)

```
import random

def overlappingArea(l1, r1, l2, r2):
    x = 0
```

```
    y = 1
    x_dist = (min(r1[x], r2[x]) - max(l1[x], l2[x]))
    y_dist = (min(r1[y], r2[y]) - max(l1[y], l2[y]))
    areaI = 0
    if x_dist > 0 and y_dist > 0:
        areaI = x_dist * y_dist
    return areaI

def p(x, y):
  return abs(x) < 1 and abs(y) < 1

x = 0
y = 0
n = 20000
h = 1
samples = []

# Generate n samples
for i in range(n):
  e_x = random.uniform(-h, h)
  e_y = random.uniform(-h, h)
  x_candidate = x + e_x
  y_candidate = y + e_y
  # Calculate the overlapping area with square
  origin_area = overlappingArea([x-h,y-h], [x+h,y+h], [-1,-
1], [1,1])
  candidate_area = overlappingArea([x_candidate-h,y_candidate-
h], [x_candidate+h,y_candidate+h], [-1,-1], [1,1])

  if random.random() < min(1, (p(x_candidate, y_candidate) / p(x,
 y)) * (candidate_area / origin_area)):
    x = x_candidate
    y = y_candidate
  samples.append((x, y))

estimator = (4/n) * sum(1 for x, y in samples if x**2 + y**2 <= 1
)
print("PI: ",estimator)
```

Result:

```
PI:  3.4118
```

Increasing the value of h will decrease the accuracy in some cases. Decreasing the value of h will also decrease the accuracy in some cases. There is a sweet point of h. The value of h cannot be too big or too small.

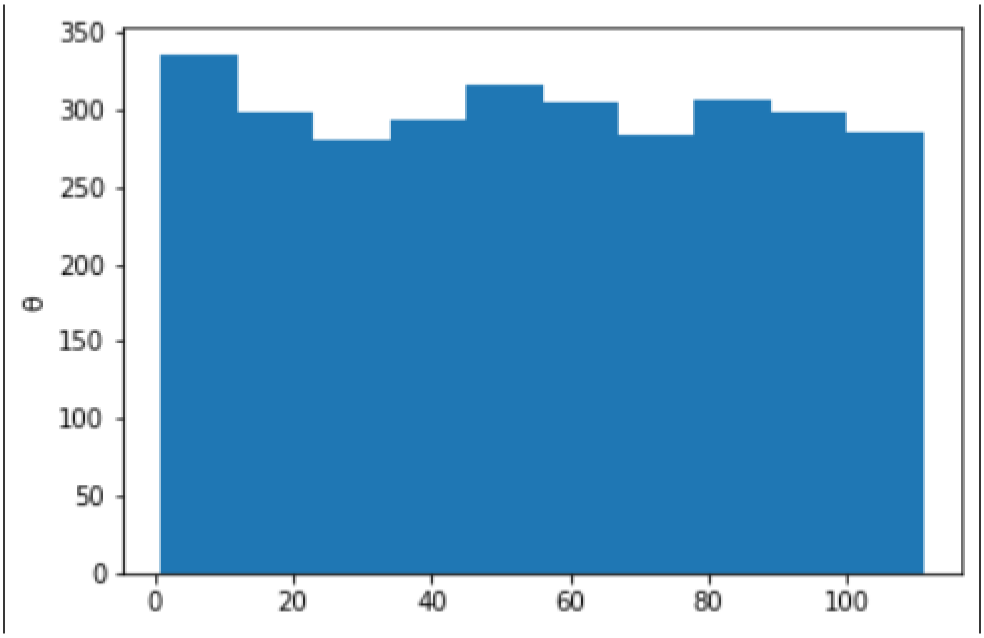
4(a)

```
import numpy as np
import scipy.stats as stats
import matplotlib.pyplot as plt
import pandas as pd

iter = 3000
lambda1 = []
lambda2 = []
theta = []
for i in range(iter):
    theta.append(np.random.randint(1, 112))
    a1 = np.random.gamma(shape=10, scale=1/10)
    lambda1.append(np.random.gamma(shape=3, scale=1/a1))

    a2 = np.random.gamma(shape=10, scale=1/10)
    lambda2.append(np.random.gamma(shape=3, scale=1/a2))
print(f"Mean of θ = {np.mean(theta)}\nMean of λ1 = {np.mean(lambd
a1)}\nMean of λ2 = {np.mean(lambda2)}")
plt.hist(theta)
plt.ylabel('θ', fontsize="11")
plt.show()
plt.hist(lambda1)
plt.ylabel('λ1', fontsize="11")
plt.show()
plt.hist(lambda2)
plt.ylabel('λ2', fontsize="11")
plt.show()
```

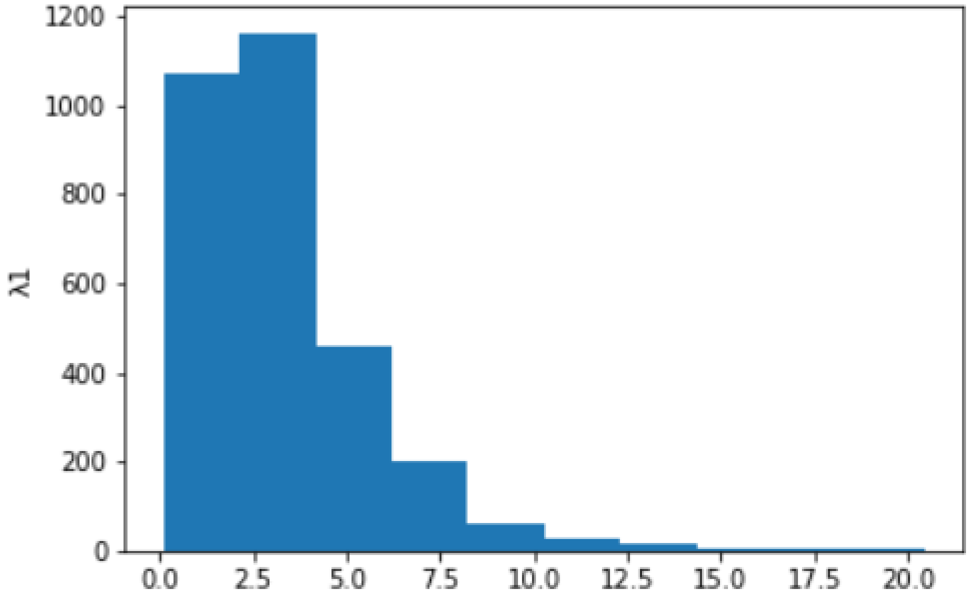
Result:

```
Mean of θ = 54.86666666666667
Mean of λ1 = 3.289895459661857
Mean of λ2 = 3.271441861240158
```

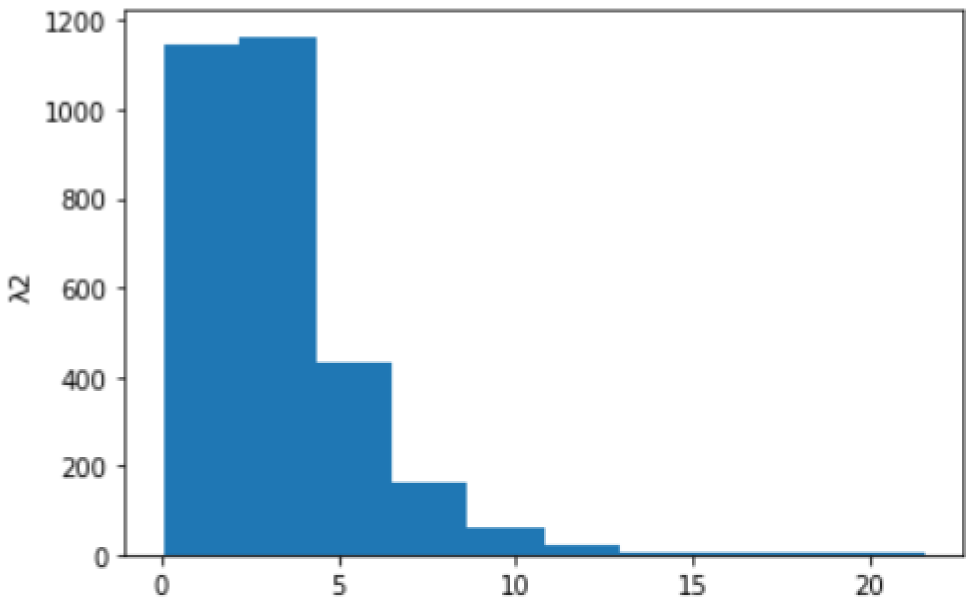
θ Histogram:



λ1 Histogram:



λ2 Histogram:



4(b)

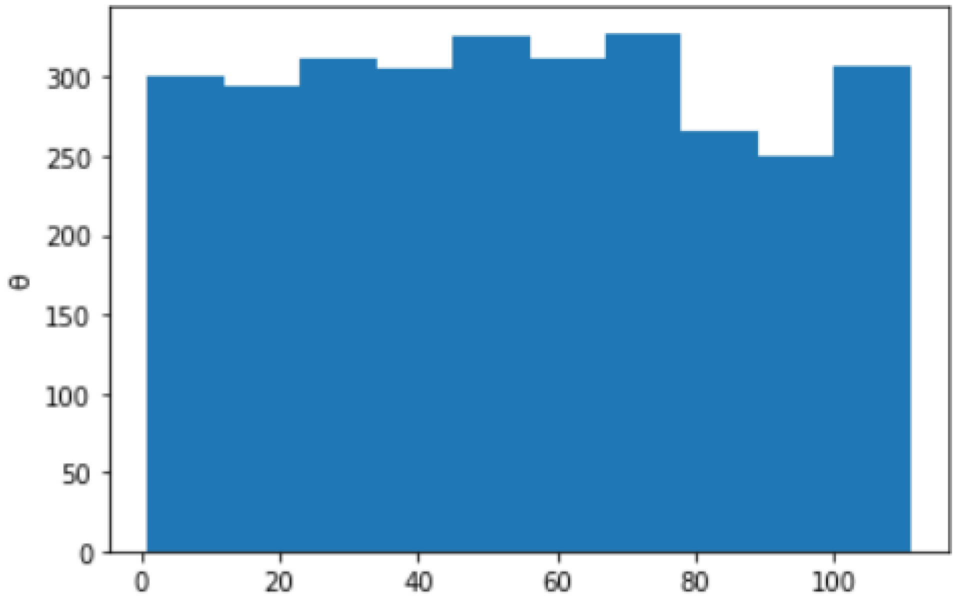
```
import numpy as np
import scipy.stats as stats
import matplotlib.pyplot as plt
import pandas as pd

iter = 3000
lambda1 = []
lambda2 = []
theta = []
for i in range(iter):
    theta.append(np.random.randint(1, 112))
    a1 = np.random.gamma(shape=10, scale=1/10)
    lambda1.append(np.random.gamma(shape=3, scale=1/a1))
    lambda2.append(lambda1[i] * np.random.uniform(np.log(1/8), np
.log(2)))
print(f"Mean of θ = {np.mean(theta)}\nMean of λ1 = {np.mean(lambd
a1)}\nMean of λ2 = {np.mean(lambda2)}")
plt.hist(theta)
plt.ylabel('θ', fontsize="11")
plt.show()
plt.hist(lambda1)
plt.ylabel('λ1', fontsize="11")
plt.show()
```

```
plt.hist(lambda2)
plt.ylabel('λ2', fontsize="11")
plt.show()
```

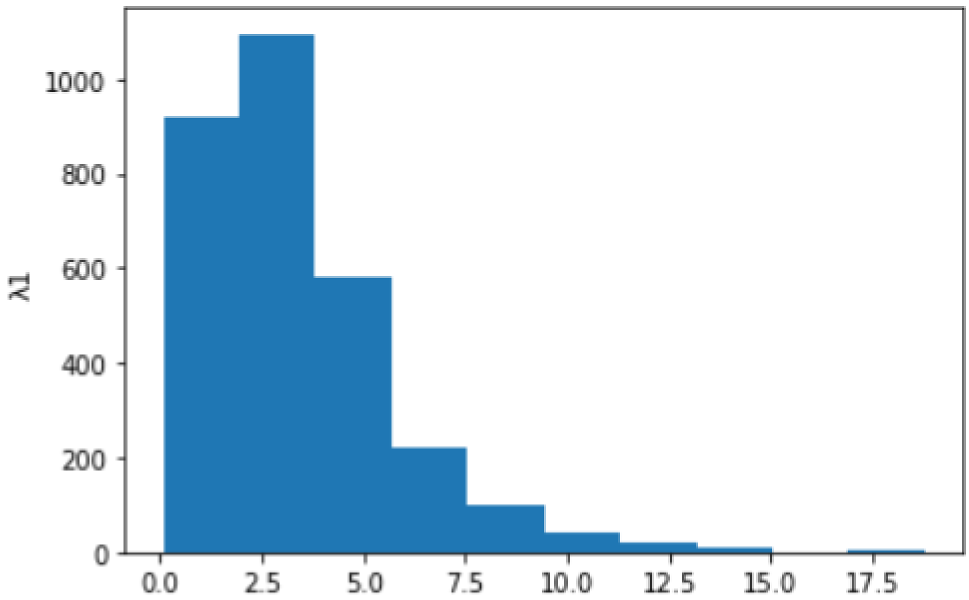
Result:

```
Mean of θ = 54.71
Mean of λ1 = 3.369011645294816
Mean of λ2 = -2.370182829332286
```

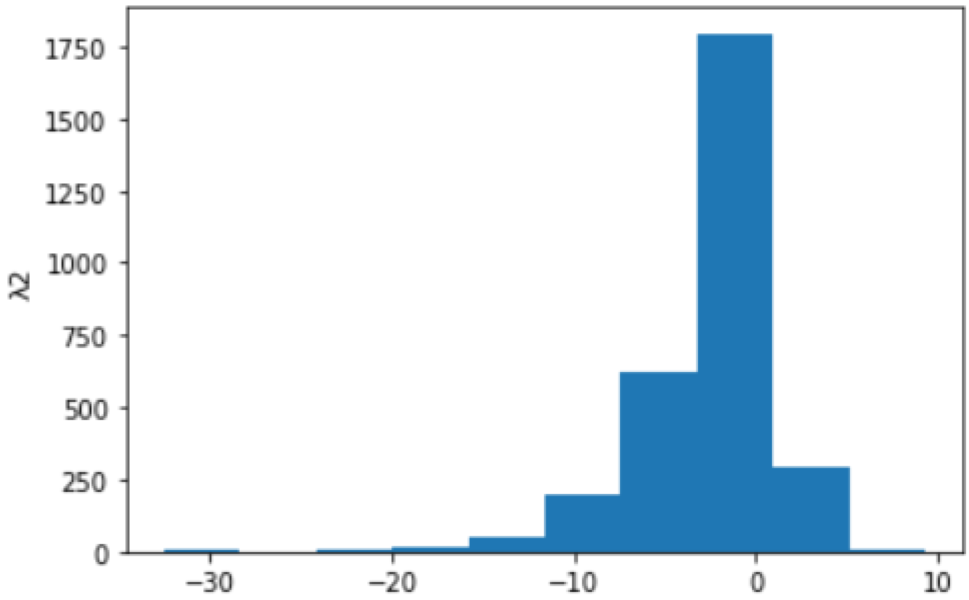
θ Histogram:



λ1 Histogram:



λ2 Histogram:



4(c)

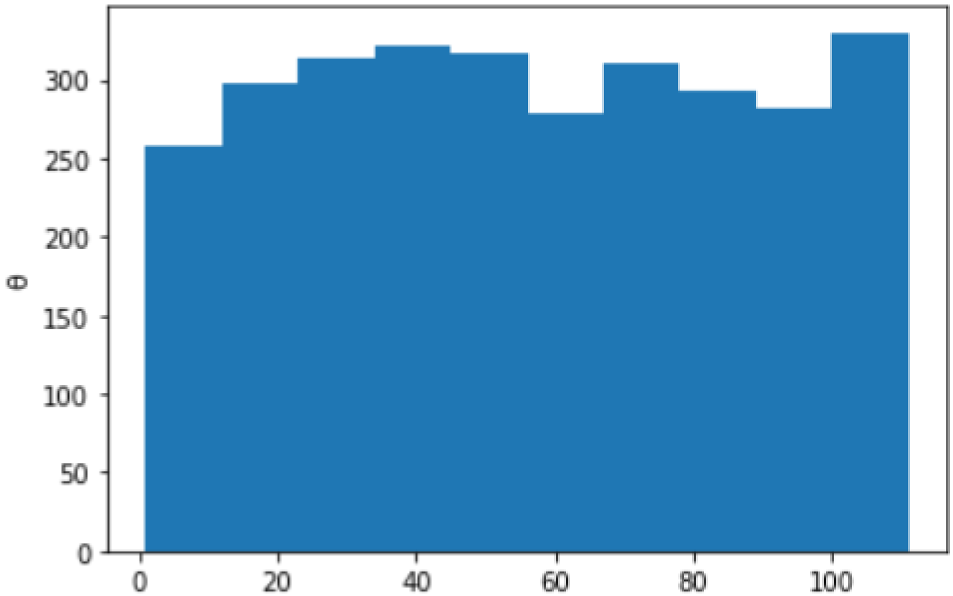
```
import numpy as np
import scipy.stats as stats
import matplotlib.pyplot as plt
import pandas as pd

iter = 3000
lambda1 = []
lambda2 = []
theta = []
for i in range(iter):
    theta.append(np.random.randint(1, 112))
    a1 = np.random.uniform(0, 100)
    lambda1.append(np.random.gamma(shape=3, scale=1/a1))

    a2 = np.random.uniform(0, 100)
    lambda2.append(np.random.gamma(shape=3, scale=1/a2))
print(f"Mean of θ = {np.mean(theta)}\nMean of λ1 = {np.mean(lambd
a1)}\nMean of λ2 = {np.mean(lambda2)}")
plt.hist(theta)
plt.ylabel('θ', fontsize="11")
plt.show()
plt.hist(lambda1)
plt.ylabel('λ1', fontsize="11")
```

```
plt.show()
plt.hist(lambda2)
plt.ylabel('λ2', fontsize="11")
plt.show()
```

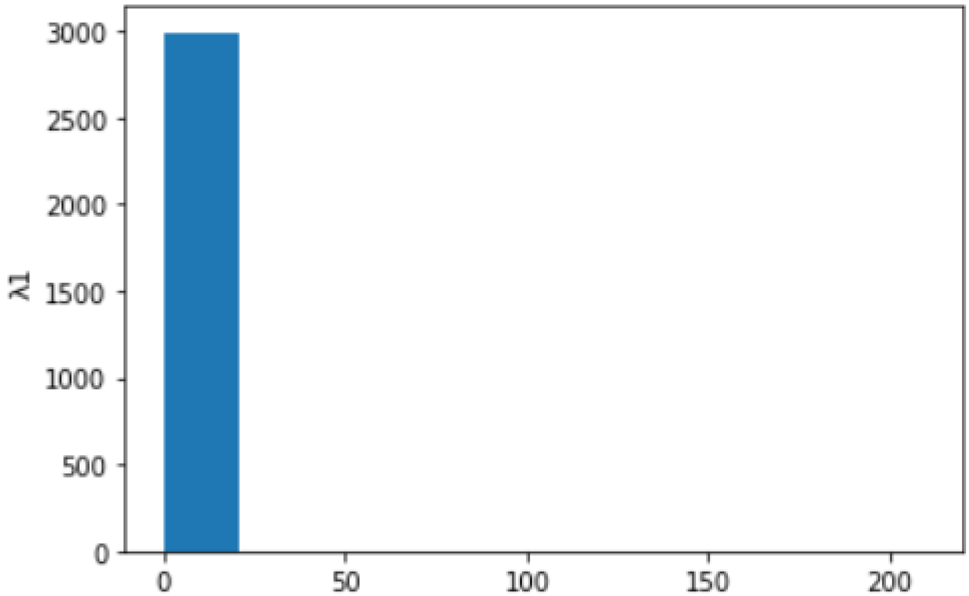
Result:

```
Mean of θ = 56.10666666666667
Mean of λ1 = 0.33042836770939954
Mean of λ2 = 0.2616818074413835
```

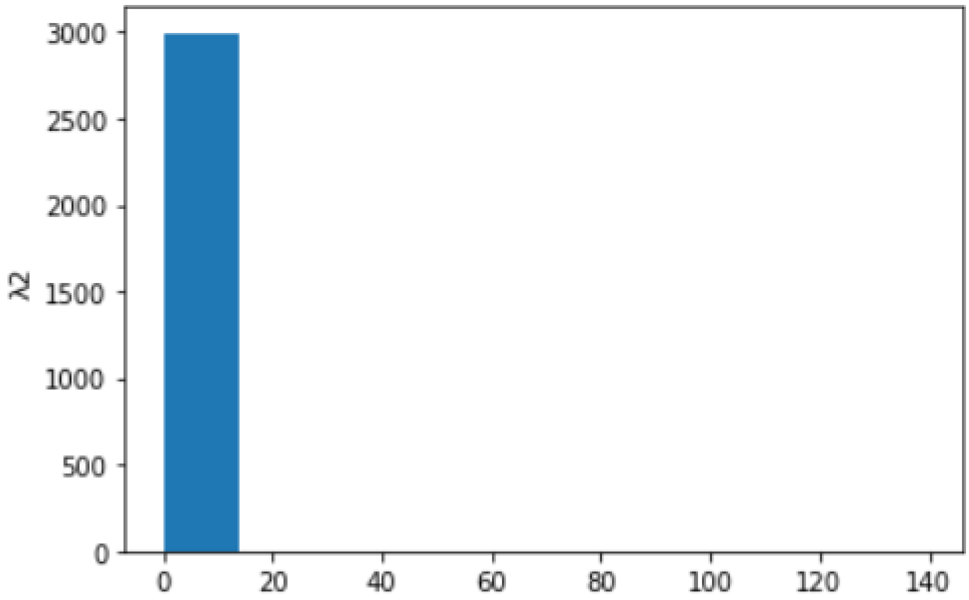
θ Histogram:



λ1 Histogram:



λ2 Histogram:



4(d)

$$P(\lambda 1|X) \propto \frac{e^{-\lambda 1}\lambda 1^{X}}{X!} \times \lambda 1^{2} e^{-a1\lambda 1}$$

$$a1 \sim \text{Gamma}(10{,}10) = \frac{1}{X!} e^{\lambda 1(-1-a1)} \lambda 1^{x+2}$$

⇨ Same as P(λ2|X)

$X^{(i)} \sim Poisson(\lambda^{i-1})$

$\theta^{(i)} \sim Gamma(3 + X, a1 + 1$

4(e)

```
import numpy as np
import scipy.stats as stats
import matplotlib.pyplot as plt
import pandas as pd

def gibbsSampler():
    theta = np.random.randint(1, 112)
    x_arr, lambda1_arr, lambda2_arr = [], [], []
    x_arr.append(0)
```

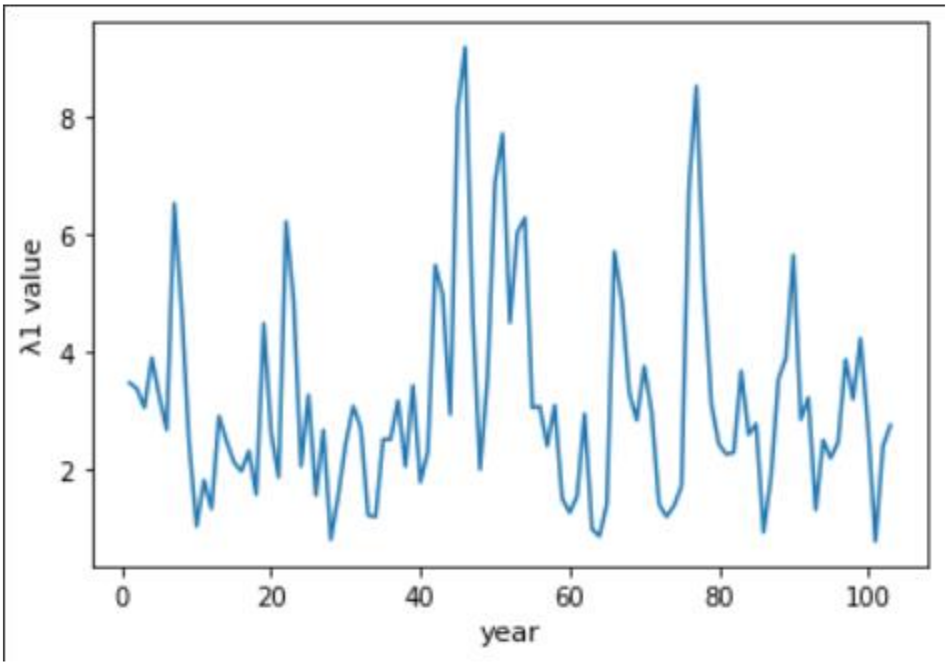
```
    a1 = np.random.gamma(shape=10, scale=1/10)
    lambda1_arr.append(np.random.gamma(shape=3, scale=1/a1))
    for i in range(theta):
        xi = np.random.poisson(lam=lambda1_arr[i])
        x_arr.append(xi)
        lambda1_arr.append(np.random.gamma(shape=3+xi, scale=1/(a
1+1)))

    x_arr.append(0)
    a2 = np.random.gamma(shape=10, scale=1/10)
    lambda2_arr.append(np.random.gamma(shape=3, scale=1/a2))
    for i in range(112 - theta):
        xi = np.random.poisson(lam=lambda2_arr[i])
        x_arr.append(xi)
        lambda2_arr.append(np.random.gamma(shape=3+xi, scale=1/(a
1+1)))

    return theta, lambda1_arr, lambda2_arr

theta, lambda1, lambda2 = gibbsSampler()
print(f"Mean of λ1 : {np.mean(lambda1)}\nMean of λ2 : {np.mean(la
mbda2)}")
print(f"Standard deviation of λ1 : {np.std(lambda1)}\nStandard de
viation of λ2 : {np.std(lambda2)}")
print(f"Standard error of λ1 : {np.std(lambda1)/(theta**0.5)}\nSt
andard error of λ2 : {np.std(lambda2)/((112-theta)**0.5)}")

x1 = []
x2 = []
for i in range(len(lambda1)):
    x1.append(i+1)
for i in range(len(lambda2)):
    x2.append(theta+i+1)
plt.plot(x1, lambda1)
plt.xlabel('year', fontsize="11")
plt.ylabel('λ1 value', fontsize="11")
plt.show()
plt.plot(x2, lambda2)
```

```
plt.xlabel('year', fontsize="11")
plt.ylabel('λ2 value', fontsize="11")
plt.show()
```

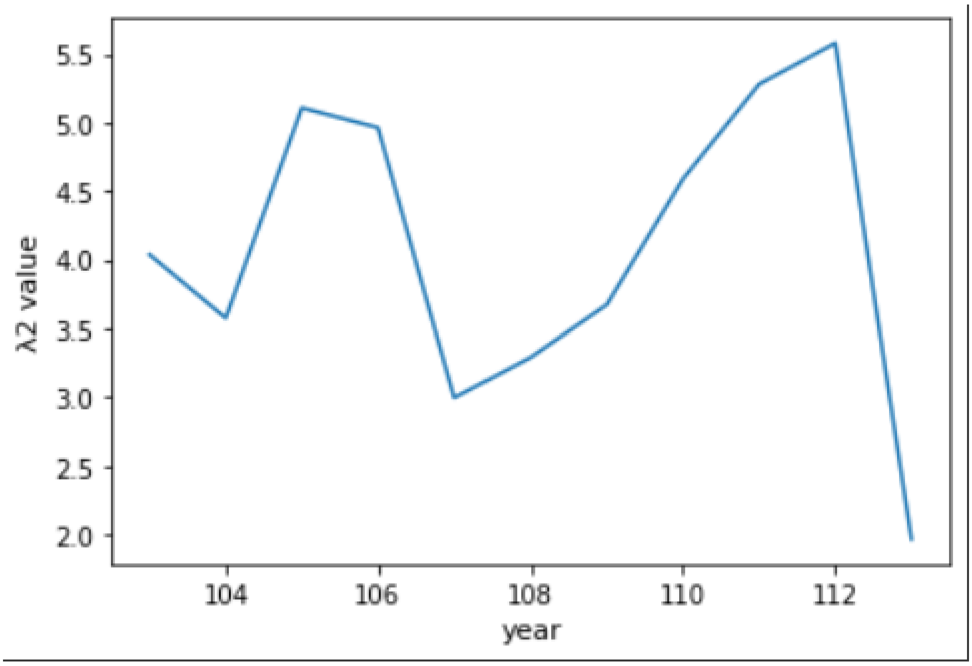
Result:

```
Mean of λ1 : 2.4298703561037085
Mean of λ2 : 2.4148699068420054
Standard deviation of λ1 : 1.2258030426299547
Standard deviation of λ2 : 1.2053257842671004
Standard error of λ1 : 0.1751147203757078
Standard error of λ2 : 0.15185677495164937
```

λ1 Value graph:



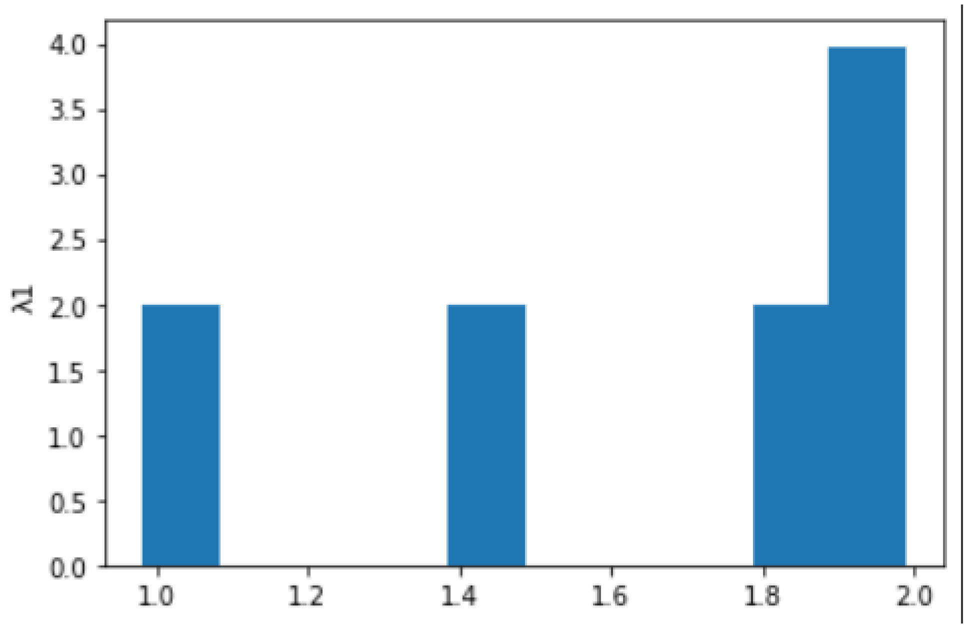
λ2 Value graph:



4(f)

density histogram:

λ1



λ2

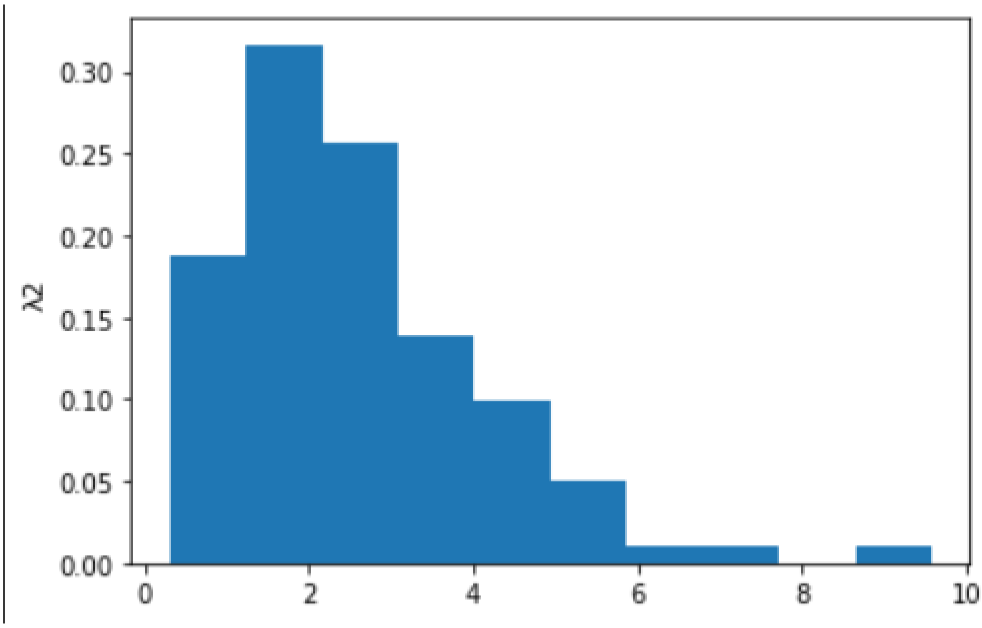


table of summary statistics:

λ1 statistics:	
count	5.000000
mean	1.644957
std	0.424358
min	0.983174
25%	1.469364
50%	1.815852
75%	1.967625
max	1.988774

λ2 statistics:	
count	109.000000
mean	2.565402
std	1.558780
min	0.306494
25%	1.397507
50%	2.251526
75%	3.419249
max	9.577643

6(a)

We know the fact that the expected value of X* is equal to the expected value of the original data X, x̄. The variance of X* is equal to the variance of the original data divided by the number of samples n, μ^2/n.
I use a simple Python code to show this:

```
import numpy as np

# Suppose we have a dataset X with mean x¯ and variance μ^2
X = [1, 2, 3, 4, 5]
x_mean = np.mean(X)
x_var = np.var(X)

# We can calculate the expected value and variance of X*
X_star = np.random.choice(X, size=(len(X), len(X)), replace=True)
x_temp = np.mean(X_star, axis=1)
x_star_mean = np.mean(x_temp)
x_star_var = np.var(x_temp)

print("Expected value of X:", x_mean)
print("Expected value of X*:", x_star_mean)
print("Variance of X:", x_var)
print("Variance of X*:", x_star_var)
```

Result:

```
Expected value of X: 3.0
Expected value of X*: 2.9200000000000004
Variance of X: 2.0
Variance of X*: 0.5216000000000001
```

6(b)

The first two terms of the expansion are:

E*(R(X+, F^)) ≈ g(x̄) - g(μ) + g'(μ)*(x̄ - μ)

var*(R(X+, F^)) ≈ g''(μ)*μ^2/n