PROJET DE STATISTIQUE NON PARAMÉTRIQUE

MASTER 2 INGÉNIERIE MATHÉMATIQUE ET BIOSTATISTIQUE
UNIVERSITÉ DE PARIS
BODIAN Pierre Denis, ELION Lessy-Lafoi, LOUIS James Kelson

Le but de ce projet est de simuler l'estimateur étudié dans le CC1.

On dispose d'un échantillon $X = (X_1, \dots, X_n)$ ***de variables aléatoires indépendantes et de même loi (i.i.d.) positives, de densité commune f inconnue et de fonction de répartition*** F, $F(x) = \mathbb{P}(X_1 \geqslant x)$ ***inconnue.***

On appelle fonction de survie la fonction S ***définie par*** $S(x) = \mathbb{P}(X_1 > x) = 1 - F(x)$.

On se propose d'estimer, sur un intervalle $[0, b]$ ***pour*** $b > 0$ ***fixé par l'utilisateur, la fonction de risque instantané***

In [339]:
```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
import warnings
warnings.filterwarnings('ignore')
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as st
from scipy.stats import norm
import pandas as pd
import random
%matplotlib inline
```

Partie I: Coder l'estimateur de la survie empirique

L'estimateur de la fonction de survie est donnée par:

$$\hat{S}_n(x) = \frac{1}{n} \sum_{i=1}^{n} 1_{\{X_i > x\}}$$

Question_1-)

Ecrivons une fonction *S_hat* qui prend en argument un échantillon X et une grille de points *grid* et renvoie la valeur de $\hat{S}_n$ aux points de la grille $grid$.

In [340]:
```
def hat_S(X,grid):

    X=np.array(X); # on transforme X en array pour utiliser les méthodes des arrays en Python
    grid=np.array(grid)
    n=len(X); # la taille de l'echantillon X
    nb=len(grid) # la taille de la grille de points
    Sn_chapo=np.zeros(nb) # le vecteur rempli de zéros de taille la grille de points
    for j in range(len(grid)):
        d = X > grid[j] # la fonction indicatrice de Xi > x
        Sn_chapo[j]=np.mean(d) # calcule de la moyenne de d
    return Sn_chapo
```

In [ ]:
```
```

Question_2-) tracer sur un même graphique la fonction S et son estimée.

Pour $n = 1000$ et $f(x) = \lambda e^{-\lambda x} 1_{\{x>0\}}$ traçons sur un même graphique la fonction S et son estimée $\hat{S}_n$.

$S(x) = 1 - F(x)$

$F(t) = \int_{-\infty}^{t} f(x)\mathrm{d}x = \int_{0}^{t} f(x)\mathrm{d}x = \left(1 - e^{-\lambda t}\right) \times 1_{\mathrm{R}_+^*}(t)$

$S(x) = e^{-\lambda x} 1_{\{\mathbb{R}_+^*\}}(x)$ si $x > 0$ et 1 sinon

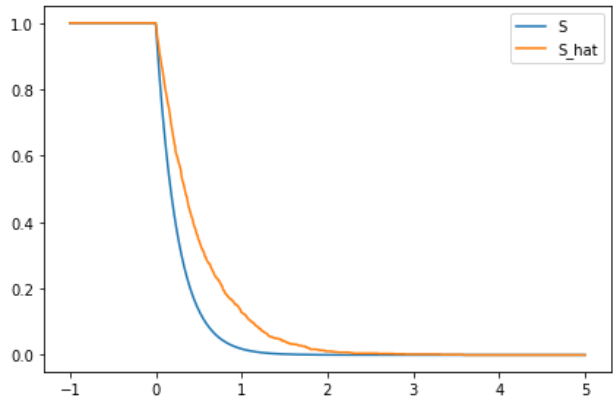
In [341]:
```
random.seed(30)
n=1000 # la taille de l'échantillon

grid = np.linspace(-1, 5, n)  # définition de la grille de points

Lambda=4 # le paramètre de loi exponentielle, on choisit au hasard Lambda=4

S=lambda x: np.exp(-Lambda*x) if x>0 else 1 # définition de la fonction S, égale à
#la e^{-lambda * x}  pour x > 0 ou égale 1 sinon
y=[i for i in map(S, grid)]  ## Vecteur qui renferme les images par S evaluees sur grid
```

In [342]:
```
# traçage de s et de son estimée.

fig, ax = plt.subplots()

n = 1000

X = st.expon.rvs(0,0.5,n)

ax.plot(grid, y,label="S")
ax.plot(grid,hat_S(X,grid),label="S_hat")
ax.legend()

fig.tight_layout()
fig.show()
```

Out[342]:
```
[<matplotlib.lines.Line2D at 0x1fc9ca4eb08>]
```

Out[342]:
```
[<matplotlib.lines.Line2D at 0x1fc9b143a48>]
```

Out[342]:
```
<matplotlib.legend.Legend at 0x1fc9c7b04c8>
```



In [343]:
```
# on affiche les courbes pour différentes valeurs de lambda
fig, ax = plt.subplots()
ax.plot(grid,hat_S(X,grid),label="S_hat")
for Lambda in [0.5, 1, 2,4]:
    y=[i for i in map(S, grid)]
    ax.plot(grid, y,label="lambda="+str(Lambda))
ax.legend()

fig.tight_layout()
fig.show()
```

Out[343]: [<matplotlib.lines.Line2D at 0x1fc9ca26f88>]

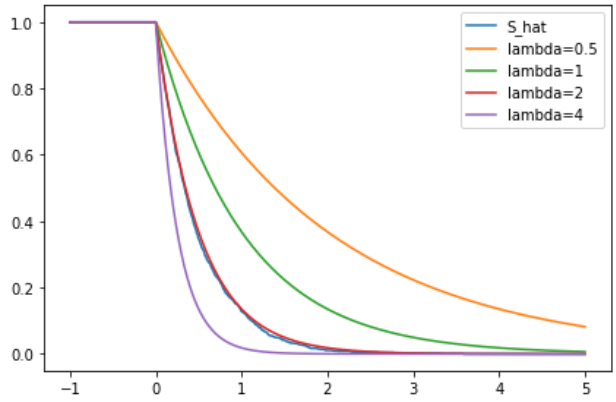
Out[343]: [<matplotlib.lines.Line2D at 0x1fc9ca31f48>]

Out[343]: [<matplotlib.lines.Line2D at 0x1fc9ca3be48>]

Out[343]: [<matplotlib.lines.Line2D at 0x1fc9ca3eec8>]

Out[343]: [<matplotlib.lines.Line2D at 0x1fc9cc0a708>]

Out[343]: <matplotlib.legend.Legend at 0x1fc9ca31d08>



Nous allons calculer l'erreur quadratique pour différentes valeurs de lambda afin de sélectionner celle qui minimise l'erreur

In [344]:
```
z=hat_S(X,grid)
Erreur=[]
for Lambda in np.linspace(-5,10,1000):
    y=np.array([i for i in map(S, grid)])
    Erreur.append(np.sqrt(np.sum((z-y)**2)))
min_err=np.argmin(Erreur)
```

In [345]:
```
# Le minimum est atteint en min_err  pour Lambda= Lambda
Lambda=np.linspace(-5,10,1000)[min_err]
min_err
Lambda
```

Out[345]: 471

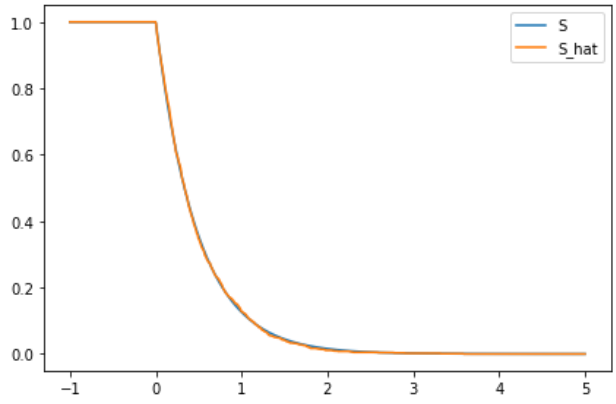
Out[345]: 2.0720720720720722

In [346]:
```
# on affiche les graphiques de S et S_hat pour Lambda
Lambda=np.linspace(-5,10,1000)[min_err]
y=np.array([i for i in map(S, grid)])
fig, ax = plt.subplots()
ax.plot(grid, y,label="S")
ax.plot(grid,hat_S(X,grid),label="S_hat")
ax.legend()

fig.tight_layout()
fig.show()
```

Out[346]:
```
[<matplotlib.lines.Line2D at 0x1fc9c9ad088>]
```

Out[346]:
```
[<matplotlib.lines.Line2D at 0x1fc9adb4788>]
```

Out[346]:
```
<matplotlib.legend.Legend at 0x1fc9cb36748>
```



On constate que la fonction hat_S estime bien la fonction S

In [ ]:
```
```

Partie 2: Etude d'un estimateur de f

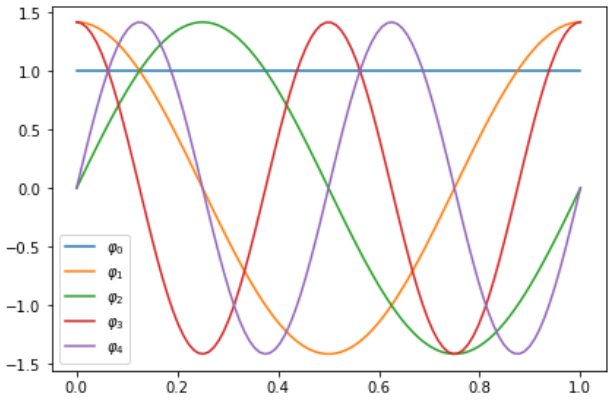
Question_3)

In [347]:
```
def func_trig_tilde(d,b,x):


# Coder une fonction qui prend en argument un entier d, un réel b > 0 et une grille grid et renvoie les d premières
# fonctions évaluées sur la grille de points grid
    A=[]; # création d'une liste à allouer

    x = x/b
    for i in range(len(x)):

        phi=[1] # phi_1=1

        for j in range(1,d+1):

            # définition de la base trigonométrique dans le cas l'esti. par projection

            if (x[i]>=0 and x[i]<=1):  # car 0<= x <=b ssi   0<=x/b<=1

                phi.append(np.sqrt(2)*np.cos(2*np.pi*j*x[i]))
                phi.append(np.sqrt(2)*np.sin(2*np.pi*j*x[i]))

            else:
                phi.append(0)
                phi.append(0)

        A.append(phi)

        res=(1/np.sqrt(b))*np.array(A).T
    return(res)
```

In [348]:
```
x = np.array(np.linspace(0, 1, 100))
y = func_trig_tilde(2,1,x)

fig, ax = plt.subplots()

ax.plot(x, y.T)
ax.legend([r"$\varphi_0$", r"$\varphi_1$", r"$\varphi_2$", r"$\varphi_3$", r"$\varphi_4$"])

fig.tight_layout()
fig.show()
```

Out[348]:
```
[<matplotlib.lines.Line2D at 0x1fc9ca84548>,
 <matplotlib.lines.Line2D at 0x1fc9ca9eb48>,
 <matplotlib.lines.Line2D at 0x1fc9cabba08>,
 <matplotlib.lines.Line2D at 0x1fc9cabb9c8>,
 <matplotlib.lines.Line2D at 0x1fc9cabbb88>]
```

Out[348]:
```
<matplotlib.legend.Legend at 0x1fc993f6688>
```



In [ ]:

Question_4)

Coder une fonction f_hat qui prend en argument un entier D, un réel b > 0, une grille grid c [0; b] et un échantillon X et qui renvoie l'estimateur par projection de f construit à partir de la base orthonormée

In [349]:
```
def hat_f(X, d,b,grid):
# Coder une fonction f_hat qui prend en argument un entier D,

# X = data, grid = grille où est évaluée l'estimateur

    coeff = np.mean(func_trig_tilde(d,b, X),axis = 1)

    f_hat =np.sum(coeff.reshape(-1,1)*func_trig_tilde(d,b, grid), axis = 0)

    return f_hat, coeff
```

In [ ]:

Question_5): tracer sur un même graphique la fonction f et sa valeur estimée $\hat{f}_D$, pour différents choix de D.

Pour n = 1000, b = 3/4 et $X1 \sim B(3;2)$ (distribution Beta de paramètres (3,2)) tracer sur un même graphique la fonction f et sa valeur estimée $\hat{f}_D$, pour différents choix de D.

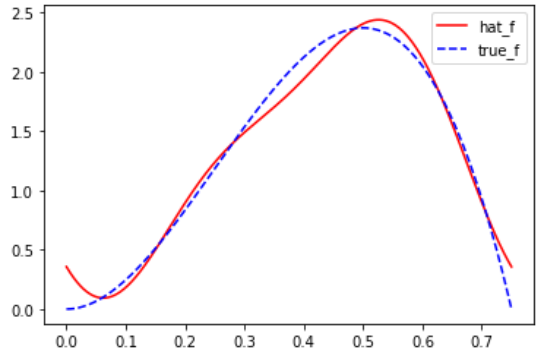
In [350]:
```
n = 1000
d = 2
b=3/4
grid = np.linspace(0,b,1000)
X = st.beta.rvs(3,2, size = n)*b  # on normalise les données

f_hat = hat_f(X, d,b, grid)
f_hat[1] # Coeffs estimés
true_f =st.beta.pdf(grid/b,3,2)/b  #X2
plt.plot(grid, f_hat[0], 'r-', label="hat_f")
plt.plot(grid, true_f, 'b--',label="true_f")
plt.legend()
```

Out[350]:
```
array([ 1.15470054, -0.4465771 , -0.47895213, -0.15122399, -0.06601969])
```

Out[350]:
```
[<matplotlib.lines.Line2D at 0x1fc9c8edb48>]
```

Out[350]:
```
[<matplotlib.lines.Line2D at 0x1fc9cad8b08>]
```

Out[350]:
```
<matplotlib.legend.Legend at 0x1fc9cabbd08>
```



Pour d = 2, 4, 6, 10, nous visualisons ci-dessous la fonction f et sa valeur estimée $\hat{f}_D$

In [351]:
```
plt.plot(grid, true_f, 'b--',label="true_f")

for d in [2,4,6,10]:
    f_hat = hat_f(X, d,b, grid)
    plt.plot(grid, f_hat[0], label="d="+str(d))
    plt.legend()
```

Out[351]:
```
[<matplotlib.lines.Line2D at 0x1fc9aff3748>]
```

Out[351]:
```
[<matplotlib.lines.Line2D at 0x1fc9ca7bc08>]
```

Out[351]:
```
<matplotlib.legend.Legend at 0x1fc9affdb48>
```

Out[351]:
```
[<matplotlib.lines.Line2D at 0x1fc9affddc8>]
```

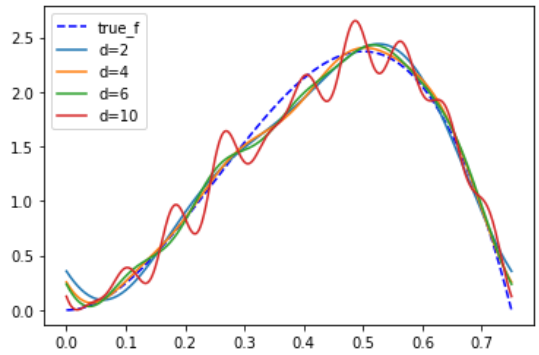
Out[351]:
```
<matplotlib.legend.Legend at 0x1fc9b039508>
```

Out[351]:
```
[<matplotlib.lines.Line2D at 0x1fc9affd608>]
```

Out[351]:
```
<matplotlib.legend.Legend at 0x1fc9b016e08>
```

Out[351]:
```
[<matplotlib.lines.Line2D at 0x1fc9affd748>]
```

Out[351]:
```
<matplotlib.legend.Legend at 0x1fc9b09a988>
```



Pour d = 50, 100, 150 nous visualisons ci-dessous la fonction f et sa valeur estimée $\hat{f}_D$

In [352]:
```
plt.plot(grid, true_f, 'b--',label="true_f")

for d in [50, 100, 150]:
    f_hat = hat_f(X, d,b, grid)
    plt.plot(grid, f_hat[0], label="d="+str(d))
    plt.legend()
```

Out[352]: [<matplotlib.lines.Line2D at 0x1fc9b0e1d08>]

Out[352]: [<matplotlib.lines.Line2D at 0x1fc9b0e1c88>]

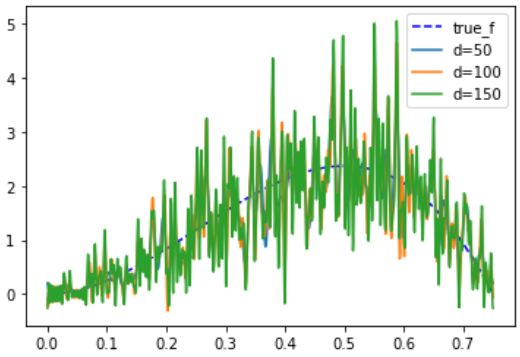
Out[352]: <matplotlib.legend.Legend at 0x1fc9b0ea488>

Out[352]: [<matplotlib.lines.Line2D at 0x1fc9b0ea5c8>]

Out[352]: <matplotlib.legend.Legend at 0x1fc9c6956c8>

Out[352]: [<matplotlib.lines.Line2D at 0x1fc9b0ea248>]

Out[352]: <matplotlib.legend.Legend at 0x1fc9b080f48>



conclusion: Pour des grandes valeurs de d, on arrive quasiement pas à voir les graphes, on revient sur le rôle de D qui est beaucoup capital pour la méthode d'estimation par projection

In [ ]:

Question_6):

Commenter le rôle de D Sa valeur semble-t-elle dépendre de n ? Expliquer.

Après le choix du critère de la base ou de la famille orthonormée, il vient le choix D qui est très important car s'il est mal fait, l'estimateur peut être catastrophique.

D doit être choisi suffisamment grand pour que le terme de biais soit petit : l'approximation déterministe de f par f_D est d'autant plus performante que D est grand.

Oui La valeur de D dépend de celle de n car le terme de variance qui(est liée à n) croit avec D(qui ne doit pas être choisi trop grand)

In [353]:
```
#Dépendance de n et D

plt.plot(grid, true_f, 'b--',label="true_f")
for n in [50,100,1000]:
    X = st.beta.rvs(3,2, size = n)*b

    f_hat = hat_f(X, d,b, grid)

    plt.plot(grid, f_hat[0], label="n="+str(n))
    plt.legend()
```

Out[353]: [<matplotlib.lines.Line2D at 0x1fc9dffb5c8>]

Out[353]: [<matplotlib.lines.Line2D at 0x1fc9c6c1348>]

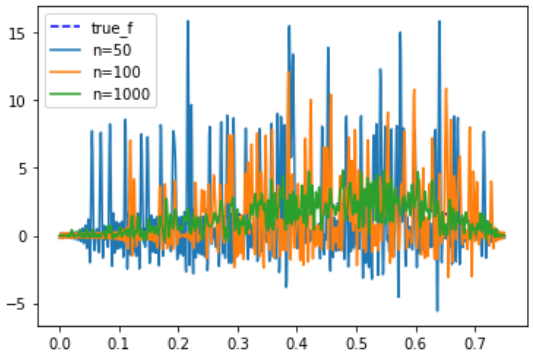
Out[353]: <matplotlib.legend.Legend at 0x1fc9dfff588>

Out[353]: [<matplotlib.lines.Line2D at 0x1fc9dfffd48>]

Out[353]: <matplotlib.legend.Legend at 0x1fc9dffbac8>

Out[353]: [<matplotlib.lines.Line2D at 0x1fc9c6995c8>]

Out[353]: <matplotlib.legend.Legend at 0x1fc9dfff808>



In [ ]:

Partie III: Etude d'un estimateur de r

En supposant que $\forall x \in [0,b], S(x) > a$ pour un $a > 0$ supposé connu, estimons r avec $\hat{r}_D(x) = \frac{\hat{f}_D(x)}{\hat{S}_n(x)} 1_{\{\hat{S}_n(x) > \frac{a}{2}\}}$ où $\hat{f}_D$ est l'estimateur de f

On considère $n = 1000, X_1 = Z_1 + U_1$ où $Z_1 \sim B(1,3)$ indépendante de $U_1 \sim U([0,2])$ et $b = 1$

In [354]:
```
n=1000; a=1/2; d=4; b=2
X=st.beta.rvs(1,3, size=n)+st.uniform.rvs(0,2, size=n)
grid1=np.linspace(0,1,n)
```

Question_7)

Justifions pourquoi on peut choisir $a = \frac{1}{2}$:

D'abord pour corriger l'effet que le dénominateur tend vers zéro ensuite, on peut choisir a = 1/2 pour permettre de voir la dissymétrie de la distribution empirique de l'échantillon. a = 1/2 est lié au calcule de la médiane.

La médiane est une valeur centrale de l'échantillon : il y a autant de valeurs qui lui sont inférieures que supérieures.

Si la distribution empirique de l'échantillon est peu dissymétrique, comme par exemple pour un échantillon simulé à partir d'une loi uniforme ou normale, la moyenne et la médiane sont proches.

Si l'échantillon est dissymétrique, avec une distribution très étalée vers la droite, la médiane pourra être nettement plus petite que la moyenne.

Contrairement à la moyenne, la médiane est insensible aux valeurs aberrantes. Elle possède une propriété d'optimalité par rapport à l'écart absolu moyen.

In [ ]:

Question_8)

Pour différentes valeurs de D, traçons différentes estimateur $\hat{r}_D$ construit à partir d'un n-échantillon X de variables aléatoires $i.i.d$ de même loi que X_1

In [355]:
```
def hat_r(X,a,d,b,grid):
    r=(hat_f(X,d,b,grid)[0]/hat_S(X,grid))*np.array([int(i) for i in hat_S(X,grid)>a/2])
    return r
```

In [356]:
```
for d in [1,2,3,4]:
    r_hat = hat_r(X,a,d,b,grid1)
    plt.plot(grid1,r_hat, label="d="+ str(d))
plt.legend()
```

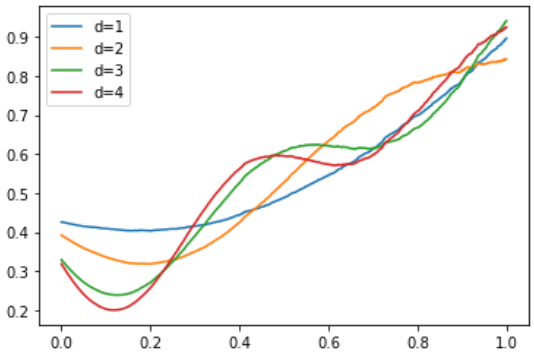
Out[356]: [<matplotlib.lines.Line2D at 0x1fc9e0c0d08>]

Out[356]: [<matplotlib.lines.Line2D at 0x1fc9b0ec848>]

Out[356]: [<matplotlib.lines.Line2D at 0x1fc9e0c2fc8>]

Out[356]: [<matplotlib.lines.Line2D at 0x1fc9e0b7c08>]

Out[356]: <matplotlib.legend.Legend at 0x1fc9e0cd788>



Bonus:

Une procédure pour sélectionner D de facon optimale et adaptative est: $\hat{D}_n = argmin_D\{-\sum_{j=1}^{D}\hat{a}_j^2 + \frac{\phi_0 D}{n}\}$

prenons $\phi_0 = 1$

In [357]:
```
def f(X,d,b,n):
    a_chapo=np.mean(func_trig_tilde(d,b, X),axis = 1)
    return -np.sum(a_chapo**2) + (2*d+1)/n
```

In [358]:
```
D_hat=[]
for d in range(20):
    D_hat.append(f(X,d,b,n))
np.argmin(D_hat)+1 # on ajoute 1 car les indices commencent à zéro en python
```

Out[358]: 4