Project in Probability and Statistics
Master HPC-IA Mines Paris

Thomas Bancel
Dimitrios Tsitsos

December 5, 2022

1 Parameter estimation of the Gamma distribution.

1.1 Log-likelihood of the model.

$$P(X) = f(a,b)(x) = \frac{b^a}{\Gamma(a)}\ x^{(a-1)}\ e^{(-bx)}$$

$$L(a,b) = ?$$

$$L(x;a,b) = \prod_{i=1}^{n} f_{(a,b)}(x_i) = \prod_{i=1}^{n} \frac{b^a}{\Gamma(a)} x_i^{(a-1)} e^{(-bx_i)} = \frac{b^a}{\Gamma(a)} \prod_{i=1}^{n} x_i^{(a-1)} e^{(-bx_i)}$$

$$L(a,b) = naln(b) - nln(\Gamma(a)) - b\sum_{i=1}^{n} x_i + (a-1)\sum_{i=1}^{n} ln(x_i)$$

1.2 Generate one sample of X denoted x.

We have used the already implemented function of numpy to generate a sample following the gamma distribution.

```
a, b = 2, 2
n = 20
x = np.random.gamma(a, 1/b, n)
```

See above the code to generate a sample of size n = 20 for fixed values a = 2 and b = 2. The second argument of the function "np.random.gamma" is $\theta = \frac{1}{b}$.

1.3 Evaluation of log-likelihood function.

Here is the implementation of the log-likelihood function calculated in the first question :

```
def log_likelihood(a,b,x):
    return
        len(x)*a*math.log(b)-len(x)*math.log(math.gamma(a))-b*np.sum(x)+(a-1)*np.sum(np.log(x))
```

This function take as input, a, b and the sample x to compute the log-likehood function.

1.4 Visualisation of log-likelihood.

For the visualization we use Matplotlib, but first we are computing the value of the log-likelihood function for a mesh of a and b :

```python
def plot_2_3d(x, a, b):
    A =np.arange(0.1, 10, 0.1)
    B = np.arange(0.1, 10, 0.1)
    #Computing the value of the log-likelihood function for the mesh of a, b
    aa, bb = np.meshgrid(A,B)
    ll = np.zeros(aa.shape)
    for i in range(aa.shape[0]):
        for j in range(aa.shape[1]):
            ll[i, j] = log_likelihood(aa[i,j],bb[i,j],x)
    #2D plot
    fig = plt.figure(figsize = (5, 5))
    plt.imshow(ll.reshape(aa.shape),
              origin='lower', aspect='auto',
              extent=[min(A), max(A), min(B), max(B)],
              cmap='plasma', vmin = np.max(ll)+np.max(ll)/58, vmax = np.max(ll))
    cb = plt.colorbar()
    plt.title('2D loglikelihood with a_true= {}, b_true={} for a size n={}'.format(a, b,
        len(x)))
    fig.show()
    #3D plot
    fig = plt.figure(figsize = (10, 10))
    ax = plt.axes(projection='3d')
    plot = ax.scatter3D(aa, bb, ll, c=ll.reshape(aa.shape), cmap='plasma', vmin =
        np.max(ll)+np.max(ll)/58, vmax = np.max(ll));
    ax.set_xlabel('coef a')
    ax.set_ylabel('coef b')
    ax.set_zlabel('loglikelihood')
    ax.view_init(-140, 40)
    fig.colorbar(plot, ax = ax, shrink = 0.5, aspect = 10)
    plt.title('3D loglikelihood with a_true= {}, b_true={} for a size n={}'.format(a, b,
        len(x)))
    fig.show()
```

The visualisation for N=2000, with a=5 and b=2:

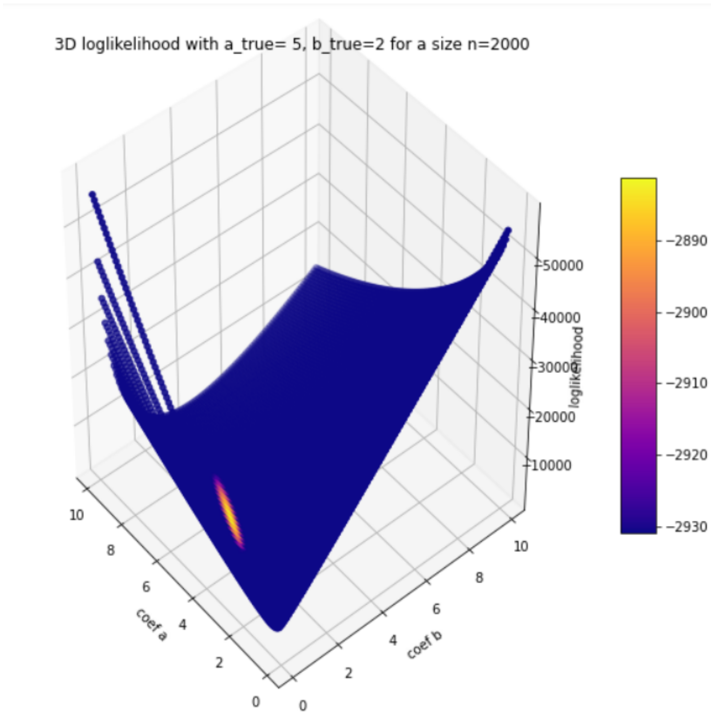

Figure 1: 3D plot of the log-likelihood function

We have also made a visualization in 2D because it is easier to see where is the maximum.

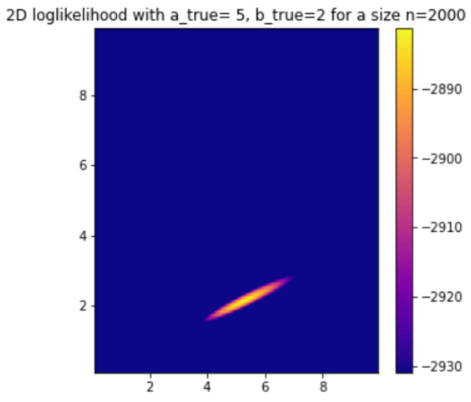

Figure 2: 2D plot of the log-likelihood function to see where is the maximum (yellow)

1.5 Visualisation with different data sets, different sizes and different parameters.

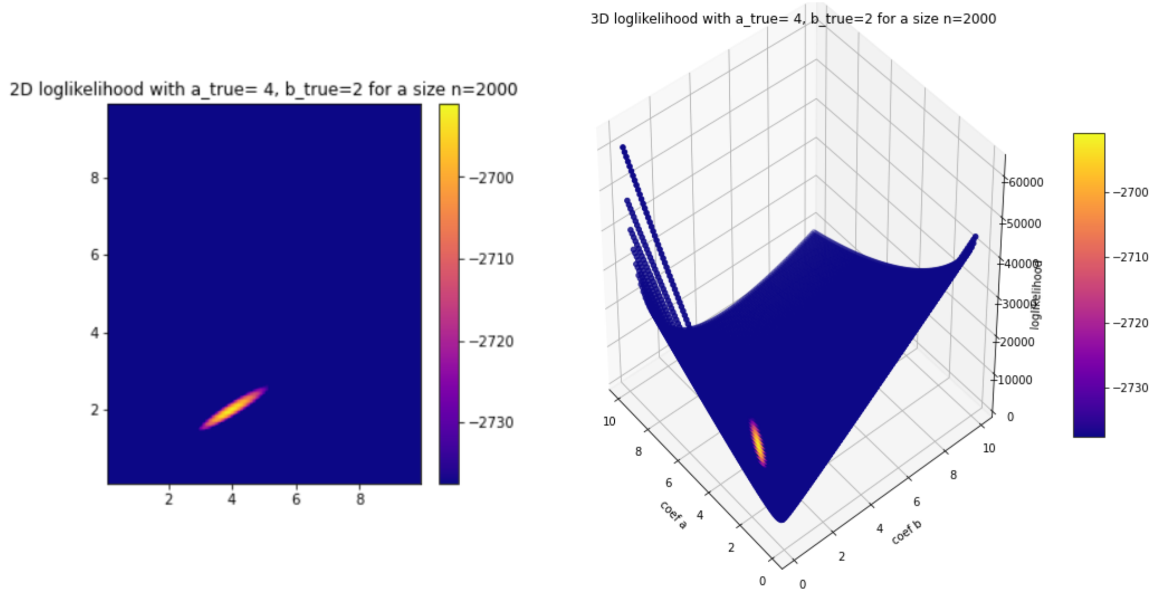

Figure 3: Plot of the log-likelihood function with a=4, b=2 and n=2000

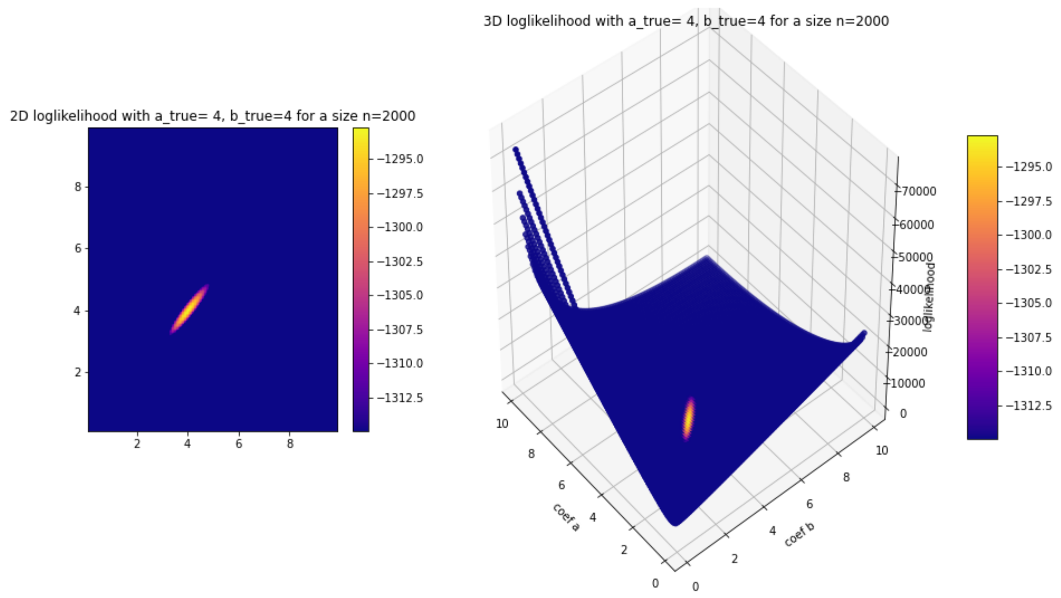

Figure 4: Plot of the log-likelihood function with a=4, b=4 and n=2000

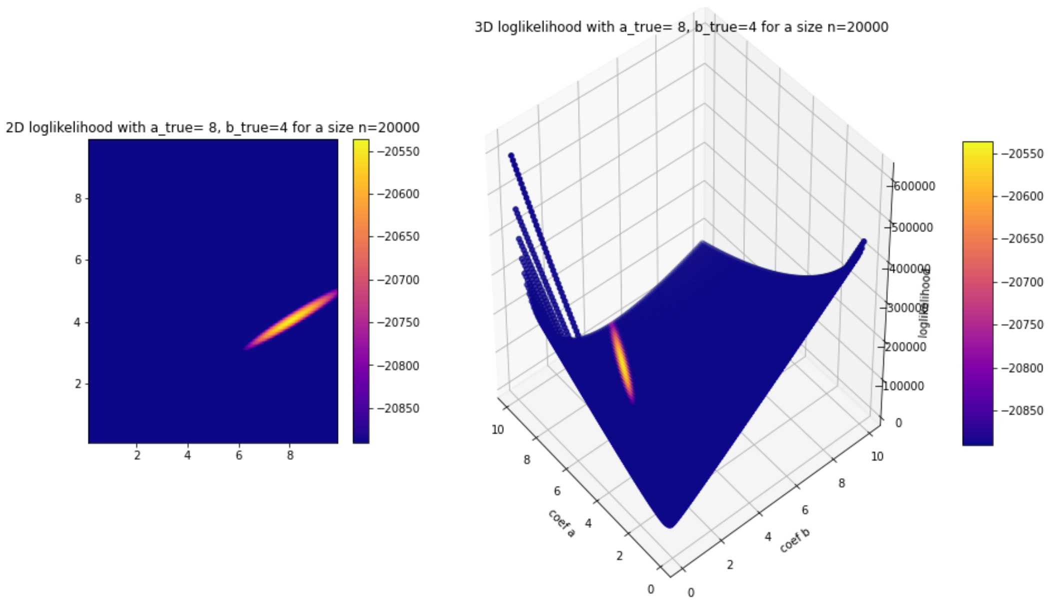

Figure 5: Plot of the log-likelihood function with a=8, b=4 and n=20000

1.5.1 What is the typical shape of the log-likelihood function?

The shape of the function look like a cone.

1.5.2 How many critical points does it have?

There is only one critical point.

1.5.3 Where is the maximum of the function apparently located ?

The critical point seems to be when a is equal to the real a and b is equal to the real b.

1.6 Gradient, Hessian calculation.

For the **Gradient** we need to verify that:

$$\nabla l(a,b) = (\frac{\delta}{\delta a}l(a,b), \frac{\delta}{\delta b}l(a,b))^T = (nln(b) - nln(\Gamma(a))' + \sum_{i=1}^{n} ln(x_i), \frac{na}{b} - \sum_{i=1}^{n} x_i)$$

First we calculate the first member:

$$\frac{\delta}{\delta a}L(a,b) = nln(b) - nln(\Gamma(a))' + \sum_{i=1}^{n} ln(x_i)$$

Then we calculate the second member:

$$\frac{\delta}{\delta b}L(a,b) = fracnab - \sum_{i=1}^{n} x_i$$

We can see that we have what we expect!

For the **Hessian** we need to verify that:

$$H(a,b)^{-1} = \begin{pmatrix} \frac{\delta^2}{\delta^2 a}l(a,b) & \frac{\delta^2}{\delta a \delta b}l(a,b) \\ \frac{\delta^2}{\delta a \delta b}l(a,b) & \frac{\delta^2}{\delta^2 b}l(a,b) \end{pmatrix}^{-1} = \frac{1}{n(1 - aln(\Gamma(a))'')} \begin{pmatrix} a & b \\ b & b^2 ln(\Gamma(a))'' \end{pmatrix}$$

First we calculate:

$$\frac{\delta^2}{\delta^2 a}l(a,b) = -nln(\Gamma(a))''$$

$$\frac{\delta^2}{\delta a \delta b}l(a,b) = \frac{n}{b}$$

$$\frac{\delta^2}{\delta^2 b}l(a,b) = -\frac{na}{b^2}$$

$$det = -\frac{n^2(1 - ln(\Gamma(a))'')}{b^2}$$

And then we have:

$$H(a,b)^{-1} = -\frac{b^2}{n^2(1 - a(ln(\Gamma(a)))'')} \begin{pmatrix} -\frac{na}{b^2} & -\frac{n}{b} \\ -\frac{n}{b} & -nln(\Gamma(a))'' \end{pmatrix}$$

$$H(a,b)^{-1} = \frac{1}{n(1 - aln(\Gamma(a))'')} \begin{pmatrix} a & b \\ b & b^2 ln(\Gamma(a))'' \end{pmatrix}$$

We can see that we have what we expect!

1.7 Update of parameters a, b.

Implementation of the update with the Newton method :

```python
def UpdateNewton(x, a, b):
  ab = np.array([[a],[b]])
  sum_log = 0
  grad = np.array([[len(x)*math.log(b) - len(x)*(polygamma(0,a)) + np.sum(np.log(x))],
     [((len(x)*a)/b) - np.sum(x) ]])
  hessian_coef = 1/(len(x)*(1 - a*(polygamma(1, a))))
  hessian = np.array([[hessian_coef*a, hessian_coef*b],
                   [hessian_coef*b, hessian_coef * ((b**2)*polygamma(1, a))]])
  result = np.zeros((2,1))
  result = ab - hessian @ grad
  return result[0,0], result[1,0]
```

1.8 Calculation of MLE with Newton-Raphson method.

Definition of the function MLEgamma which use the Newton-Raphson method coded in the question 1.7, to estimate a and b. The function also save the number of iteration to converge.

```
def MLEgamma(x, a0, b0):
  iter = 1
  a_newton, b_newton = UpdateNewton(x, a0, b0)
  if ((a_newton < 0) or (b_newton < 0)):
    print('negative value')
    return
  while (abs((log_likelihood(a_newton,b_newton,x) - log_likelihood(a0,b0,x)) /
      log_likelihood(a_newton,b_newton,x)) > 0.001):
    if (iter >= 100):
      return a_newton, b_newton
    else:
      a0 = a_newton
      b0 = b_newton
      a_newton, b_newton = UpdateNewton(x, a0, b0)
      if ((a_newton < 0) or (b_newton < 0)):
        print('negative value')
        return
      iter = iter+1
  return [a_newton, b_newton, iter]
```

1.9 Modification of the algorithm.

Verifying the number of iteration of the MLEgamma, in the case if the convergence is slow:

```
 if (iter >= 100):
     return a_newton, b_newton
```

1.10 Testing of function.

Testing the function MLEgamma with first guess is equal to the real a and b :

```
a, b = 2, 2
n = 2000
x = np.random.gamma(a, 1/b, n)

result = MLEgamma(x,a,b)
print_result_lme(result)
```

The estimation of MLEgamma :

```
a_hat = 1.9426107289934047 and b_hat = 1.9217064982063528 with nb of iteration : 1
```

1.11 Further initialization of initial points.

Now we test with further initial points, a0 and b0 are the initial point. We can see, further the initial point is, more iteration we will be needed.

```
a_true = 5 and b_true = 4
a0 = 2 and b0 = 2
a_hat = 4.9788003762985715 and b_hat = 3.908935504873119 with nb of iteration : 4
a0 = 2 and b0 = 2
a_hat = 4.973521428021889 and b_hat = 3.904820098115128 with nb of iteration : 5
a0 = 1 and b0 = 1
a_hat = 4.984010697315969 and b_hat = 3.912872326279453 with nb of iteration : 8
a0 = 0.01 and b0 = 0.01
a_hat = 4.984096123330059 and b_hat = 3.912938738937324 with nb of iteration : 13
a0 = 0.01 and b0 = 0.01
a_hat = 4.977806312333816 and b_hat = 3.9080006977208206 with nb of iteration : 19
```

1.12 Modification of function MLEGamma.

We sometimes observe that this produces errors. In fact, it happens that the function UpdateNewton returns negative values for a and/or b (which is not admissible from the point of view of interpreting a and b as parameters of the Gamma law, but Newton's method does not respect the definition domain).

Verifying the value of the estimated a and b, if one of this value are negative, stop the function.

```
if ((a_newton < 0) or (b_newton < 0)):
    print('negative value')
    return
```

1.13 MME.

$$E(X) = \frac{a}{b} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

(1)

$$a = bM_1$$

$$E(X^2) = \frac{a(a+1)}{b^2} = \frac{1}{n}\sum_{i=1}^{n} x_i^2$$

$$M_2 = \frac{M_1}{b} + \frac{b^2M_1^2}{b^2} = \frac{M_1}{b} + M_1^2$$

(2)

$$b = \frac{M_1}{M_2 - M_1^2}$$

From (1) and (2) we have that

$$a = bM_1 = \frac{M_1^2}{M_2 - M_1^2}$$

$$\sigma_x^2 = var(x) = E(X^2) - E(X)^2 = M_2 - M_1$$

As a result we have that:

$$M_1 = E(X) = \overline{x}$$

$$M_1^2 = E(X)^2 = \overline{x}^2$$

We have : $\hat{b} = \frac{\overline{X_n}}{\frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{X_n})^2}$ and $\hat{a} = \frac{\overline{X_n}^2}{\frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{X_n})^2}$

1.14 Implementation of MMEGamma.

Implementation of the function MMEgamma, first we use sig2() to compute $\frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{X_n})^2$ Then the function MMEgamma compute the estimation of a and b.

```
def sig2(x):
  x_m = np.mean(x)
  a = 0
  for i in range(len(x)):
    a = a + x[i]**2 - x_m**2
  return a/len(x)

def MMEgamma(x):
  sigma2 = sig2(x)
  b_hat = np.mean(x)/sigma2
  a_hat = np.mean(x)**2/sigma2
  return a_hat, b_hat
```

1.15 Modification of MLEGamma.

In this part the first guess of the MLEgamma is the guess of the MMEgamma :

```
    a, b = 2, 2
n = 20000
iter = []
a_list = []
b_list = []
coord = []
for i in range(2,6):
  x = np.random.gamma(a, 1/b, 10**i)
  coord.append(10**i)
  a_hat, b_hat = MMEgamma(x)
  result = MLEgamma(x,a_hat,b_hat)
  iter.append(result[2])
  a_list.append(abs(a-result[0]))
  b_list.append(abs(b-result[1]))

plt.plot(coord, a_list, label='a error')
plt.plot(coord, b_list, label='b error')
plt.plot(coord, iter, label='number of iteration (MLE)')
plt.xscale('log')
plt.title('absolute error of the estimation of a and b with different size of n for a_true =
    2 and b_true =2')
plt.legend()
plt.show()
```

Here is the absolute error of a and b for MLE when the first guess is taken by MME:

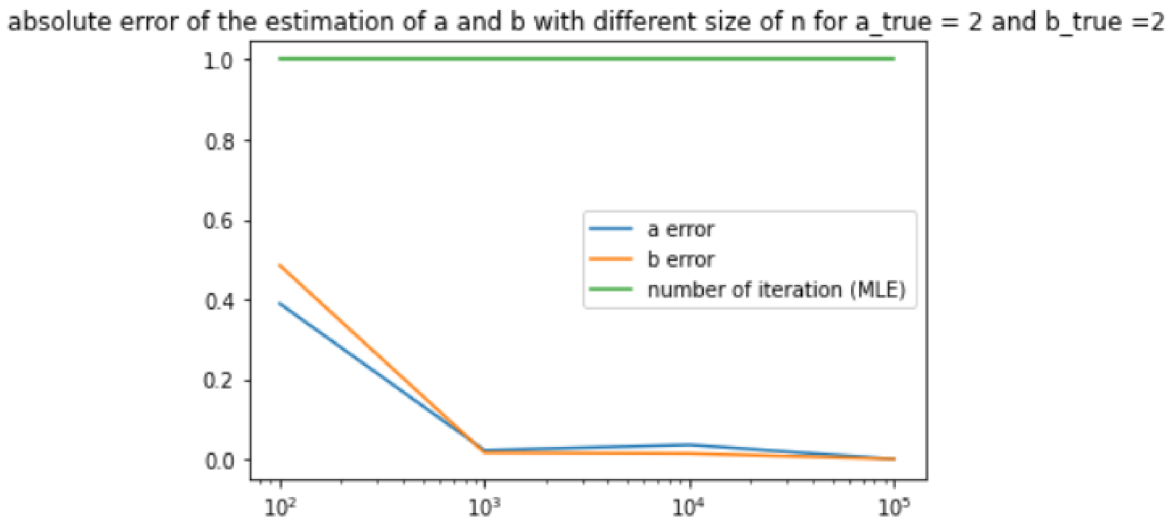

Figure 6: MLE error when n increase

1.16 Comparison of MLE and MME.

First we have created a function to compute the quadratic risk with : $\frac{1}{n}\sum_{i=1}^{n}(a-\hat{a})^2$

```python
def quadratic_risk(true_v, mle_v, mme_v):
    risk_mme = 0
    risk_mle = 0
    for i in range(len(true_v)):
        risk_mme = risk_mme + ((mme_v[i] - true_v[i])**2)
        risk_mle = risk_mle + ((mle_v[i] - true_v[i])**2)
    return [risk_mme/len(true_v), risk_mle/len(true_v)]
```

Then we generate random a and b, and we compute the estimator with the following function:

```python
def gamma_mme_mle(iteration, n):
    a_l = []
    a_mme = []
    a_mle = []
    b_l = []
    b_mme = []
    b_mle = []
    for i in range(iteration):
        a = round(random.uniform(0.5, 10), 1)
        b = round(random.uniform(0.5, 10), 1)
        x = np.random.gamma(a, 1/b, n)
        a_hat_MME, b_hat_MME = MMEgamma(x)
        result = MLEgamma(x,a_hat_MME,b_hat_MME)
        a_l.append(a)
        a_mme.append(a_hat_MME)
        a_mle.append(result[0])
        b_l.append(b)
        b_mme.append(b_hat_MME)
        b_mle.append(result[1])
    return [a_l, a_mme, a_mle, b_l, b_mme, b_mle]
```

This function return the list of the true value of a and b and the estimator of a and b for the MME and MLE method. Then we use these value on the function quadratic risk. And we obtain the quadratic value for a and b for each method (with 50 iteration):

```
    quadratic risk MME for a : 0.009874674055979553 for b : 0.011597847964319707
    quadratic risk MLE for a : 0.009870411839502862 for b : 0.013397908731281091
```

1.17 Estimator of Performance.

Same as the question 1.16 but the n variate from 10^2 to 10^6.

```
for i in range(2,7):
    test = gamma_mme_mle(50, 10**i)
    risk_a = quadratic_risk(test[0], test[1], test[2])
    risk_b = quadratic_risk(test[3], test[4], test[5])
    n_list.append(10**i)
    risk_a_mme.append(risk_a[0])
    risk_a_mle.append(risk_a[1])
    risk_b_mme.append(risk_b[0])
    risk_b_mle.append(risk_b[1])
```

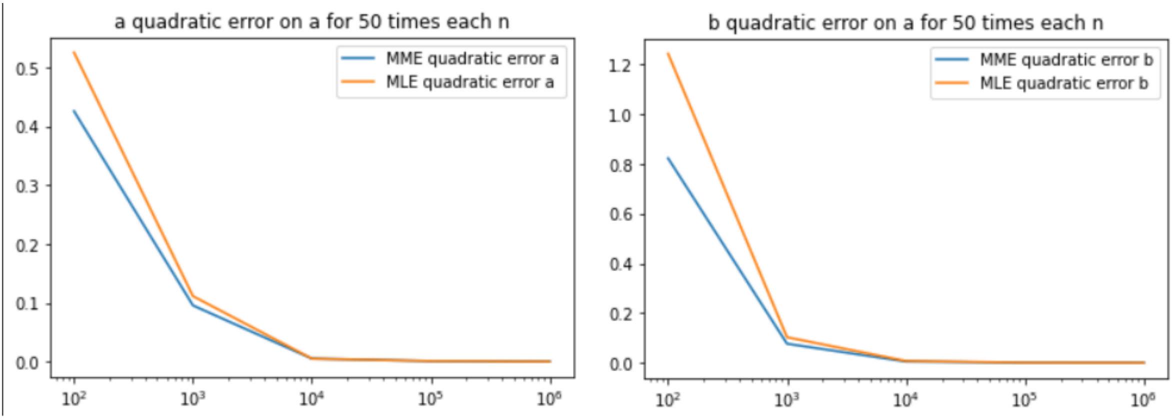

Figure 7: Quadratic risk

The MME perform a bit better when n is low, less than 1000, but when n is greater than 1000 the MLE and MME have the same performance.

2 Simulation of a posterior distribution with the Monte Carlo Markov Chain algorithm.

2.1 Simulation of the MCMC algorithm.

First we define the posterior distribution.

```
k, lamb = 1, 1
def posterior(x, k, theta):
     return stats.gamma(k, scale=theta, loc=0).pdf(x)
```

Then we define a wrapper function containing both the likelihood and the prior as a product.

```
def posterior_estim(x, k, lamb):
    prior = stats.gamma(k, scale=1/lamb, loc=0).pdf(x)
    return stats.poisson.pmf(10, x) * prior
```

And we plot our posterior.

```
x_array = np.linspace(0, 50, 100)
y_array = np.asarray( [posterior(x, k+10, 1/(lamb+1)) for x in x_array] )

plt.plot(x_array,y_array)
plt.grid()
plt.title('Metropolis Hastings Posterior')
plt.show()
plt.close()
```

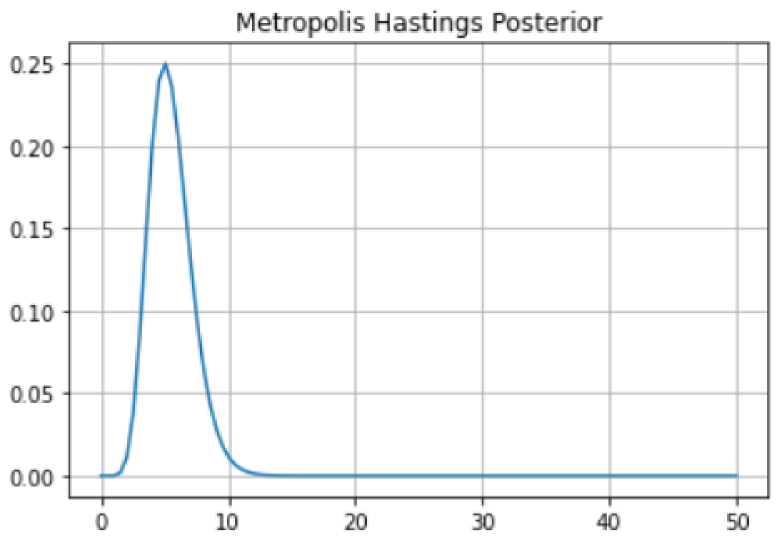

Figure 8: Metropolis Hastings Posterior

Then we create the Metropolis Hastings sampling from the posterior distribution and we plot the samples.

```
N = 100000
s = 10
x = 0
p = posterior_estim(x, k, lamb)
samples = []

for i in range(N):
    xn = x + np.random.normal(size=1)
    pn = posterior_estim(xn,k,lamb)
    if pn >= p:
        p = pn
        x = xn
    else:
        u = np.random.rand()
        if u < pn/p:
            p = pn
            x = xn
    if i % s == 0:
        samples.append(float(x))

samples = np.array(samples)
samples = samples[int(len(samples)/2):]

# Plot Samples
plt.scatter(samples, np.zeros_like(samples), s=10)
plt.plot(x_array,y_array)
plt.hist(samples, bins=50, density=True)
plt.title('Metropolis Hastings sampling')
plt.grid()
plt.show()
plt.close()
```

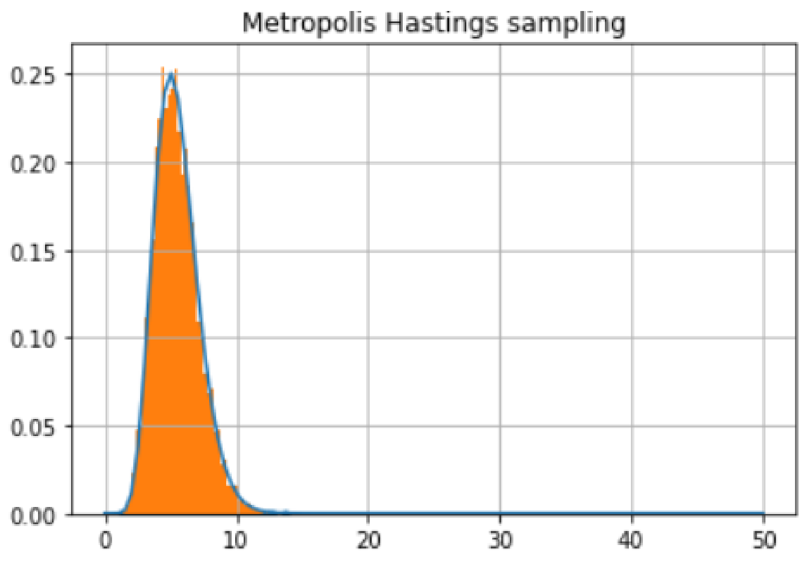

Figure 9: Metropolis Hastings Sampling

2.2 Kernel Approximation.

What we are doing in this part is that we implement different Kernels with scikit-learn library for different bandwidths and we check for which of them our function fits better.

First we check for **Gaussian** kernels.

```
samples = samples.reshape(-1, 1)
create_kde = lambda bw: KernelDensity(kernel="gaussian", bandwidth=bw).fit(samples)
bandwidths = (0.5, 1, 0.1)
x_trunc = x_array[:, np.newaxis]

from sklearn.neighbors import KernelDensity
kdes = [create_kde(bw) for bw in bandwidths]
densities = [np.exp(kde.score_samples(x_trunc)) for kde in kdes]

fig, axs = plt.subplots(3, 1, figsize=(12, 12))
for i, density in enumerate(densities):
    axs[i].plot(x_array, density, "b", alpha=0.7, label=f"Estimation bw = {bandwidths[i]}")
    axs[i].plot(x_array, y_array, "r", alpha=0.7, label="True posterior")
    axs[i].legend()
plt.show()
```

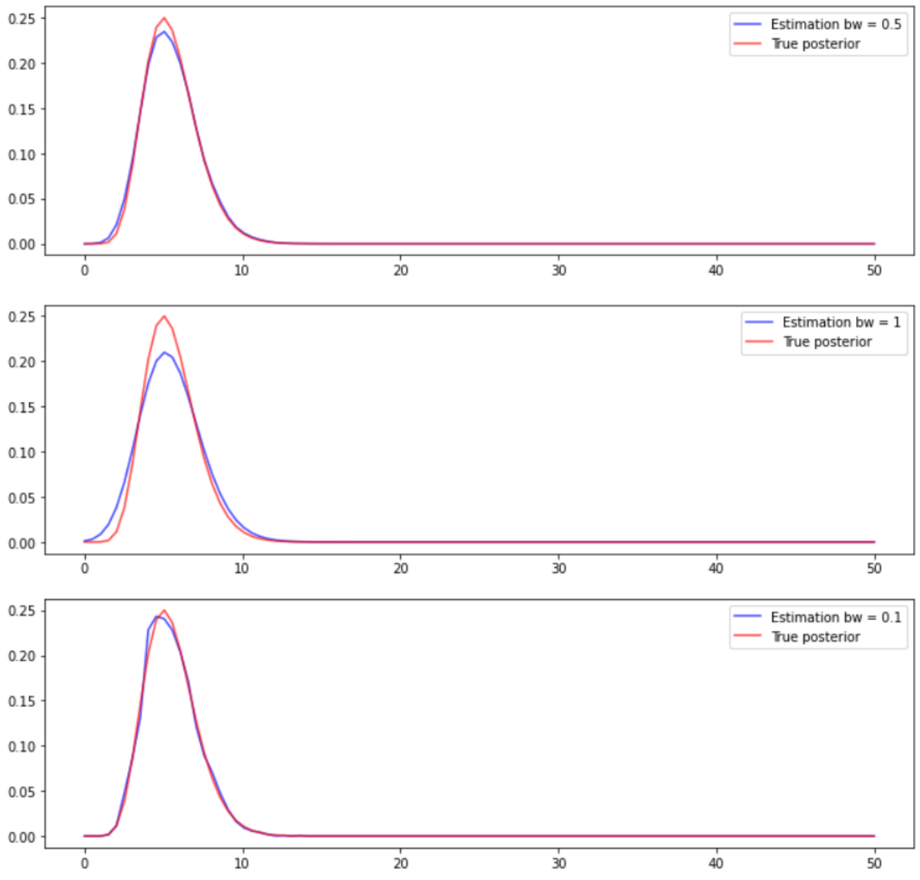

Figure 10: Gaussian Kernels

Then we check for **Epanechikov** kernels.

```
samples = samples.reshape(-1, 1)
create_kde = lambda bw: KernelDensity(kernel="epanechnikov", bandwidth=bw).fit(samples)
bandwidths = (0.5, 1, 0.1)
x_trunc = x_array[:, np.newaxis]

from sklearn.neighbors import KernelDensity
kdes = [create_kde(bw) for bw in bandwidths]
densities = [np.exp(kde.score_samples(x_trunc)) for kde in kdes]

fig, axs = plt.subplots(3, 1, figsize=(12, 12))
for i, density in enumerate(densities):
    axs[i].plot(x_array, density, "b", alpha=0.7, label=f"Estimation bw = {bandwidths[i]}")
    axs[i].plot(x_array, y_array, "r", alpha=0.7, label="True posterior")
    axs[i].legend()
plt.show()
```

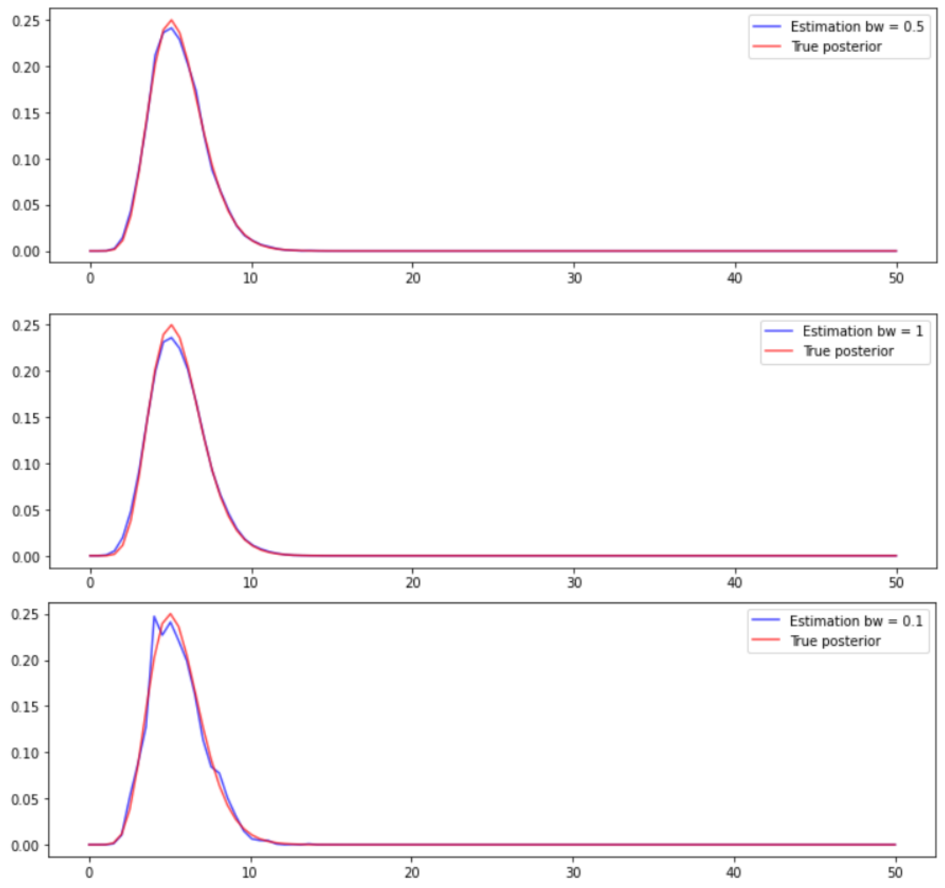

Figure 11: Epanechikov Kernels

First we check for **Linear** kernels.

```
samples = samples.reshape(-1, 1)
create_kde = lambda bw: KernelDensity(kernel="linear", bandwidth=bw).fit(samples)
bandwidths = (0.5, 1, 0.1)
x_trunc = x_array[:, np.newaxis]

from sklearn.neighbors import KernelDensity
kdes = [create_kde(bw) for bw in bandwidths]
densities = [np.exp(kde.score_samples(x_trunc)) for kde in kdes]

fig, axs = plt.subplots(3, 1, figsize=(12, 12))
for i, density in enumerate(densities):
    axs[i].plot(x_array, density, "b", alpha=0.7, label=f"Estimation bw = {bandwidths[i]}")
    axs[i].plot(x_array, y_array, "r", alpha=0.7, label="True posterior")
    axs[i].legend()
plt.show()
```

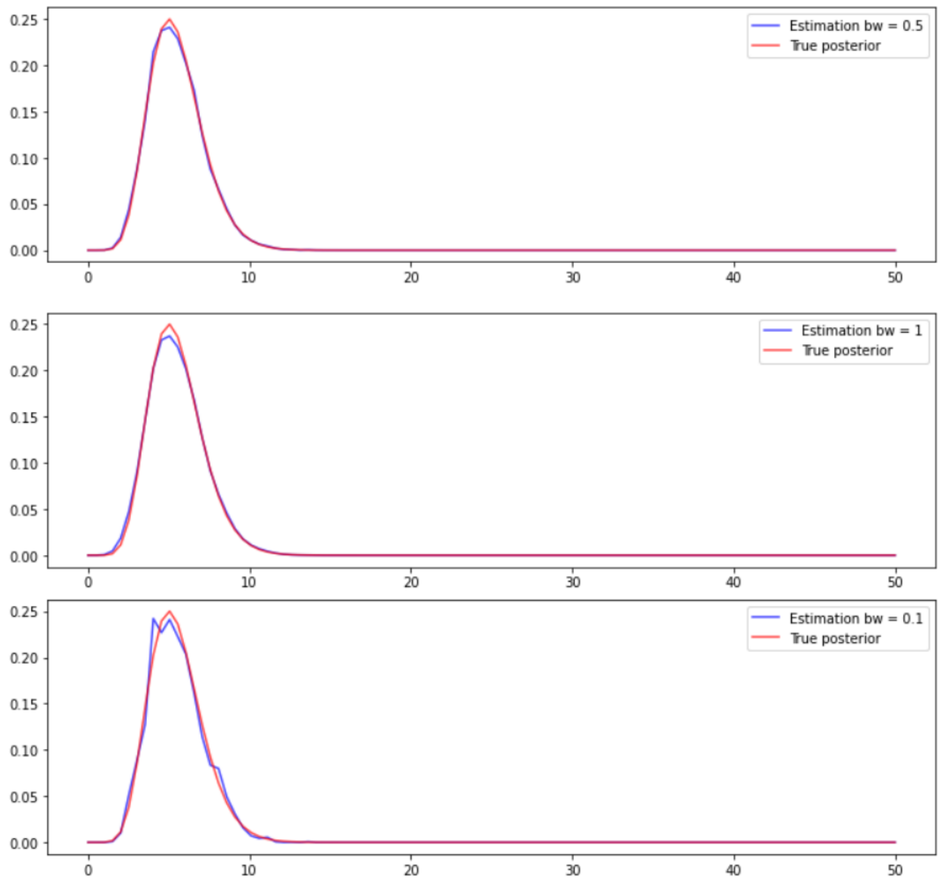

Figure 12: Linear Kernels

- **Gaussian**: We can see for this Kernel the posterior fits better with BW = 0.1.
- **Epanechikov**: We can see for this Kernel the posterior fits better with BW = 0.5.
- **Linear**: We can see for this Kernel the posterior fits better with BW = 0.5.

3 Calibration of the hyper-parameters of a Gaussian process regressor by maximum likelihood.

3.1 Gaussian Process.

3.1.1 Calculation of mean and covariance.

First we calculate the expectation of the the vector:

$$m + \mathbf{A}^T G$$

$$E(m + \mathbf{L}^T G) = E(m) + E(\mathbf{L}^T G) = E(m) + E(\mathbf{L}^T)E(G)$$

We know that :

$$E(m) = m$$

and

$$E(\mathbf{L}^T) = \mathbf{L}^T$$

because they are constants.
Moreover

$$E(G) = 0$$

because G is centered.

As a result:

$$E(m + \mathbf{L}^T G) = m + \mathbf{L}^T E(G) = m$$

For the Covariance we know that:

$$cov(y_i, y_j) = cov(m + [\mathbf{L}^T G)]_i, m + [\mathbf{L}^T G)]_j) = cov(\sum_{p=1}^{n} \bar{l}_{i,p} G_p, \sum_{q=1}^{n} \bar{l}_{i,q} G_q) =$$

$$\sum_{q}^{n} \sum_{p}^{n} cov(\bar{l}_{i,q} G_q, \bar{l}_{j,p} G_p) = \sum_{q}^{n} \sum_{p}^{n} \bar{l}_{i,q} \bar{l}_{j,p} cov(G_q, G_p)$$

We know that

$$cov(G_q, G_p) = \delta_{p,q}$$

Moreover,

$$\delta_{p,q} = \begin{cases} 0, i \neq j \\ 1, i = 1 \end{cases}$$

So we have that

$$\delta_{p,q} = 1$$

and

$$p = q = 1$$

As a result we have that:

$$\sum_{q}^{n} \sum_{p}^{n} \bar{l}_{i,q} \bar{l}_{j,p} = \sum_{q}^{n} \bar{l}_{i,q} \bar{l}_{j,p} = \sum_{q}^{n} \bar{l}_{i,q} l_{j,p} = [L\mathbf{L}^T] = \Sigma_{i,j}$$

3.1.2 Compute $Z = L^T G$

First we have define the function k to compute the $k(i, j)$ and the function kmatrix which build the covariance matrix:

```
def k(x,y,lambda_):
  return ((1 + ( abs(x-y)/lambda_ ) + (( (abs(x-y))**2)/(3*lambda_**2)) ) *
      math.exp(-abs(x-y)/lambda_))

def k_matrix(x,lambda_):
  k_arr = np.zeros((x.size,x.size))
  for i in range(x.size):
    for j in range(x.size):
      k_arr[i, j] = k(x[i], x[j], lambda_)
  return k_arr
```

Then we have define a function, to create the random value x with a uniform distribution and g the random value following a Gaussian distribution. We compute the covariance matrix and the Cholensky decomposition with a function from Numpy, and then we compute Z:

```
def plot_z(n, lambda_, N, lab):
  for i in range(N):
    x = np.random.uniform(0,1,n)
    x = np.sort(x,axis=None)
    km = k_matrix(x, lambda_)
    L = np.linalg.cholesky(km)
    g = np.random.normal(0, 1, n)
    Z = L.T @ g
    plt.plot(x[int(n*0.1):],Z[int(n*0.1):], label = lab)
```

For lambda = 1 and n = 50 we obtain:

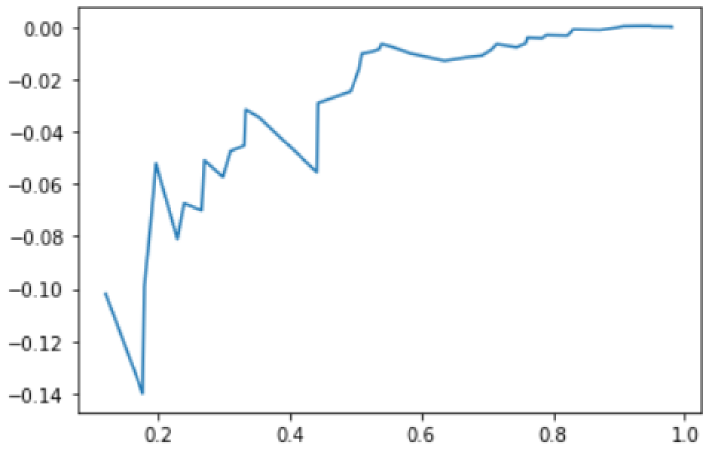

Figure 13: Z in function of x for lambda = 1 and n = 50

3.1.3 Several plot for lambda = 1

Plot of 6 times the function for lambda = 1 and n = 50:

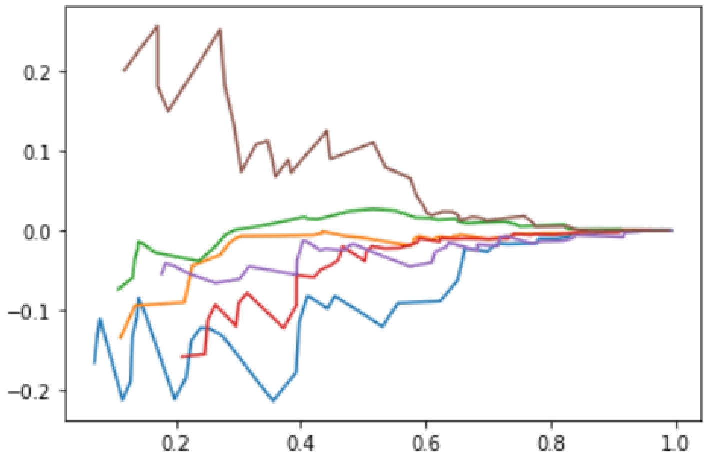

Figure 14: 6 different Z in function of x for lambda = 1 and n = 50

3.1.4 Lambda variation

Now we plot the Z function for n = 25 when lambda is 1, 2 and 3:

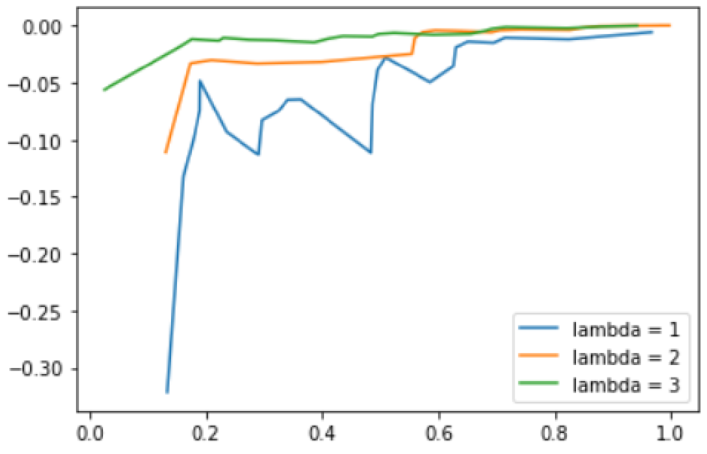

Figure 15: Z in function of x n = 25

When can see when lambda is high the convergence is faster.

3.2 Estimation of λ for MLE.

3.2.1 Generation of x samples.

```
x = np.array([np.random.uniform(i / n, (i + 1) / n) for i in range(n)])
```

3.2.2 Calculation of the negative log-likelihood function and minimization of it.

```
def neg_log_likelihood(lambda_, x, z, n):
    cov = k_matrix(x, lambda_)
    logdetcov = np.linalg.slogdet(cov)[1] # computes the log
    invcov = np.linalg.inv(cov)
    return 0.5 * (n * np.log(2 * math.pi) + logdetcov + z.T @ invcov @ z)

min_lambda = minimize_scalar(neg_log_likelihood, args=(x, z, n))
```

We obtain the minimize lambda:

```
success: True
      x: 3.597513934150737
```

3.2.3 Plotting the function for different .

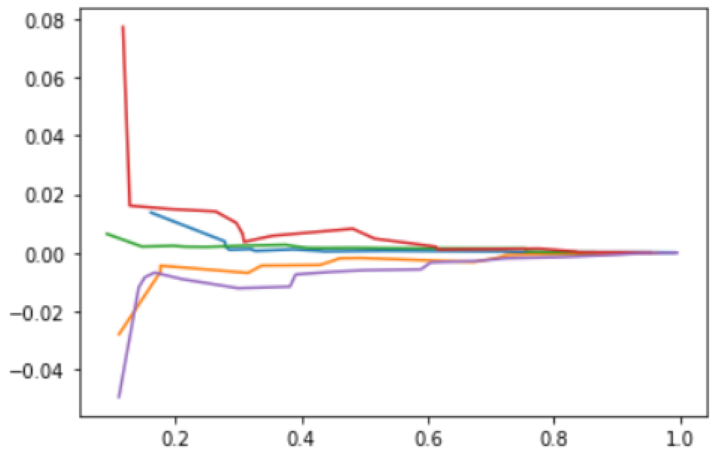

Figure 16: 5 different Z for the minimize lambda

With the minimize lambda the first value of Z are closer to 0 than for the previous lambda.