TP4 Correction

February 14, 2024

1 Réduction de variance et calcul de sensibilités

Sur on problème de modélisation classique en assurance, on illustre l'importance de l'erreur relative puis deux techniques de réduction de variance:

- méthode par préconditionnement,
- échantillonage d'importance (important pour les événements rares).

Dans une deuxième partie on s'intéresse aux sensibilités et comment implémenter efficacement la méthode des différences finies avec la méthode de Monte Carlo.

[1]:
```
import numpy as np
from scipy import stats
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme()
from numpy.random import default_rng
rng = default_rng()
```

1.1 Charge sinistre et loi Poisson-composée

On définit la *charge sinistre totale* (sur une période T) par la variable aléatoire positive

$$S = \sum_{i=1}^{N} X_i$$

où N est une variable aléatoire à valeurs dans $\mathbf{N}$ représentant le nombre de sinistres sur la période T, et pour $i \geq 1$, X_i est une variable aléatoire à valeurs dans $\mathbf{R}_+$ représentant le coût du i-ème sinistre, avec la convention selon laquelle la somme est nulle si $N = 0$. Les $(X_i)_{i\geq 1}$ sont supposées indépendantes et identiquement distribuées, et indépendantes de N (indépendance fréquences - coûts).

Une modélisation classique est de considérer

- N de loi de Poisson de paramètre $\lambda > 0$,
- X_1 de loi log-normale de paramètres $\mu > 0$, $\sigma^2 > 0$, c'est à dire $X_1 = \exp(G_1)$ avec $G_1 \sim \mathcal{N}(\mu, \sigma^2)$.

Le but est d'estimer la **probabilité de dépassement** c'est à dire calculer la probabilité que la charge sinistre totale dépasse un seuil K:

$$p = \mathbf{P}[S > K] \quad \text{pour } K \text{ grand}$$

Dans la suite on prend $\lambda = 10$, $\mu = 0.1$ et $\sigma = 0.3$ et on considère plusieurs valeurs du seuil K.

1.1.1 Question: simulation de la charge sinistre totale

Ecrire une fonction `simu_S(size, mu, sigma, lambd)` qui renvoie un échantillon de taille `size` de réalisations indépendantes de S.

[2]:
```
def simu_S(size, mu, sigma, lambd):
    sample_N = rng.poisson(size=size, lam=lambd)
    sample_S = np.empty(size)
    for k, Nk in enumerate(sample_N):
        sample_S[k] = np.sum(rng.lognormal(size=Nk, mean=mu, sigma = sigma))
    return sample_S
```

[3]:
```
lambd, mu, sigma = 10, 0.1, 0.3
simu_S(10, mu, sigma, lambd)
```

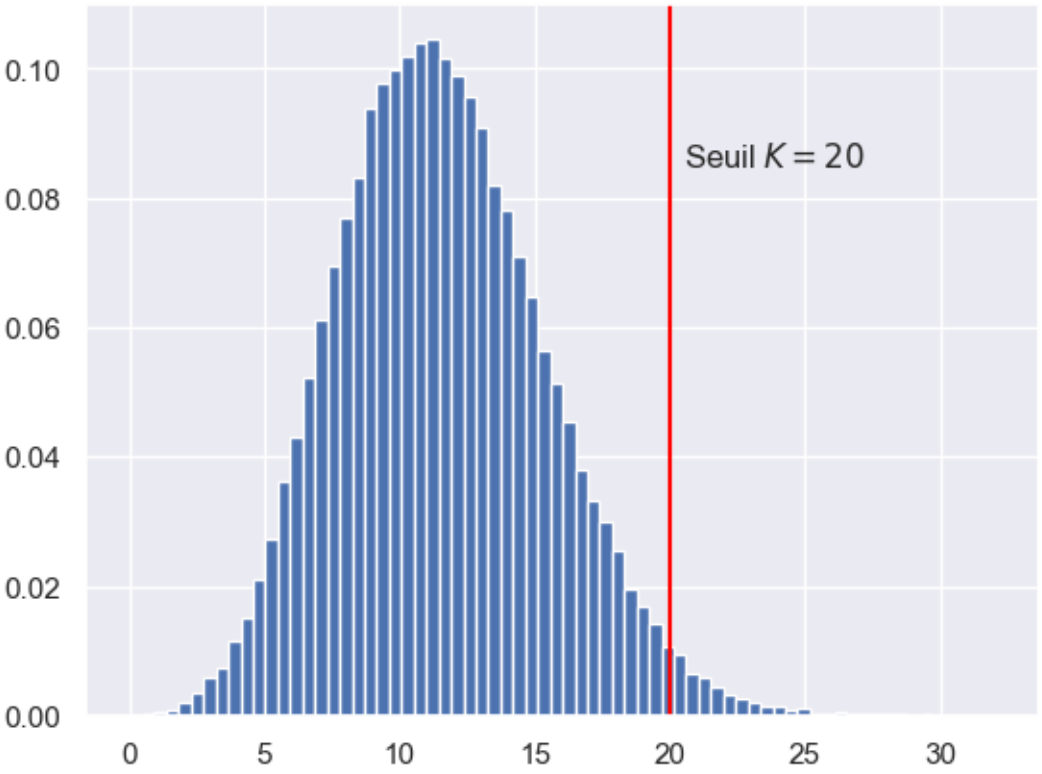
[3]:
```
array([15.2440497 , 10.66101154, 12.82027272,  6.14722746, 11.68993145,
       10.84421946, 13.62459963, 16.3147697 ,  5.7755706 , 18.07679182])
```

1.1.2 Question: représentation graphique

Représenter l'histogramme d'un échantillon de 100 000 réalisations de S et du seuil $K = 20$ par une ligne verticale rouge.

[4]:
```
sample_S = simu_S(int(1e5), mu, sigma, lambd)
K = 20

fig, ax = plt.subplots()
ax.hist(sample_S, bins=70, density=True)
ax.axvline(K, color='red')
ax.text(K+.5, 0.085, fr'Seuil $K={K}$', size=12)
plt.show()
```



1.2 Estimateur Monte Carlo et erreur relative

Soit $p_n = \frac{1}{n}\sum_{j=1}^{n} \mathbf{1}_{S^{(j)}>K}$ l'estimateur Monte Carlo de $p = \mathbf{P}[S > K]$ où $(S^{(j)})_{j=1,\dots,n}$ est une suite *i.i.d.* de même loi que S.

On rappelle que:

- l'**erreur absolue** de l'estimateur Monte Carlo p_n est définie par $|p_n - p|$ et qu'avec probabilité 0.95 cette erreur est bornée par $e_n = 1.96\frac{\sigma_n}{\sqrt{n}}$ avec $\sigma_n^2 = p_n - p_n^2$,
- l'**erreur relative** de l'estimateur Monte Carlo est définie par $\frac{|p_n - p|}{p}$ que l'on majore avec probabilité 0.95 par $\frac{e_n}{p_n}$.

1.2.1 Question: erreur relative

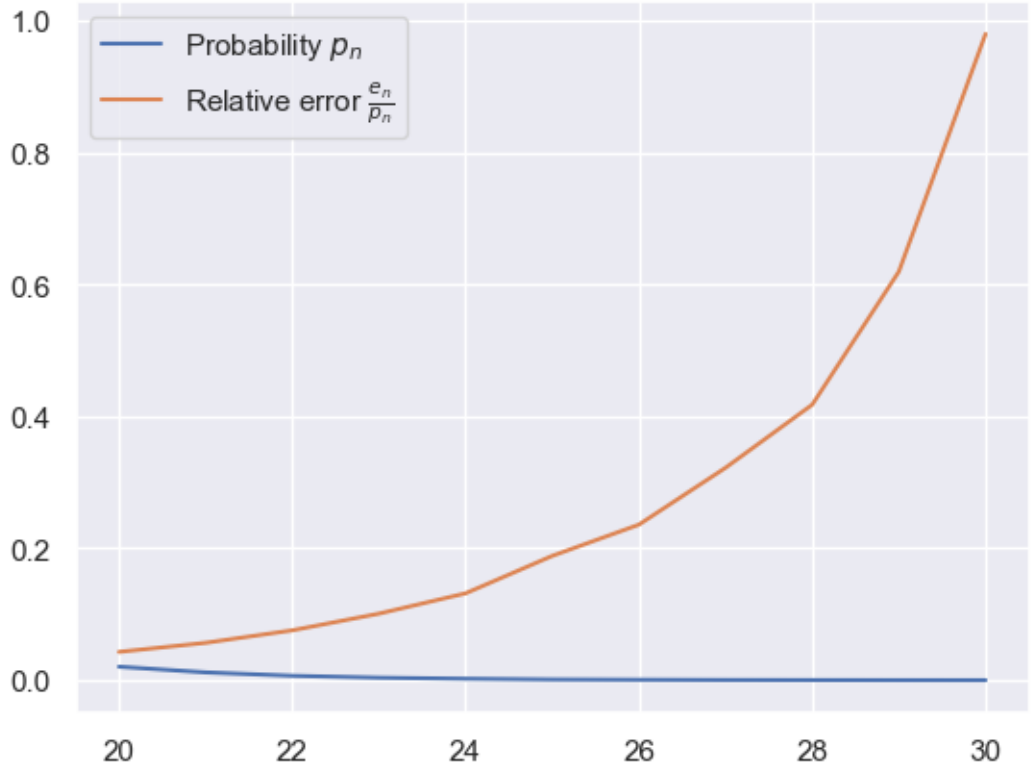
Ecrire une fonction `relative_error` qui à partir d'un échantillon de S (de taille n) et d'une valeur de seuil K renvoie la probabilité p_n et l'erreur relative (plus exactement la borne $\frac{e_n}{p_n}$ à 95%).

Tracer l'erreur relative d'un échantillon de taille 100 000 en fonction de K pour K allant de 20 à 30. Comment interpréter cette courbe?

[5]:
```
def relative_error(sample, K):
    p_n = np.mean(sample > K)
```

```
    v_n = p_n - p_n**2
    e_n = 1.96 * np.sqrt(v_n / sample.size)
    return p_n, e_n / p_n
```

[6]:
```
Ks = np.arange(20, 31)
errors = np.array([relative_error(sample_S, K) for K in Ks])
fig, ax = plt.subplots()
ax.plot(Ks, errors)
ax.legend((r'Probability $p_n$',r'Relative error $\frac{e_n}{p_n}$'))
plt.show()
```



1.2.2 Question: Monte Carlo à précision fixée

Mettre en oeuvre un estimateur de Monte Carlo qui s'arrête dès que l'erreur relative est de 5%. On pourra par exemple introduire la variable aléatoire

$$\tau^{(m)} = \inf\{n \geq 1, e_{nm} \leq 0.05 p_{nm}\},$$

qui dépend d'un paramètre m fixé, par exemple $m = 10\,000$, et renvoyer $p_{\tau^{(m)}}$ ainsi que l'erreur relative et la taille de l'estimateur associé. Le paramètre m permet de recalculer l'estimateur et l'erreur uniquement toutes les m itérations et donc de réduire la complexité par rapport au choix naïf $m = 1$. On appelle ce paramètre m la taille du *batch* (size batch). Le nombre d'itérations (la taille de l'échantillon) dans la méthode de Monte Carlo pour un $\tau^{(m)}$ donné est donc $\tau^{(m)} \times m$.

Définir la fonction qui code cet estimateur Monte Carlo:

`monte_carlo_relative(mu, sigma, lambd, K, size_batch = 10000, error = 0.05)`

[7]:
```
def monte_carlo_relative(mu, sigma, lambd, K, size_batch = 10000, error = 0.05):
    sample_S = simu_S(size_batch, mu, sigma, lambd)
    while True:
        p_n, er_n = relative_error(sample_S, K)
        if er_n < error:
            return p_n, er_n, len(sample_S)
        else:
            new_sample = simu_S(size_batch, mu, sigma, lambd)
            sample_S = np.append(sample_S, new_sample)
```

1.2.3 Question: complexité en fonction de K

Reproduire un tableau de résultat similaire au tableau suivant obtenu avec cet estimateur de Monte Carlo adaptatif jusqu'à l'itération $\tau^{(m)} \times m$ pour une erreur relative de 10% et pour différentes valeurs de $K = 20, \dots, 25$.

Tracer le nombre d'itérations nécessaires en fonction de K.

[8]:
```
import pandas as pd
df = pd.read_pickle("data/iterations_df.pkl")
df
```

[8]:

	Probabilité p_n	Erreur relative	Itérations
20	0.021600	0.093277	20000
21	0.013100	0.098219	30000
22	0.007820	0.098733	50000
23	0.003910	0.098927	100000
24	0.002259	0.099908	170000
25	0.001258	0.099186	310000

[9]:
```
Ks = np.arange(20, 26)
result = np.array([monte_carlo_relative(mu, sigma, lambd, K, error = 0.1) for K␣
 ↪in Ks])
```

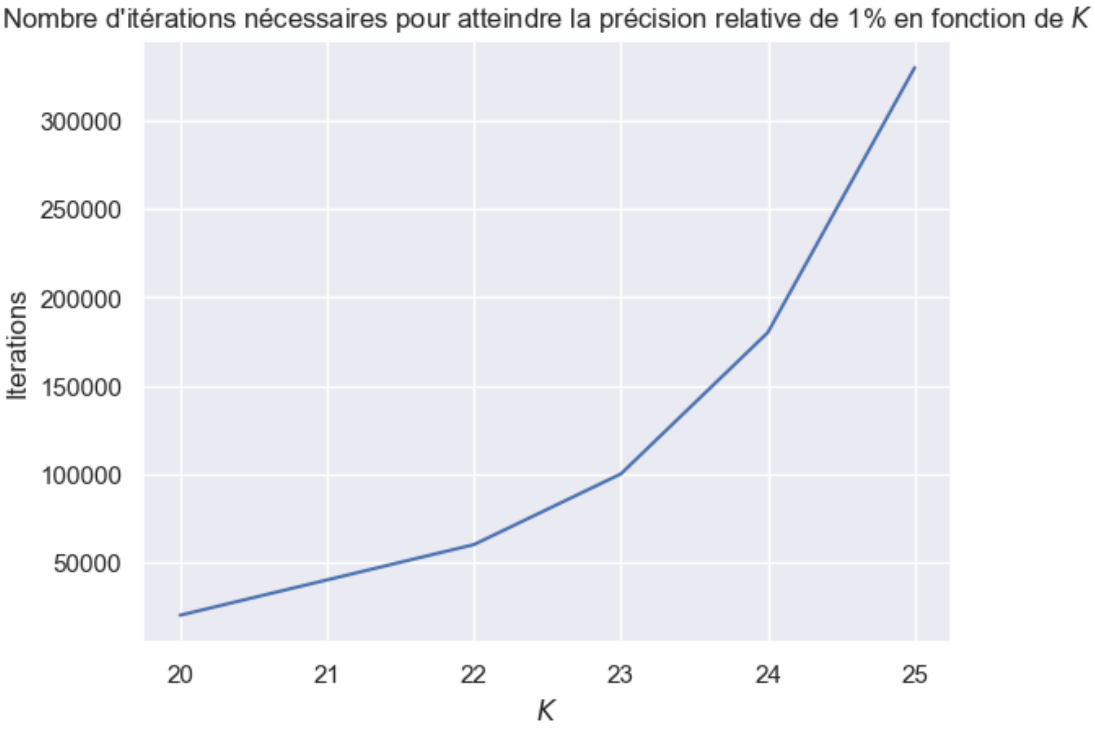
[10]:
```
import pandas as pd
res_df = pd.DataFrame(result,
                      columns=("Probabilité $p_n$", "Erreur relative",␣
 ↪"Itérations"),
                      index = Ks)
res_df = res_df.astype({'Itérations': 'int'})
res_df
#res_df.to_pickle("data/iterations_df.pkl")
```

[10]:

	Probabilité p_n	Erreur relative	Itérations
20	0.022950	0.090429	20000

```
21          0.012250        0.088000       40000
22          0.007283        0.093417       60000
23          0.004210        0.095323      100000
24          0.002233        0.097647      180000
25          0.001197        0.098559      330000
```

[11]:
```
fig, ax = plt.subplots()
ax.plot(Ks, res_df['Itérations'], label="Iterations")
ax.set_xlabel(r"$K$")
ax.set_ylabel(r"Iterations")
ax.set_title(r"Nombre d'itérations nécessaires pour atteindre la précision␣
 ↪relative de 1% en fonction de $K$")
plt.show()
```



1.3 Réduction de variance par préconditionnement

Pour réduire la variance on teste d'abord l'idée présentée dans l'exercice 1 du TD3, c'est à dire qu'on considère la variable aléatoire

$$M = \inf\{r \geq 1, \sum_{i=1}^{r} X_i > K\}$$

et la représentation suivante

$$p = \mathbf{E}[\phi(M)] \quad \text{avec} \quad \phi(m) = \mathbf{P}[N \geq m]$$

1.3.1 Question: simulation de M

Ecrire une fonction `simu_M` similaire à la fonction `simu_S` avec l'argument K supplémentaire qui renvoie un échantillon *i.i.d.* de même loi que M.

[12]:
```
def simu_M(size, mu, sigma, lambd, K):
    def one_M():
        sum_Xi = rng.lognormal(size=1, mean=mu, sigma = sigma)
        r = 1
        while sum_Xi <= K:
            sum_Xi += rng.lognormal(size=1, mean=mu, sigma = sigma)
            r += 1
        return r
    sample_M = np.array([ one_M() for _ in range(size) ])
    return sample_M

# la fonction précédente est correcte mais trop lente car il y a de très
# nombreux appels successifs au générateur de taille size=1 ce qui est
# à éviter. voici une version plus optimisée

def simu_M(size, mu, sigma, lambd, K, batch_size=20):
    def one_M():
        sample_x = np.zeros(1)
        while True:
            sample_x = np.append(sample_x,
                                 rng.lognormal(size=batch_size, mean=mu, sigma
 ↪= sigma))
            r = np.argmax(np.cumsum(sample_x) > K)
            if r > 0: return r
        return r
    sample_M = np.array([ one_M() for _ in range(size) ])
    return sample_M

# exercice: faire un code similaire sans "grossir" sample_x avec np.append
```

1.3.2 Question: Monte Carlo et ratio de variance

En utilisant la fonction `monte_carlo` du TP précédent. Calculer le ratio de variance entre l'estimateur p_n et l'estimateur basé sur la représentation $p = \mathbf{E}[\phi(M)]$ où ϕ est calculée en utilisant la fonction de survie et la fonction de masse de la loi de Poisson (cf. la documentation de `stats.poisson`). Faire ce calcul pour différentes valeurs de K et $n = 20\,000$

[13]:
```
def monte_carlo(sample, proba = 0.95):
    mean = np.mean(sample)
    var = np.var(sample, ddof=1)
    alpha = 1 - proba
    quantile = stats.norm.ppf(1 - alpha/2)  # fonction quantile
    ci_size = quantile * np.sqrt(var / sample.size)
```

```
    return (mean, var, mean - ci_size, mean + ci_size)
```

[14]:
```
Ks = np.arange(20,26)
sample_S = simu_S(int(2e4), mu, sigma, lambd)
result = [ monte_carlo(sample_S > K) for K in Ks ]
import pandas as pd
res_df = pd.DataFrame(result,
                      columns=["mean", "var", "lower", "upper"],
                      index=Ks)
res_df
```

[14]:

	mean	var	lower	upper
20	0.02255	0.022043	0.020492	0.024608
21	0.01315	0.012978	0.011571	0.014729
22	0.00795	0.007887	0.006719	0.009181
23	0.00450	0.004480	0.003572	0.005428
24	0.00255	0.002544	0.001851	0.003249
25	0.00140	0.001398	0.000882	0.001918

[15]:
```
def phi(m, lambd = 10):
    N = stats.poisson(mu = lambd)
    return N.pmf(m) + N.sf(m)
```

[16]:
```
result = []
for K in Ks:
    sample_M = simu_M(int(2e4), mu, sigma, lambd, K)
    result.append(monte_carlo(phi(sample_M)))
import pandas as pd
res_df_precond = pd.DataFrame(result,
                      columns=["mean", "var", "lower", "upper"],
                      index=Ks)
res_df_precond
```

[16]:

	mean	var	lower	upper
20	0.021298	0.000354	0.021037	0.021559
21	0.012581	0.000153	0.012409	0.012752
22	0.007132	0.000063	0.007021	0.007242
23	0.003990	0.000024	0.003922	0.004058
24	0.002184	0.000009	0.002142	0.002226
25	0.001149	0.000003	0.001125	0.001173

[17]:
```
res_df["var"] / res_df_precond["var"]
```

[17]:
```
20     62.228610
21     84.772280
22    124.403484
23    186.071356
```

```
24    280.463358
25    460.006198
Name: var, dtype: float64
```

1.4 Réduction de variance par échantillonage d'importance

Pour réduire la variance sans faire exploser la complexité pour les grandes valeurs de K on propose une méthode d'échantillonage d'importance (Importance Sampling) en modifiant la loi de la variable aléatoire N (on peut faire un autre choix, en changeant la loi des X_i ou bien en changeant la loi de N et des X_i). Le changement de loi proposé ici repose sur le changement de probabilité, pour $\theta \in \mathbf{R}$

$$\frac{\mathrm{d}\mathbf{P}}{\mathrm{d}\mathbf{P}_\theta} = L_\theta \quad \text{avec} \quad L_\theta = \exp(-\theta N + \psi(\theta)),$$

où $\psi(\theta) = \log \mathbf{E}[\exp(\theta N)] = \lambda(e^\theta - 1)$. On vérifie par le calcul que la loi de N sous $\mathbf{P}_\theta$ est la loi de Poisson de paramètre $\tilde{\lambda} = \lambda e^\theta$. Ainsi on a la représentation

$$\mathbf{P}\left[\sum_{i=1}^{N} X_i > K\right] = \mathbf{E}_{\mathbf{P}_\theta}\left[\mathbf{1}_{\sum_{i=1}^{N} X_i > K} \exp\big(-\theta N + \psi(\theta)\big)\right] \quad \text{avec } N \sim \mathcal{P}(\tilde{\lambda}) \text{ sous } \mathbf{P}_\theta.$$

Il est d'usage pour la loi de Poisson d'écrire la variable L_θ à partir de λ et $\tilde{\lambda}$ (la valeur du paramètre de la loi de Poisson sous la nouvelle probabilité) *i.e.*

$$L_\theta = \exp\big(-\theta N + \lambda(e^\theta - 1)\big) = \left(\frac{\lambda}{\tilde{\lambda}}\right)^N \exp\big(\tilde{\lambda} - \lambda\big).$$

1.4.1 Question: simulation sous $\mathbf{P}_\theta$

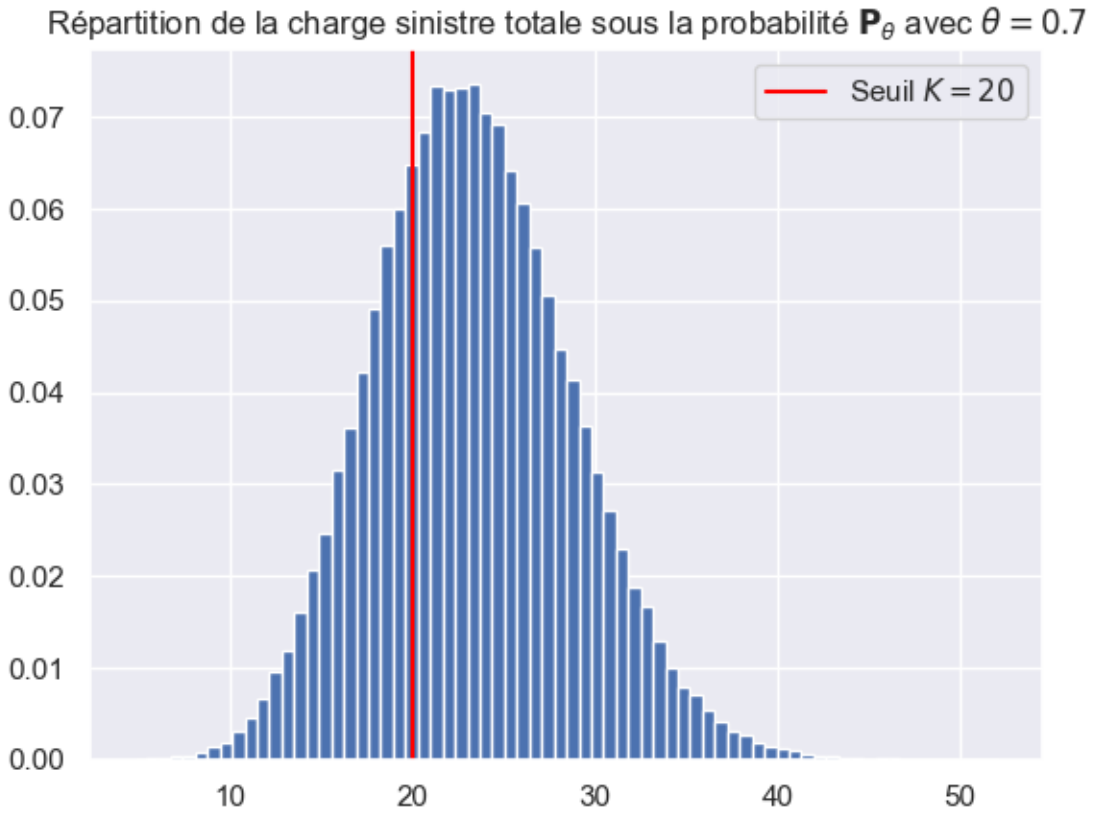
La loi de N sous $\mathbf{P}_\theta$ est la loi de Poisson de paramètre $\tilde{\lambda} = \lambda e^\theta$ et la suite $(X_i)_{i\geq 1}$ est indépendante de N donc de L_θ et n'est donc pas impactée par le changement de probabilité: la loi des $(X_i)_{i\geq 1}$ est inchangée.
Ecrire une fonction `simu_S_tilde` inspirée de `simu_S` qui prend un paramètre supplémentaire θ et qui renvoie un échantillon de $\sum_{i=1}^{N} X_i$ sous $\mathbf{P}_\theta$.

[18]:
```python
def simu_S_tilde(size, mu, sigma, lambd, theta):
    sample_N_tilde = rng.poisson(size=size, lam=lambd * np.exp(theta))
    sample_sum = np.empty(size)
    for k, Nk in enumerate(sample_N_tilde):
        sample_sum[k] = np.sum(rng.lognormal(size=Nk, mean=mu, sigma = sigma))
    return sample_sum
```

[19]:
```python
theta = 0.7
sample_S = simu_S_tilde(int(1e5), mu, sigma, lambd, theta=theta)
K = 20

fig, ax = plt.subplots()
ax.hist(sample_S, bins=70, density=True)
ax.axvline(K, color='red', label=fr'Seuil $K={K}$')
```

```
ax.set_title(fr"Répartition de la charge sinistre totale sous la probabilité␣
 ↪$\mathbf{{P}}_\theta$ avec $\theta = {theta}$")
ax.legend()
plt.show()
```



1.4.2 Question: Monte Carlo sous $\mathbf{P}_\theta$

Comparer pour différentes valeurs de K, avec $\theta = 0.7$, l'estimateur de Monte Carlo basé sur la représentation

$$\mathbf{P}\left[\sum_{i=1}^{N} X_i > K\right] = \mathbf{E}\left[\mathbf{1}_{\sum_{i=1}^{\tilde{N}} X_i > K} \left(\frac{\lambda}{\tilde{\lambda}}\right)^{\tilde{N}} \exp(\tilde{\lambda} - \lambda)\right] \quad \text{avec } \tilde{N} \sim \mathcal{P}(\tilde{\lambda}).$$

Pour $K = 22$ le ratio de variance est de l'ordre de 16-17.

Que se passe-t-il si le paramètre θ est mal choisi? (prendre par exemple $\theta = 1.2$ puis $\theta = 1.5$, et $\theta = -0.1$...)

[20]:
```
# on modifie un peu la fonction précédente car on a besoin de N_tilde !
def simu_sous_P_tilde(size, mu, sigma, lambd, theta):
    sample_N_tilde = rng.poisson(size=size, lam=lambd * np.exp(theta))
    sample_sum = np.empty(size)
```

```
    for k, Nk in enumerate(sample_N_tilde):
        sample_sum[k] = np.sum(rng.lognormal(size=Nk, mean=mu, sigma = sigma))
    return sample_sum, sample_N_tilde
```

[21]:
```
Ks = np.arange(20,26)
theta = 0.7

lambd_tilde = lambd * np.exp(theta)
sample_S_tilde, sample_N_tilde = simu_sous_P_tilde(int(2e4), mu, sigma, lambd,␣
 ↪theta)
result = []
for K in Ks:
    sample = (sample_S_tilde > K) * (lambd / lambd_tilde)**sample_N_tilde * np.
 ↪exp(lambd_tilde - lambd)
    result.append(monte_carlo(sample))

import pandas as pd
res_df_is = pd.DataFrame(result,
                         columns=["mean", "var", "lower", "upper"],
                         index=Ks)
res_df_is
```

[21]:

	mean	var	lower	upper
20	0.021564	0.003435	0.020751	0.022376
21	0.012632	0.001283	0.012135	0.013128
22	0.007246	0.000431	0.006958	0.007534
23	0.004111	0.000145	0.003944	0.004278
24	0.002234	0.000053	0.002134	0.002335
25	0.001188	0.000018	0.001129	0.001246

[22]:
```
res_df["var"] / res_df_is["var"]
```

[22]:
```
20     6.417963
21    10.112129
22    18.303563
23    30.890552
24    48.349999
25    77.232698
Name: var, dtype: float64
```

1.5 Calcul de sensibilités

On utilisera la notation $S^{(\lambda)}$ pour indiquer la dépendance de variable aléatoire $S = \sum_{i=1}^{N} X_i$ en le paramètre $\lambda > 0$ (paramètre de la loi de Poisson sous-jacente). On s'intéresse à la sensibilité de la probabilité p en fonction de lambda c'est à dire

$$\frac{\partial}{\partial \lambda} p(\lambda) = \frac{\partial}{\partial \lambda} \mathbf{P}[S^{\lambda} > K]$$

1.5.1 Différences finies

Implémenter l'estimateur Monte Carlo basé sur les différences finies d'ordre 2

$$\frac{\partial}{\partial\lambda}p(\lambda) = \frac{p(\lambda+h)-p(\lambda-h)}{2h} + \mathcal{O}(h^2)$$

Comme vu en cours, il y a plusieurs façon d'implémenter l'estimateur Monte Carlo dans ce cadre biaisé.

- Le premier estimateur naïf $J^{(1)}_{n,h}(\lambda)$ est basé sur des réalisations indépendantes de $S^{(\lambda+h)}$ et $S^{(\lambda-h)}$ et n'est pas efficace: la variance explose lorsque h tend vers 0. Ainsi on pose

  $$J^{(1)}_{n,h}(\lambda) = \frac{1}{2hn}\Big(\sum_{k=1}^{n}\mathbf{1}_{\{S_k^{(\lambda+h)}>K\}} - \sum_{k=1}^{n}\mathbf{1}_{\{\tilde{S}_k^{(\lambda-h)}>K\}}\Big),$$

  où $(S_k^{(\lambda+h)})_{k\geq1}$ et $(\tilde{S}_k^{(\lambda-h)})_{k\geq1}$ sont des suites indépendantes de variables aléatoires *i.i.d.*.

- Le deuxième estimateur $J^{(2)}_{n,h}(\lambda)$ utilise des réalisations fortements corrélées de la loi de Poisson au sens suivant: on utilise la même réalisation uniforme U pour constuire deux réalisations $N^{(\lambda+h)}$ et $N^{(\lambda-h)}$ en utilisant la méthode de l'inverse de la fonction de répartition. Dans ce deuxième estimateur, les lois log-normales sont indépendantes. On a donc

  $$J^{(2)}_{n,h}(\lambda) = \frac{1}{2hn}\sum_{k=1}^{n}\Big(\mathbf{1}_{\{S_k^{(\lambda+h)}>K\}} - \mathbf{1}_{\{\bar{S}_k^{(\lambda-h)}>K\}}\Big),$$

  où pour $k\geq1$, $S_k^{(\lambda+h)} = \sum_{i=1}^{G(\lambda+h,U_k)} X_{i,k}$ et $\bar{S}_k^{(\lambda-h)} = \sum_{i=1}^{G(\lambda-h,U_k)} \bar{X}_{i,k}$ avec $G(\lambda,u)$ l'inverse généralisée de la loi de Poisson de paramètre λ, $(U_k)_{k\geq1}$ suite *i.i.d.* uniforme sur $[0,1]$ indépendante de $(X_{i,k})_{i\geq1,k\geq1}$ et $(\bar{X}_{i,k})_{i\geq1,k\geq1}$ deux suites (doublement indicées) *i.i.d.* de loi log-normale (de paramètres μ et σ inchangés).

- Un troisième estimateur $J^{(3)}_{n,h}(\lambda)$ utilise des réalisations fortements corrélées de la loi de Poisson et des variables aléatoires log-normales communes.

  $$J^{(3)}_{n,h}(\lambda) = \frac{1}{2hn}\sum_{k=1}^{n}\Big(\mathbf{1}_{\{S_k^{(\lambda+h)}>K\}} - \mathbf{1}_{\{S_k^{(\lambda-h)}>K\}}\Big),$$

  où pour $k\geq1$, $S_k^{(\lambda+h)} = \sum_{i=1}^{G(\lambda+h,U_k)} X_{i,k}$ et $S_k^{(\lambda-h)} = \sum_{i=1}^{G(\lambda-h,U_k)} X_{i,k}$ avec $G(\lambda,u)$ l'inverse généralisée de la loi de Poisson de paramètre λ, $(U_k)_{k\geq1}$ suite *i.i.d.* uniforme sur $[0,1]$ indépendante de $(X_{i,k})_{i\geq1,k\geq1}$ une suite (doublement indicée) *i.i.d.* de loi log-normale.

1.5.2 Question: plusieurs estimateurs des différences finies

On fixe les paramètres $\lambda = 10$, $\mu = 0.1$, $\sigma = 0.3$ et $K = 20$. Programmer ces 3 estimateurs pour différentes valeurs de h (par exemple, $h = 1$, 0.5, 0.1 et 0.01), et donner le résultat des estimateurs Monte Carlo avec $n = 50\,000$.

Que se passe-t-il lorsque h tend vers 0? Comparez le comportement pour ces 3 estimateurs. Il est très important de bien interpréter ces tableaux de résultats et de conclure qu'il faut utiliser l'estimateur $J^{(3)}_{n,h}(\lambda)$ et en aucun cas l'estimateur $J^{(1)}_{n,h}(\lambda)$.

Remarque: on considère ici uniquement l'étude de l'erreur statistique dûe à la méthode de Monte Carlo. On ne considère pas l'erreur de biais qui décroît lorsque h tend vers 0 et qui est peut-être non négligeable pour $h = 1$. Les IC construits ici sont biaisés et on ne peut pas affirmer que la vraie valeur est dans l'IC à 95% (au moins pour les grandes valeurs de h).

[23]:
```
lambd, mu, sigma = 10, 0.1, 0.3
K = 20
```

[24]:
```
def J1(n, h):
    sample_Sph = simu_S(n, mu, sigma, lambd+h)
    sample_Smh = simu_S(n, mu, sigma, lambd-h)
    xph = (sample_Sph > K).astype(int)
    xmh = (sample_Smh > K).astype(int)
    return monte_carlo((xph - xmh)/(2*h))
```

[25]:
```
# on crée une fonction pour ne pas répéter ces lignes de code
def result_estimator(J, n=50000, hs=[1, 0.5, 0.1, 0.01]):
    result = [ J(n, h) for h in hs ]
    df = pd.DataFrame(result,
                      columns=['mean', 'var', 'lower', 'upper'],
                      index=hs)
    return df
```

[26]:
```
result_estimator(J1)
```

[26]:

	mean	var	lower	upper
1.00	0.01679	0.012323	0.015817	0.017763
0.50	0.01754	0.044673	0.015687	0.019393
0.10	0.01860	1.023675	0.009732	0.027468
0.01	0.06700	102.447560	-0.021718	0.155718

[27]:
```
def J2(n, h):
    sample_U = rng.random(size=n)
    sample_Nph = stats.poisson(mu = lambd+h).ppf(sample_U)
    sample_Nmh = stats.poisson(mu = lambd-h).ppf(sample_U)
    sample_Sph = np.empty(n)
    sample_Smh = np.empty(n)
    for k, (Nph, Nmh) in enumerate(zip(sample_Nph, sample_Nmh)):
        sample_Sph[k] = np.sum(rng.lognormal(size=int(Nph), mean=mu, sigma =␣
 ↪sigma))
        sample_Smh[k] = np.sum(rng.lognormal(size=int(Nmh), mean=mu, sigma =␣
 ↪sigma))
    xph = (sample_Sph > K).astype(int)
    xmh = (sample_Smh > K).astype(int)
    return monte_carlo((xph - xmh)/(2*h))
```

[28]:
```
result_estimator(J2)
```

[28]:

	mean	var	lower	upper
1.00	0.01819	0.009164	0.017351	0.019029
0.50	0.01622	0.022117	0.014916	0.017524
0.10	0.02160	0.470543	0.015587	0.027613
0.01	0.04600	43.498754	-0.011810	0.103810

[29]:
```
def J3(n, h):
    sample_U = rng.random(size=n)
    sample_Nph = stats.poisson(mu = lambd+h).ppf(sample_U).astype(int)
    sample_Nmh = stats.poisson(mu = lambd-h).ppf(sample_U).astype(int)
    sample_Sph = np.empty(n)
    sample_Smh = np.empty(n)
    for k, (Nph, Nmh) in enumerate(zip(sample_Nph, sample_Nmh)):
        max_N = max(Nph, Nmh)
        sample_X = rng.lognormal(size=max_N, mean=mu, sigma = sigma)
        sample_Sph[k] = np.sum(sample_X[:Nph])
        sample_Smh[k] = np.sum(sample_X[:Nmh])
    xph = (sample_Sph > K).astype(int)
    xmh = (sample_Smh > K).astype(int)
    return monte_carlo((xph - xmh)/(2*h))
```

[30]:
```
result_estimator(J3)
```

[30]:

	mean	var	lower	upper
1.00	0.01672	0.008081	0.015932	0.017508
0.50	0.01622	0.015957	0.015113	0.017327
0.10	0.01680	0.083719	0.014264	0.019336
0.01	0.01400	0.699818	0.006667	0.021333