HW4_DECHARRIN_adaptive

15/12/2023 de Charrin Théotime - MVA

TP4 - Adaptive Metropolis-Hastings / Gibbs algorithm¶

In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import make_axes_locatable
from matplotlib.colors import LogNorm
from matplotlib.ticker import MultipleLocator
import pandas as pd
import sklearn
import numba
from numba import njit
import numba as nb
import scipy.stats as stats
from numba.core import types
from numba import int32, int64, float64
from numba.typed import Dict
from numba.typed import List
from numba.types import bool_, int_, float32
from plotly import graph_objs as go
import time
import sys
import matplotlib.gridspec as gridspec
from alive_progress import alive_bar
from numbers import Number
import copy
```

In [2]:

```
plt.rcParams['figure.figsize'] = [12, 8]
plt.rcParams['figure.dpi'] = 100
```

```
def progressbar(it, prefix="", size=60, out=sys.stdout):
    count = len(it)
    start = time.time()
    def show(j):
        x = int(size*j/count)
        remaining = ((time.time() - start) / j) * (count - j)

        mins, sec = divmod(remaining, 60)
        time_str = f"{int(mins):02}:{sec:05.2f}"

        print(f"{prefix}[{u'#'*x}{('.'*(size-x))}] {j}/{count} Est wait {time_str}", end='\r

    for i, item in enumerate(it):
        yield item
        show(i+1)
    print("\n", flush=True, file=out)
```

Exercise 1: Adaptive Metropolis-Hastings within Gibbs sampler¶

1A. Metropolis-Hastings within Gibbs¶

Question 1¶

In [3]:

```
# Log de la distribution cible
@njit
def log_pi(x,a=10):
    return - np.square(x[0]/a) - np.square(x[1]) - .25*np.square(np.square(x[0]/a) - np.squa

@np.vectorize
def target_pi(x, y):
  return np.exp(log_pi(numba.typed.List([x, y])))

#Noyau de transition
@njit
def transition_P(x,a,sigma, pi=log_pi):
    B=np.random.binomial(1,.5,1)
    accepted=False
    #On utilise P1 si B=0
    if B==0:
        xstar = np.random.normal(x[0], sigma[0],size=1)[0]
        log_alpha = pi([xstar ,x[1]] , a) - pi(x,a)
        prob_choose=0 if 0<= log_alpha else log_alpha
        u=np.random.rand()
```

```
        if prob_choose>=np.log(u):
            x[0]=xstar
            accepted=True
    #Sinon on update avec P2
    elif B==1:
        ystar = np.random.normal(x[1], sigma[1],size=1)[0]
        log_alpha = pi([x[0] ,ystar] , a) - pi(x,a)
        prob_choose=0 if 0<= log_alpha else log_alpha
        u=np.random.rand()
        if prob_choose>=np.log(u):
            x[1]=ystar
            accepted=True
    return x, [accepted], B

# Échantillonneur MCMC-HM issu du TP3
#Répondra aussi à la question 1B (simple changement d'update par rapport au TP3)

#On introduit dès maintenant le delta_j
@njit(float64(int64))
def delta(j):
    change = 1e-2 if j**(-0.5) >= 1e-2 else j**(-0.5)
    return change

#On introduit l'update des sigma
@njit
def update_sigma(taux_accept, l, j):
    delta_j=delta(j)
    for i in range(taux_accept.shape[0]):
        l[i] += delta_j*np.sign(taux_accept[i]-0.24)
    return l

def SRWHM_Gibbs_sampler(x, a,sigma,noyau= transition_P, pi=log_pi,max_iter=20000,
                  fenetre=5000,burnin=0,verbose=True,verbose_plus=False, adaptive=False):
    sampled=np.zeros((max_iter,x.shape[0]))
    sampled[0,:]=x
    start=np.zeros_like(x)
    l_list=np.zeros_like(x)
    if adaptive:
        sigma=np.exp(l_list)
    # On ajoute une première ligne de 0 pour que l_list[-1] soit toujours disponible
    l_list=np.vstack((start, l_list))
    sigma_list=np.vstack((start,sigma))

    #On initie les compteurs pour les taux d'acceptation
    accept=np.zeros_like(x)
    taux_accept=np.zeros_like(x)
```

```
nombre_noyau= np.zeros_like(x)
j=0

for k in range(max_iter-1):
    x, accepted, component = noyau(numba.float64(x),a,numba.float64(sigma_list[-1]),pi=p
    sampled[k+1,:] = x
    #On regarde nos taux d'acceptation tous les batchs de {fenetre} pas.
    if k%fenetre==0 and k > 0:
        j+=1
        #On ajoute les taux d'acceptation
        taux_accept=np.vstack((taux_accept, (accept/nombre_noyau)))
        #On réinitialise les compteurs
        accept=np.zeros_like(x)
        nombre_noyau= np.zeros_like(x)
        #Si on est en adaptive Gibbs
        if adaptive:
            #On update l_i en fonction du taux d'acceptation qu'on vient de calculer et
            l_update = update_sigma(numba.float64(taux_accept[-1]), numba.float64(l_list
            l_list=np.vstack((l_list,l_update))
            #On update sigma avec le nouveau l_update
            sigma_list=np.vstack((sigma_list,np.exp(l_update)))

        #Retour en direct sur notre taux d'acceptation
        if verbose_plus:
            print(f'Moyenne actuelle pour x ({k=}):  {taux_accept[-1,0]:.2%}')
            print(f'Moyenne actuelle pour y ({k=}):  {taux_accept[-1,1]:.2%}')


    #A chaque étape, on update quelle composante a été choisie et si elle a été acceptée
    #On update à la fin de la boucle pour conserver la somme = fenetre
    for i in component:
        nombre_noyau[i]+=1

    for n, bool in enumerate(accepted):
        if bool==True:
            accept[component[n]]+=1


#A la fin de la boucle for
if verbose:
    print(f'Taux d\'acceptation moyen pour x :  {np.mean(taux_accept,axis=0)[0]:.2%}')
    print(f'Taux d\'acceptation moyen pour y :  {np.mean(taux_accept,axis=0)[1]:.2%}')

#On enlève la première ligne qui était la pour l'initialisation
sigma_list=np.delete(sigma_list,0,0)
l_list=np.delete(l_list,0,0)
```

```
    return sampled, sigma_list[-1], taux_accept, sigma_list, l_list
```

Question 2¶

In [4]:

```
max_iter=900000
fenetre=50
burnin=10000*fenetre
center_start=np.array([0,0])

sampled, sigma, accept, *kwargs = SRWHM_Gibbs_sampler(center_start, 10, np.array([3,3]),
                                                      verbose=False, burnin=burni
                                                      fenetre=fenetre)
```

In [5]:

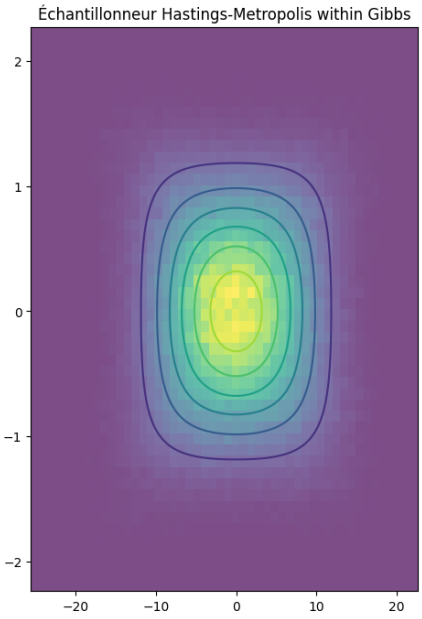
```
# Taux d'acceptation
print(f'Taux d\'acceptation moyen pour x :  {np.mean(accept, axis=0)[0]:.2%}')
print(f'Taux d\'acceptation moyen pour y :  {np.mean(accept,axis=0)[1]:.2%}')
```

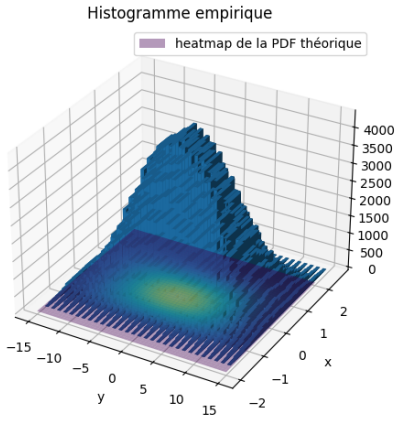
```
Taux d'acceptation moyen pour x :  86.06%
Taux d'acceptation moyen pour y :  25.98%
```

In [6]:

```
fig=plt.figure()
# Sanity check
# On compare ce qui a été échantillonné et la distribution théorique
ax=fig.add_subplot(121)
x = np.linspace(-15,15 ,100)
y = np.linspace(-2,2,100)
X, Y = np.meshgrid(x, y)
Z = target_pi(X,Y)
ax.contour(X, Y, Z)
#ax.scatter(sampled[:,0],sampled[:,1] , label = 'Échantillonné')
ax.hist2d(sampled[:,0],sampled[:,1],bins=50,alpha=0.7)
#ax.legend()
ax.set_title("Échantillonneur Hastings-Metropolis within Gibbs")

# On compare les histogrammes
ax2 = fig.add_subplot(122, projection='3d')
hist, xedges, yedges = np.histogram2d(sampled[:,0], sampled[:,1], bins=30, range=[[-15, 15],
xpos, ypos = np.meshgrid(xedges[:-1] + 0.25, yedges[:-1] + 0.25, indexing="ij")
xpos = xpos.ravel()
ypos = ypos.ravel()
zpos = 0
dx = dy = 0.5 * np.ones_like(zpos)
dz = hist.ravel()
```

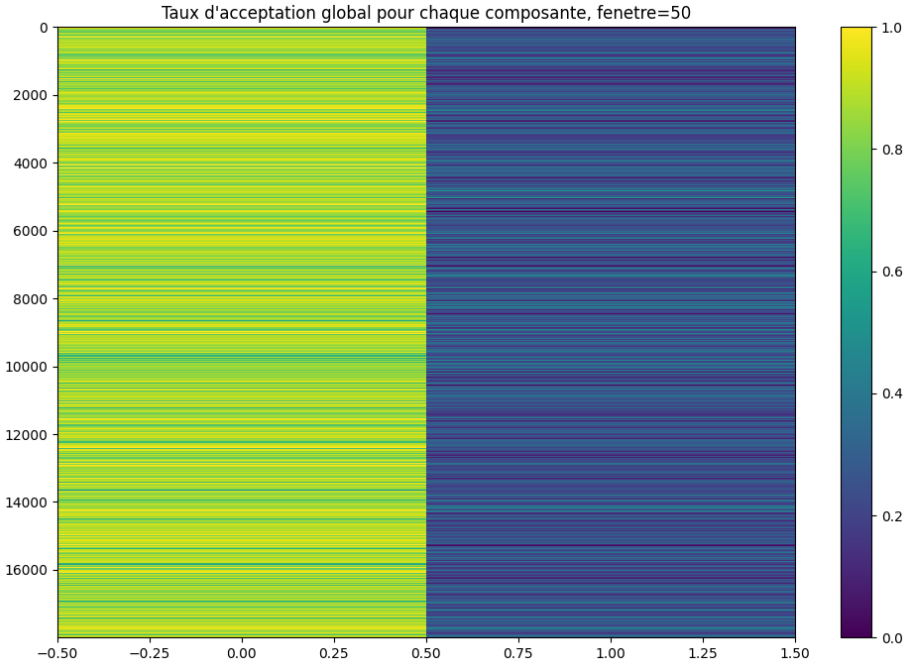
```
ax2.bar3d(xpos, ypos, zpos, dx, dy, dz, zsort='average')
ax2.set_xlabel('y')
ax2.set_ylabel('x')
ax2.set_title('Histogramme empirique')

#PDF
ax2.plot_surface(X, Y, Z, cmap='viridis', alpha=0.4,label="heatmap de la PDF théorique")
plt.legend()
plt.show()
```





In [7]:

```
plt.imshow(accept,aspect='auto', interpolation='nearest')
plt.colorbar()
plt.title(f'Taux d\'acceptation global pour chaque composante, {fenetre=}')
plt.show()
```



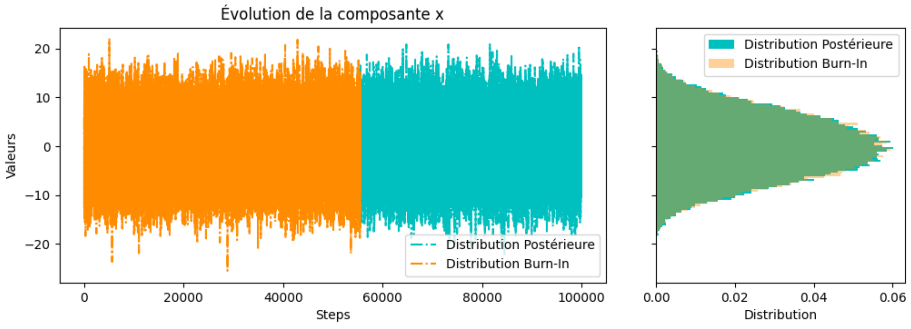
In [8]:

```
# Issu du TP3
# Pour plotter les trajectoires de chaque composante.
def plot_chain(samples, evaluated, burnin, nsig=1, fmt='-', y_range=None,
        width=3000, height=800, margins={'l':20, 'r':20, 't':50, 'b':20}):
    ##Adapté de exowanderer
    if str(evaluated)=="xi" or str(evaluated)=="tau":
        raise ValueError('Ne peut pas plotter une chaîne multidimensionnelle')


    estimate = np.mean(samples[burnin:])
    stddev = np.std(samples[burnin:])
    title = f'Évolution de la composante {str(evaluated)}'

    num_samples = len(samples)
    idx_burnin = burnin
    #On limite le nombre de points plottés à 100k
    if len(samples) > 100*1e3:
        denominateur=int(len(samples)/(100*1e3))
        samples=np.concatenate([[samples[denominateur*i]] for i in range(int(len(samples)/de
        idx_burnin=int(burnin/denominateur)
        num_samples=len(samples)
```

```
    sample_steps = np.arange(num_samples)

    samples_posterior = samples[idx_burnin:]
    samples_burnin = samples[:idx_burnin]

    estimate = np.mean(samples_posterior)
    stddev = np.std(samples_posterior)


    if y_range is None:
        std_post = np.std(samples_posterior)
        y_range = min(samples) - nsig * std_post, max(samples) + nsig * std_post
    fig = plt.figure()
    gs = gridspec.GridSpec(2, 3)
    ax_main = plt.subplot(gs[1:, :2])
    ax_xDist = plt.subplot(gs[1:, 2],sharey=ax_main)

    ax_main.plot(sample_steps[idx_burnin:], samples_posterior, color='c', label='Distributio
    ax_main.plot(sample_steps[:idx_burnin], samples_burnin, color='darkorange', label='Distr
    ax_main.set(xlabel="Steps", ylabel="Valeurs")

    ax_xDist.hist(samples_posterior,bins=100,align='mid',density=True, orientation='horizont
    ax_xDist.hist(samples_burnin,bins=100,align='mid',density=True,orientation='horizontal',
    ax_xDist.set(xlabel='Distribution')
    ax_main.set_title(title)
    ax_main.legend()
    ax_xDist.legend()
    plt.setp(ax_xDist.get_yticklabels(), visible=False)

    plt.show()
```

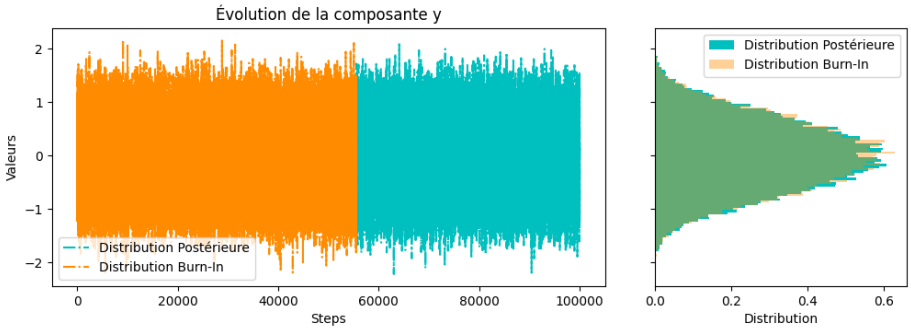
In [9]:

```
plot_chain(sampled[:,0], "x", burnin=burnin)
```



In [10]:

```
plot_chain(sampled[:,1], "y", burnin=burnin)
```

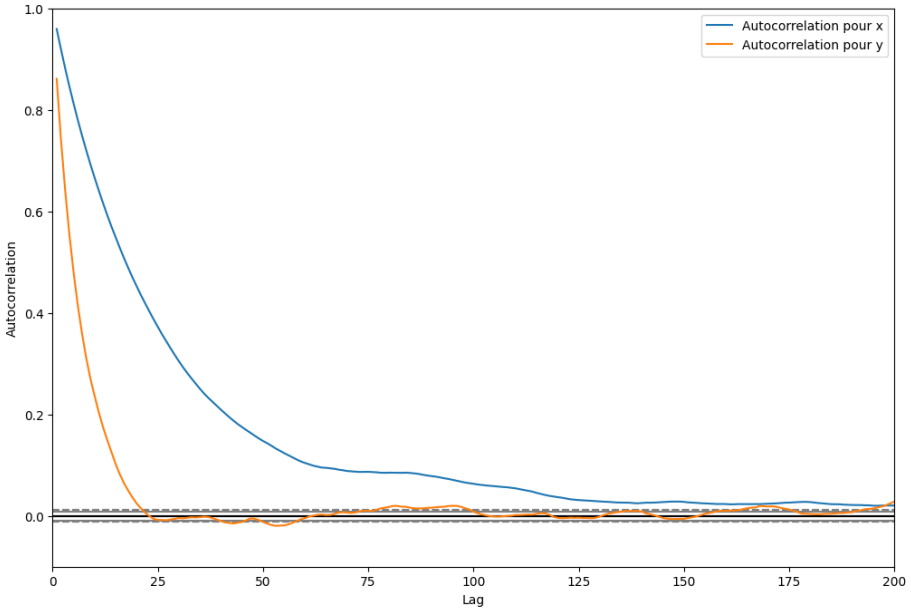


Les chaînes de Markov semblent être stables.

Sur des essais avec des starting points très éloignés (-60, 25), elles convergent en moins de 600 étapes.

In [11]:

```
# On plotte l'autocorrélation pour les 200 premiers timepoints (pas très informatif sinon)
# On triche en ne gardant que les 50000 premiers points, sinon pandas est très lent à calcul
x=pd.Series(sampled[:50000,0])
y=pd.Series(sampled[:50000,1])
autox = pd.plotting.autocorrelation_plot(x, label='Autocorrelation pour x')
autoy = pd.plotting.autocorrelation_plot(y, label='Autocorrelation pour y')
autox.plot()
autoy.plot()
plt.xlim((0,200))
plt.ylim((-0.1,1))
plt.show()
```



En partant d'un couple (x_0,y_0) très proche des valeurs moyennes cible, on voit que l'autocorrélation est bonne pour y et correcte pour x.

En partant d'un couple (x_0,y_0) très éloigné des valeurs moyennes cible (non montré), on a pu voir que l'autocorrélation était très médiocre (plus de 100 itérations pour la composante x avant de trouver des échantillons indépendants des valeurs initiales).

En modifiant notre algorithme, il est possible d'améliorer ces performances

Question 3¶

On voit ici que les échantillons sont souvent en dehors de la distribution concernant la composante x, et qu'on accepte trop souvent sur cette même composante. Ceci est dû au fait que la distribution cible n'a pas la même variance en x et en y (on le voit bien sur la distribution théorique).

Pour améliorer la performance de l'algorithme, on pourrait :

- Modifier le σ_1 pour x et l'augmenter, ce qui ferait chuter le taux d'acceptation pour x
- Modifier la probabilité de P_1 et P_2 pour donner un poids plus important à l'échantillonnage de x

In [12]:

```
# On définit un template pour évaluer l'efficacité de l'algorithme
def algo_assesment(sampled_test, sigma_test, taux_accept_test, burnin,
                   sampled=sampled, sigma=sigma, taux_accept=accept, pi="target", target=tar
```

```
# Taux d'acceptation
print(f'Taux d\'acceptation moyen pour x :  {np.mean(taux_accept_test,axis=0)[0]:.2%} (t
print(f'Dernier taux d\'acceptation pour x :  {taux_accept_test[-1][0]:.2%} (test) vs {t
print(f'Taux d\'acceptation moyen pour y :  {np.mean(taux_accept_test,axis=0)[1]:.2%} vs
print(f'Dernier taux d\'acceptation pour y :  {taux_accept_test[-1][1]:.2%} (test) vs {t
fig=plt.figure()

# Sanity check
# On compare ce qui a été échantillonné et la distribution théorique
ax=fig.add_subplot(121)
x = np.linspace(-15,15 ,100)
y = np.linspace(-2,2,100)
X, Y = np.meshgrid(x, y)
Z=target(X,Y)
CS = ax.contour(X, Y, Z, levels=8)
ax.clabel(CS, inline=True, fontsize=10)

ax.scatter(sampled_test[:,0],sampled_test[:,1] , label = 'Échantillonné (test)')
ax.legend(loc='lower right')
ax.set_title("Échantillonneur Hastings-Metropolis within Gibbs")

# On compare les histogrammes
ax2 = fig.add_subplot(122, projection='3d')
if pi=="target":
    hist, xedges, yedges = np.histogram2d(sampled_test[:,0], sampled_test[:,1], bins=30,

else:
    hist, xedges, yedges = np.histogram2d(sampled_test[:,0], sampled_test[:,1], bins=300
    ax2.set_xlim([-25, 25])
    ax2.set_ylim([-15, 15])

xpos, ypos = np.meshgrid(xedges[:-1] + 0.25, yedges[:-1] + 0.25, indexing="ij")
xpos = xpos.ravel()
ypos = ypos.ravel()
zpos = 0
dx = dy = 0.5 * np.ones_like(zpos)
dz = hist.ravel()
ax2.bar3d(xpos, ypos, zpos, dx, dy, dz, zsort='average')
ax2.set_xlabel('y')
ax2.set_ylabel('x')
ax2.set_title('Histogramme empirique')

#PDF
ax2.plot_surface(X, Y, Z, cmap='viridis', alpha=0.5,label="heatmap de la PDF théorique")
plt.legend()
plt.show()
```

```
CS = plt.contour(X, Y, Z)
plt.clabel(CS, inline=True)
plt.scatter(sampled_test[burnin:,0],sampled_test[burnin:,1] , label = 'Échantillonné (te
plt.scatter(sampled[burnin:,0],sampled[burnin:,1] , label = 'Échantillonné (non adaptati
if pi=="gaussian":
    plt.scatter(gaussian[:,0],gaussian[:,1] , label = 'Ground Truth', alpha=0.3,marker='
plt.legend(loc='lower right')
plt.title(f"Échantillonneur Hastings-Metropolis within Gibbs après {burnin} itérations")
plt.show()

fig,(ax1, ax2)=plt.subplots(2)
fig.suptitle(f'Taux d\'acceptation global pour chaque composante, {fenetre=}')
ax1.set_title('Test')
im1=ax1.imshow(taux_accept_test,aspect='auto', interpolation='kaiser')
divider1=make_axes_locatable(ax1)
cax1 = divider1.append_axes("right", size="20%", pad=0.05)
cbar1 = plt.colorbar(im1, cax=cax1, ticks=[0, 0.24, 0.5, 1],
                format="%.2f",
                extend='both')
ax1.xaxis.set_visible(False)
ax1.xaxis.set_visible(False)


ax2.set_title('Non adaptatif')
im2=ax2.imshow(taux_accept,aspect='auto', interpolation='kaiser')
divider2=make_axes_locatable(ax2)
cax2 = divider2.append_axes("right", size="20%", pad=0.05)
cbar2 = plt.colorbar(im2, cax=cax2, ticks=[0, 0.24, 0.5, 1],
                format="%.2f",
                extend='both')
ax2.xaxis.set_visible(False)
plt.tight_layout()
# Make space for title
plt.subplots_adjust(top=0.85)
plt.show()

plot_chain(sampled_test[:,0], "x", burnin=burnin)
plot_chain(sampled_test[:,1], "y", burnin=burnin)

# On plotte l'autocorrélation pour les 200 premiers timepoints (pas très informatif sino
x_test=pd.Series(sampled_test[:50000,0])
y_test=pd.Series(sampled_test[:50000,1])
autox_test = pd.plotting.autocorrelation_plot(x_test, label='Autocorrelation pour x_test
autoy_test = pd.plotting.autocorrelation_plot(y_test, label='Autocorrelation pour y_test
autox_test.plot()
```

```
autoy_test.plot()
x=pd.Series(sampled[:50000,0])
y=pd.Series(sampled[:50000,1])
autox = pd.plotting.autocorrelation_plot(x, label='Autocorrelation pour x (non adaptatif
autoy = pd.plotting.autocorrelation_plot(y, label='Autocorrelation pour y (non adaptatif
autox.plot()
autoy.plot()
plt.xlim((0,200))
plt.ylim((-0.25,1))
plt.legend(loc='lower right')
plt.show()
```

1.B – Adaptive Metropolis-Hastings within Gibbs sampler¶

Question 1¶

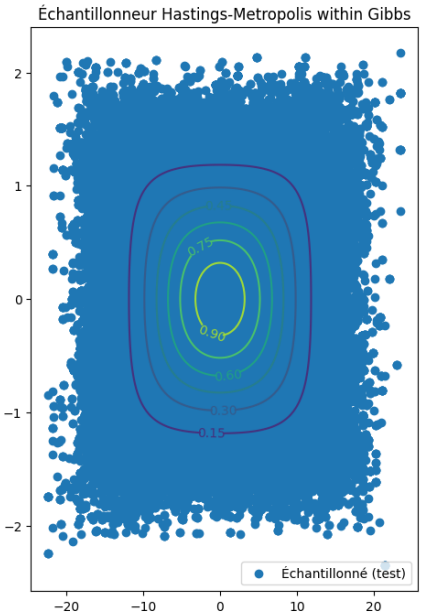
In [13]:

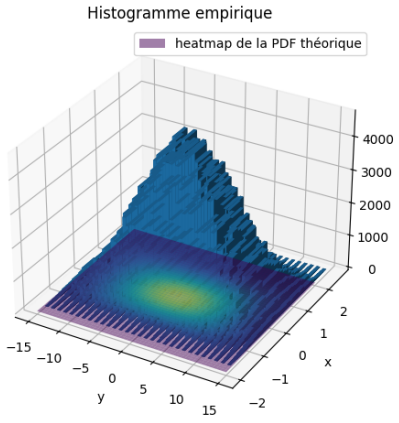
```
sampled_adapt, sigma_adapt, accept_adapt, sigma_list_adapt, l_list_adapt  = SRWHM_Gibbs_samp
```

In [14]:

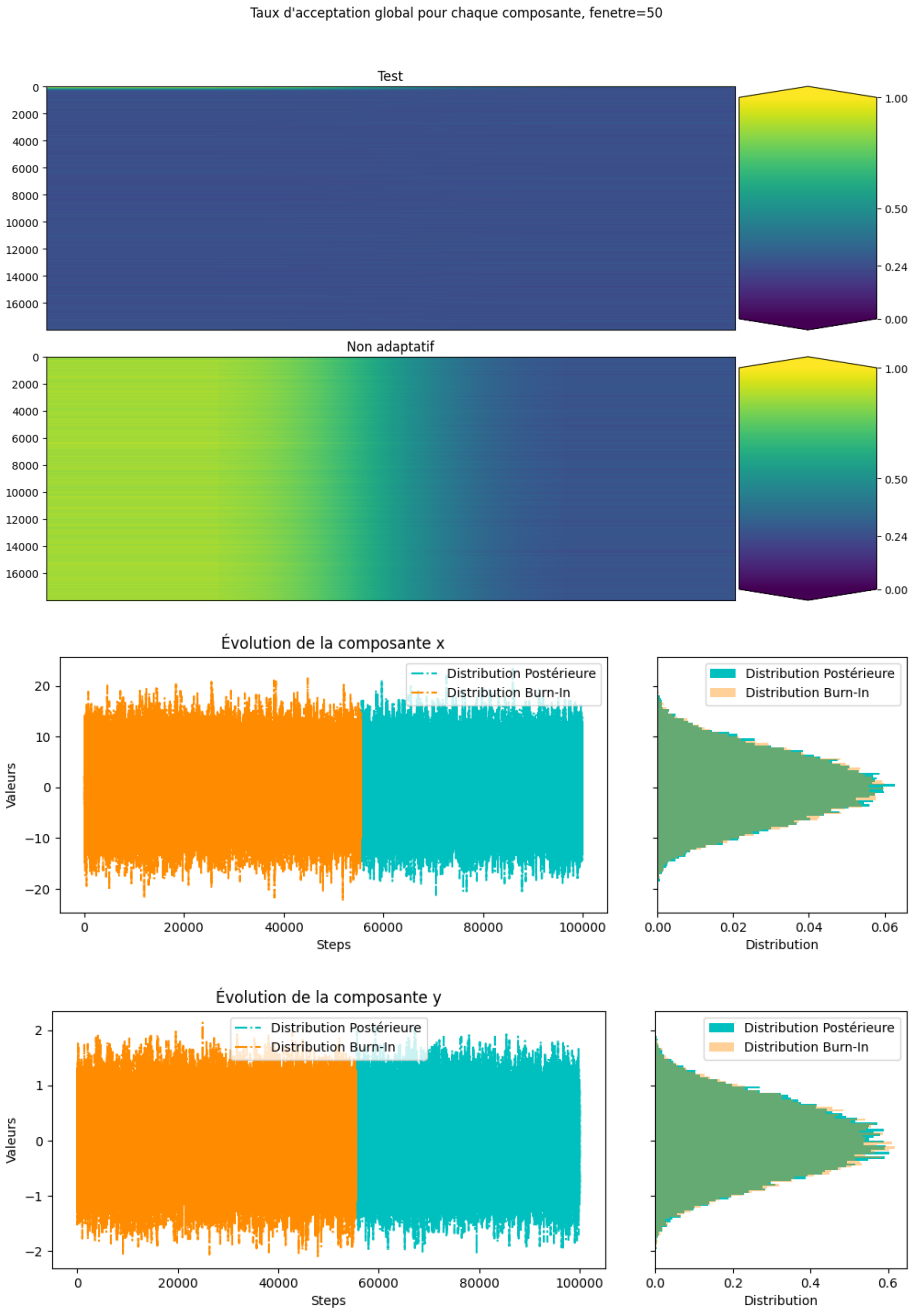
```
algo_assesment(sampled_adapt, sigma_adapt, accept_adapt, burnin=burnin)
```

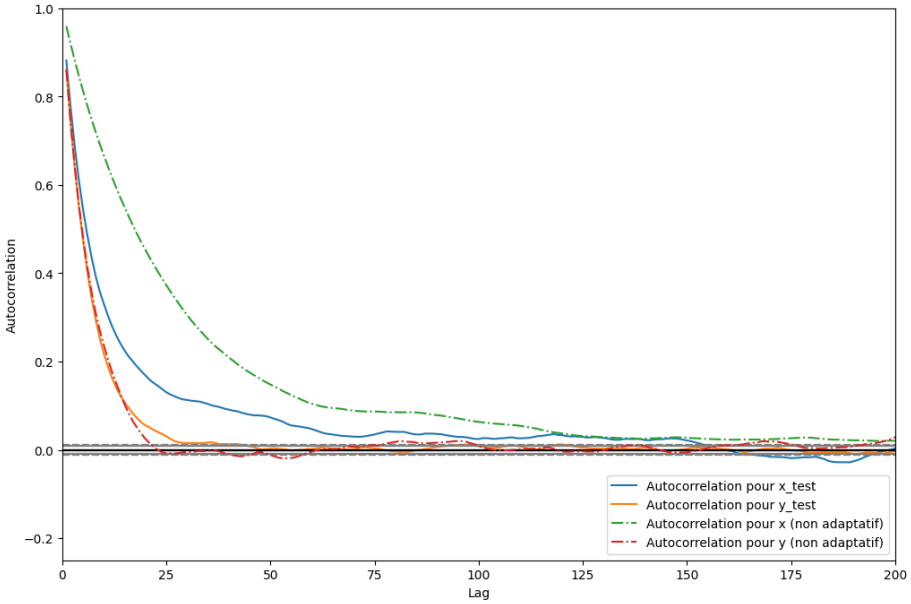
```
Taux d'acceptation moyen pour x :  25.20% (test) vs 86.06% (non adaptatif)
Dernier taux d'acceptation pour x :  30.43% (test) vs 60.87% (non adaptatif)
Taux d'acceptation moyen pour y :  24.54% vs 25.98% (non adaptatif)
Dernier taux d'acceptation pour y :  29.63% (test) vs 29.63% (non adaptatif)
```


Échantillonneur Hastings-Metropolis within Gibbs
0.75
0.90
0.60
0.30
0.15
Échantillonné (test)


Histogramme empirique
heatmap de la PDF théorique
x
y


Échantillonneur Hastings-Metropolis within Gibbs après 500000 itérations
0.45
0.90
0.60
0.30
0.15
Échantillonné (test)
Échantillonné (non adaptatif)


Taux d'acceptation global pour chaque composante, fenetre=50
Test
Non adaptatif
Évolution de la composante x
Évolution de la composante y
Distribution Postérieure
Distribution Burn-In
Valeurs
Steps
Distribution



Avec ou sans adaptation, ce qui est échantillonné "à la fin" de l'algorithme semble assez similaire (à max_iter constant). Les échantillons adaptatifs semblent mieux répartis selon l'axe x, de manière plus "uniforme".

Cependant, on voit très vite une convergence des taux d'acceptation vers 0.24 (optimal), et l'autocorrélation converge beaucoup plus vite vers 0, notamment (surtout) pour x. Ceci nous fait penser qu'on pourrait utiliser moins d'étapes avec l'algorithme adaptatif, et qu'il présentera des performances similaires.

Concernant les σ_i, l'algorithme adaptatif nous permet d'atteindre des valeurs proches de 30 pour x, tandis que les valeurs pour y restent proches de 3.

Question 2¶

Concernant la loi normale multivariée. On prend donc comme loi target : $$ \log \pi \propto -\dfrac{1}{2} \log \Sigma - \dfrac{1}{2} x^T \Sigma x $$

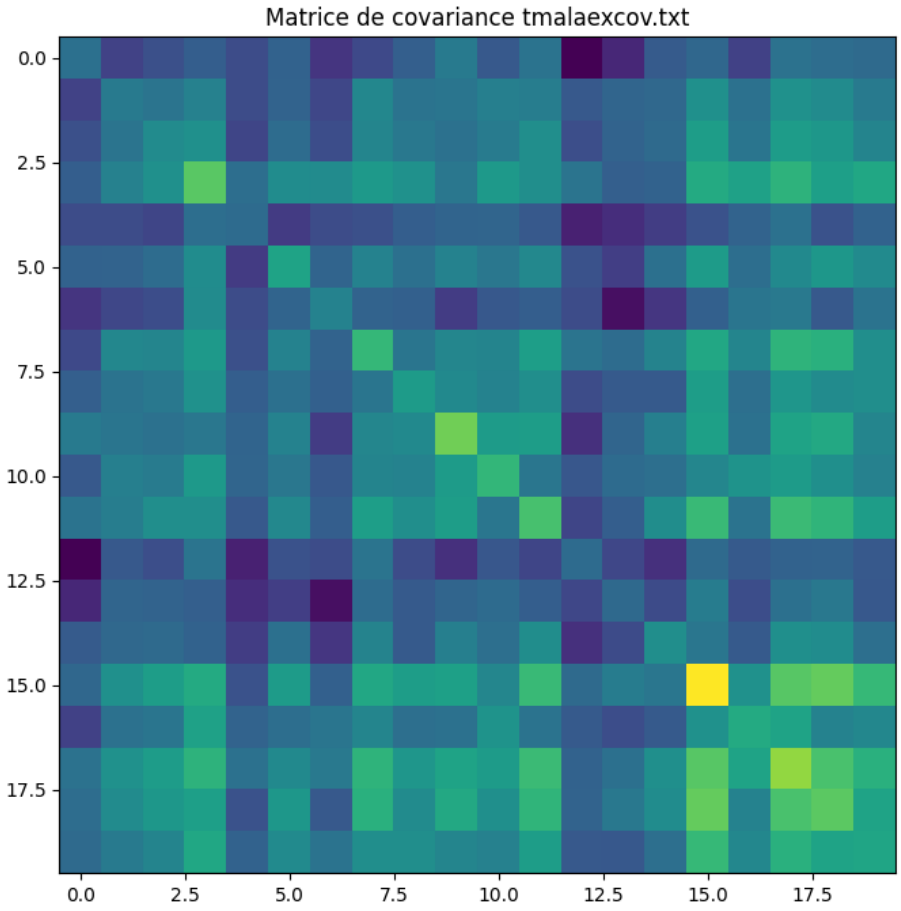
In [15]:

```
# Sigma tmalaexov.txt trouvé sur https://github.com/Aurel37/ComptStat_adaptative_langevin/bl
d=20
N=1000
Sigma_True=np.array(pd.read_csv('./tmalaexcov.txt',header=None, sep=' '))
gaussian=np.random.multivariate_normal(mean=np.zeros(d), cov=Sigma_True, size= N)
```

In [16]:

```
plt.imshow(Sigma_True)
plt.title("Matrice de covariance tmalaexcov.txt")
```

```
plt.show()
```



In [17]:

```
#On réutilise les premières fonctions mais :

#On introduit log_pi pour la normale multivariée
def log_likelimulti(x, mean=np.zeros(d), cov=Sigma_True):
    #Pour pouvoir plotter la heatmap
    if x.shape[0]!=mean.shape[0]:
        new=np.pad(x, (0,d-x.shape[0]), 'constant', constant_values=0)
        x=new
    #Volé honteusement à Gregory Gundersen
    vals, vecs = np.linalg.eigh(cov)
    logdet     = np.sum(np.log(vals))
    valsinv    = 1./vals
    U          = vecs * np.sqrt(valsinv)
```

```
    dim        = len(vals)
    dev        = x - mean
    maha       = np.square(np.dot(dev, U)).sum()
    return -0.5 * (maha + logdet)

#On introduit l'update de Gibbs
#@njit
def noyau_Gibbs(x, a, sigma, pi=log_likelimulti):
    xstar=copy.copy(x)
    acceptance_list=[]
    for toupdate in range(x.shape[0]):
        xstar[toupdate] = np.random.normal(x[toupdate], sigma[toupdate],size=1)[0]
        log_alpha=pi(x=xstar) - pi(x=x)
        prob_choose=0 if 0<= log_alpha else log_alpha
        u=np.random.rand()
        if prob_choose>=np.log(u):
            x=xstar
            acceptance_list.append(True)
        else:
            acceptance_list.append(False)
    component_list=[i for i in range(x.shape[0])]
    return x, acceptance_list, component_list

@np.vectorize
def target_multi(x, y):
    return np.exp(log_likelimulti(np.array([x, y])))
```

In [19]:

```
center_start_gauss=np.ones(d)
max_iter=500000
fenetre=50
burnin=5000*fenetre
start=time.time()
sampled_gauss, sigma_gauss, accept_gauss, sigma_list_gauss, l_list_gauss  = SRWHM_Gibbs_samp



                                                                    verbose=False, adaptive=Fal
                                                                    burnin=burnin, max_iter=max


sampled_gauss_adapt, sigma_gauss_adapt, accept_gauss_adapt, sigma_list_gauss_adapt, l_list_g



                                                                    verbose=False, adaptive=Tru
```

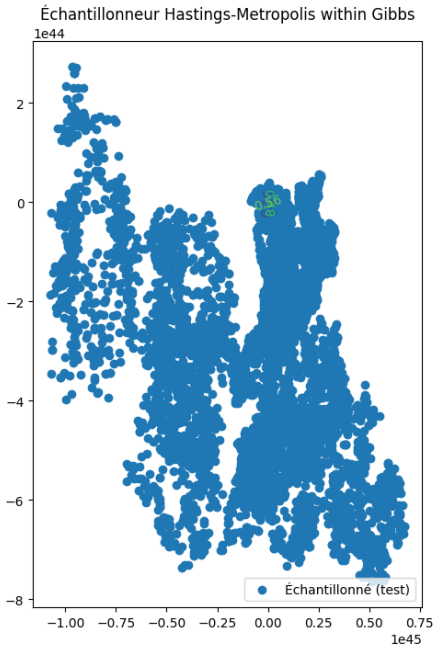
```
                                                                    burnin=burnin, max_iter=max
end=time.time()
print(end-start)
```

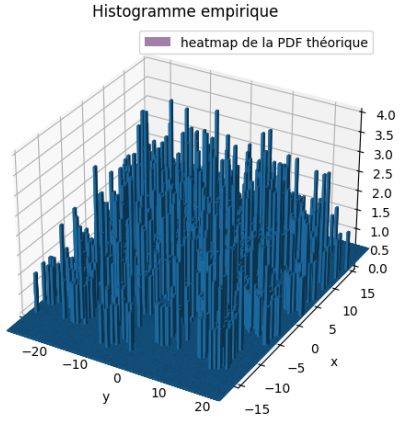
```
974.8021595478058
```

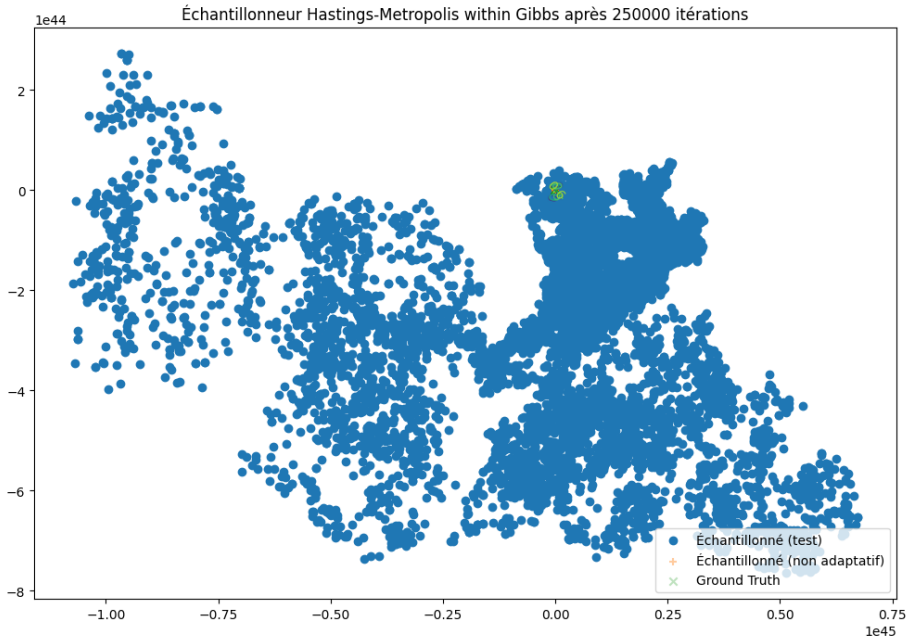
In [20]:

```
algo_assesment(sampled_gauss_adapt, sigma_gauss_adapt, accept_gauss_adapt, burnin,
               sampled_gauss, sigma_gauss, accept_gauss, pi="gaussian",target=target_multi)
```

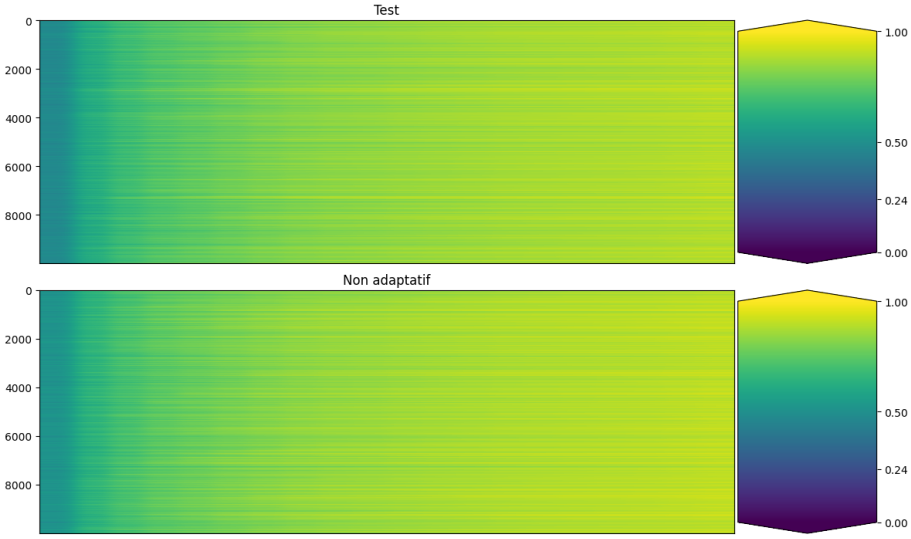
```
Taux d'acceptation moyen pour x :  47.61% (test) vs 51.49% (non adaptatif)
Dernier taux d'acceptation pour x :  50.00% (test) vs 62.00% (non adaptatif)
Taux d'acceptation moyen pour y :  61.83% vs 64.65% (non adaptatif)
Dernier taux d'acceptation pour y :  66.00% (test) vs 70.00% (non adaptatif)
```

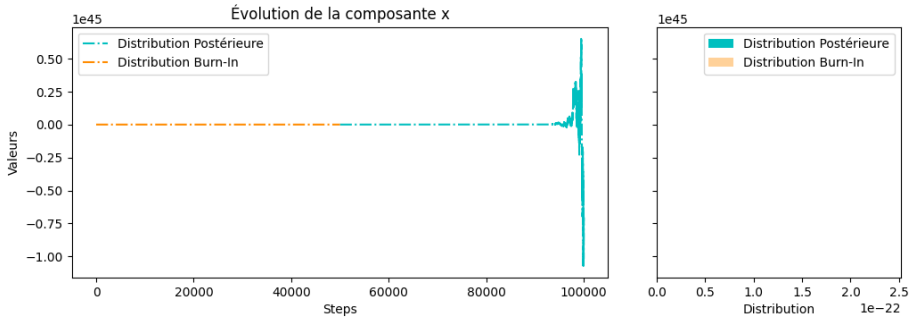


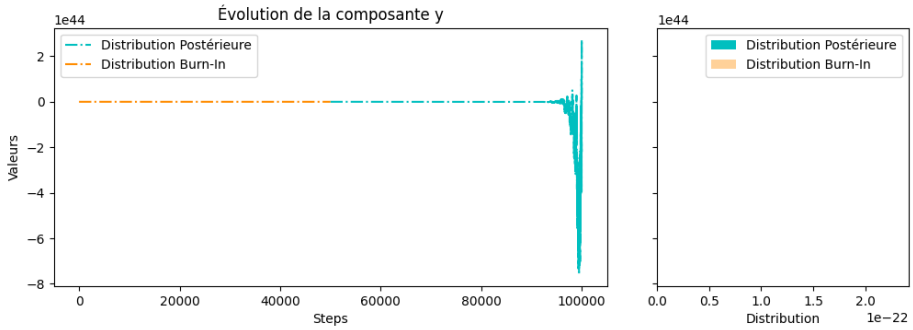


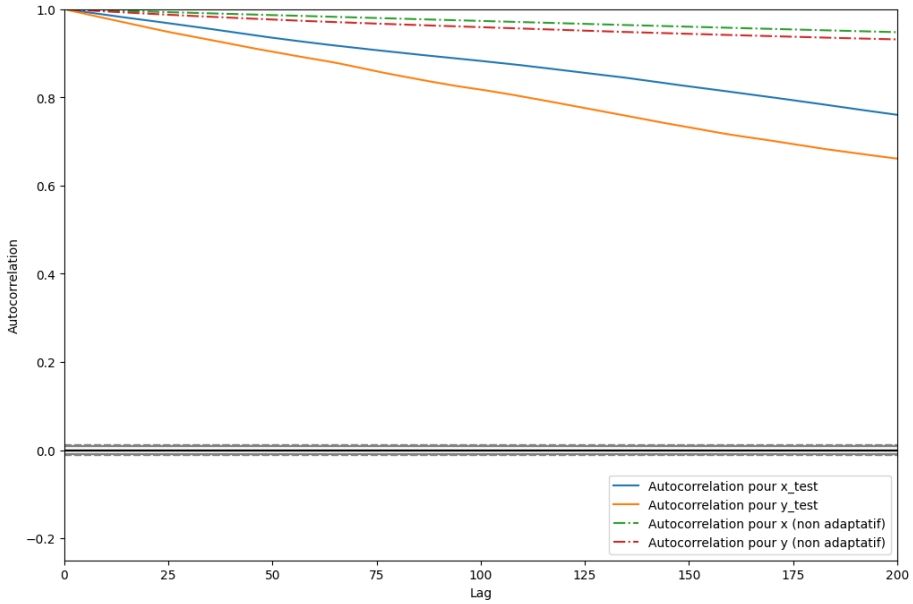


Taux d'acceptation global pour chaque composante, fenetre=50









Comme on peut le voir, l'algorithme adaptatif ne semble pas du tout mieux performer que l'algorithme non adaptatif.

Sur les 2 première composantes, il diverge complètement au bout d'un certain

temps. Ceci est dû au fait que la variance des lois de proposition grandit de façon exponentielle, et l'algorithme n'arrive jamais à accepter les propositions à un taux régulier de 24%.

À cause de la grande dimension et l'anistropie de la matrice de covariance, l'algorithme performe très mal. Il permet d'améliorer marginalement l'autocorrélation.

Concernant la densité "banane" :

In [30]:

```
B=0.1
d=20
sigma_banane=np.ones(d)
# Distribution banane
#@njit
def log_pi_banane(x,B=B,cov=sigma_banane):
    return -np.square(x[0])/200.-.5*np.square((x[1]+B*np.square(x[0])-100*B))-.5*(np.sum(np.

@np.vectorize
def target_banane(x, y):
  return np.exp(log_pi_banane(numba.typed.List([x, y])))
```

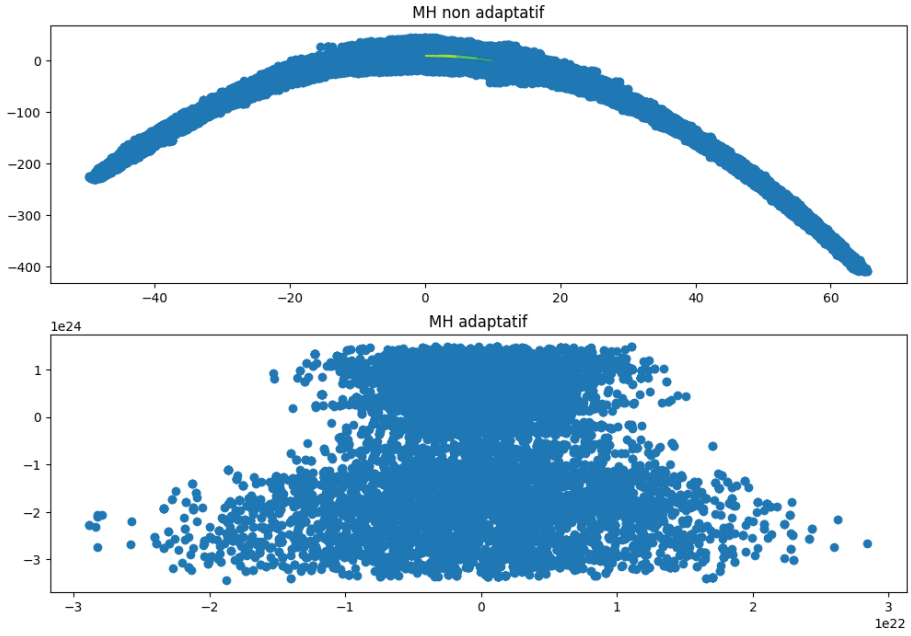
In [24]:

```
center_start_gauss=np.ones(d)
sampled_ban, sigma_ban, accept_ban, sigma_list_ban, l_list_ban  = SRWHM_Gibbs_sampler(center

                                                                                verbose=False, adaptive=Fal
                                                                                burnin=burnin, max_iter=max


sampled_ban_adapt, sigma_ban_adapt, accept_ban_adapt, sigma_ban_gauss_adapt, l_list_ban_adap

                                                                                verbose=False, adaptive=Tru
                                                                                burnin=burnin, max_iter=max
```

In [32]:

```
fig ,(ax1,ax2)= plt.subplots(2)
X, Y = np.meshgrid(np.linspace(0, 10, 100),np.linspace(0, 10, 100))
Z = target_banane(X, Y)
ax1.contour(X, Y, Z)
ax1.set_title('MH non adaptatif')
ax1.scatter(sampled_ban[:,0], sampled_ban[:,1])
```

```
ax2.contour(X, Y, Z)
ax2.set_title('MH adaptatif')
ax2.scatter(sampled_ban_adapt[:,0], sampled_ban_adapt[:,1])
plt.show()
```



Même constat que pour la loi multivariée en grande dimension. Je n'arrive pas à comprendre pourquoi l'échantillonnage est beaucoup moins performant en adaptatif.

Exercice 2 - Échantillonnage de distributions multimodales¶

2.A – A toy example¶

In [33]:

```
mu = np.array([[2.18,5.76],[8.67,9.59],[4.24,8.48],[8.41,1.68],
               [3.93,8.82],[3.25,3.47],[1.70,0.50],[4.59,5.60],
               [6.91,5.81],[6.87,5.40],[5.41,2.65],[2.70,7.88],
               [4.98,3.70],[1.14,2.39],[8.33,9.50],[4.93,1.50],
               [1.83,0.09],[2.26,0.31], [5.54,6.86],[1.69,8.11]])
d=2
Sigma_multimod=0.1
w_multi=1/d
```

In [34]:

```
# Pour un simple HMSRW, on définit un nouveau noyau qui est la loi normale multivariée

#On introduit d'abord log_pi pour le GMM
#On reconnaît une somme de gaussiennes pondérées par w_i
#@njit
def pi_gmm(x,mu=mu,sigma=Sigma_multimod,w=w_multi,Nb_gauss=20):
    return np.sum([w_multi*stats.multivariate_normal.pdf(x, mean=mu[i], cov=Sigma_multimod**

def log_pi_gmm(x):
    return np.log(pi_gmm(x))


@np.vectorize
def target_pi_gmm(x, y):
    return pi_gmm([x, y])

def noyau_SRW(x,a,se=Sigma_multimod, pi=log_pi_gmm):
    acceptance_list=[]
    xstar=np.random.normal(loc=x, scale=se)
    log_alpha=pi(x=xstar) - pi(x=x)
    prob_choose=0 if 0<= log_alpha else log_alpha
    u=np.random.rand()
    if prob_choose>=np.log(u):
        x=xstar
        acceptance_list.append(True)
    else:
        acceptance_list.append(False)
    component_list=[i for i in range(x.shape[0])]
    return x, acceptance_list, component_list
```

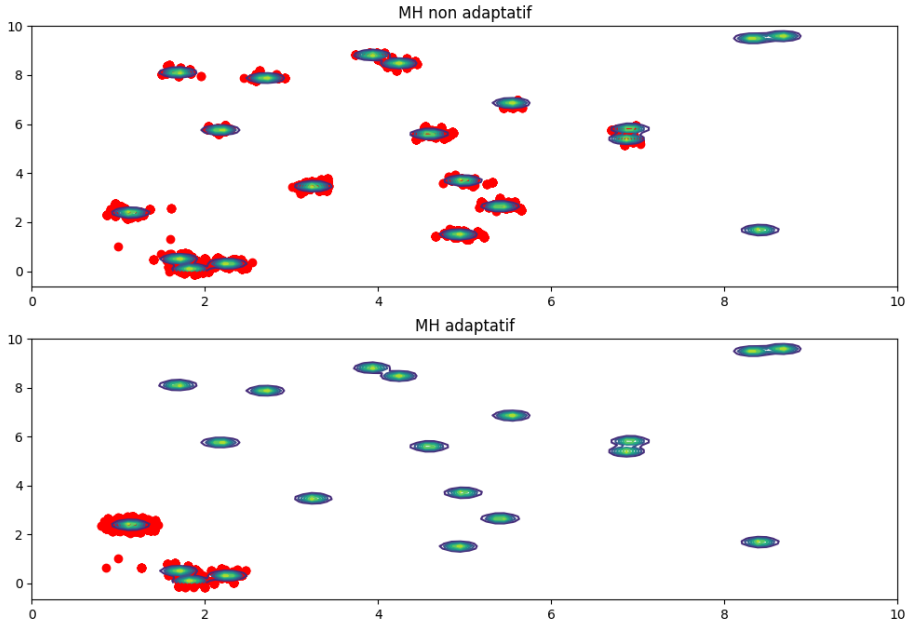
In [35]:

```
max_iter=10000
burnin=2000
fenetre=50
center_start_multi=np.ones(d)
sampled_multimod, sigma_multimod, accept_multimod, sigma_list_multimod, l_list_multimod  = S



                                                                    verbose=False, adaptive=Fal
                                                                    burnin=burnin, max_iter=max


sampled_multimod_adapt, sigma_multimod_adapt, accept_multimod_adapt, sigma_list_multimod_ada
```

```
                                                    verbose=False, adaptive=Tru
                                                    burnin=burnin, max_iter=max
```

In [36]:

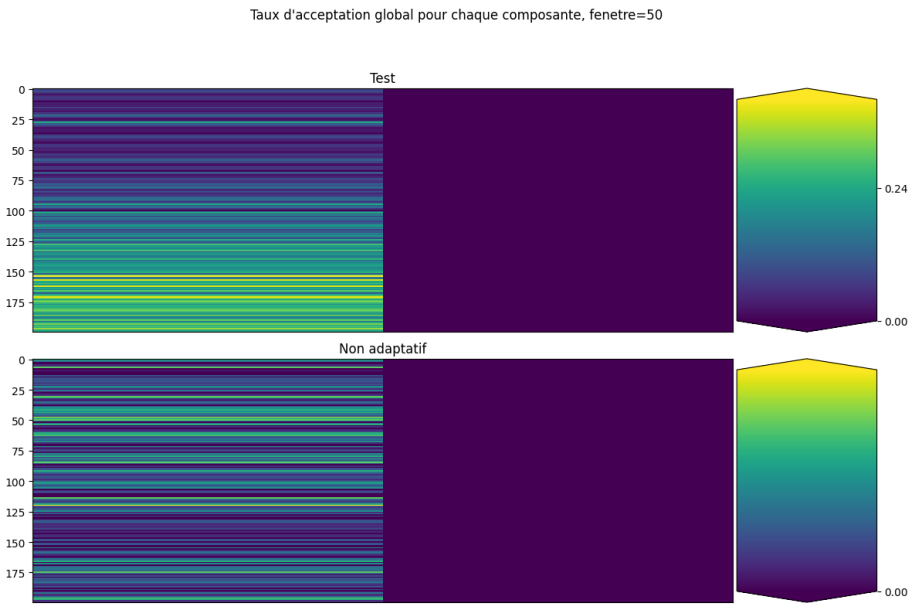
```
fig ,(ax1,ax2)= plt.subplots(2)
X, Y = np.meshgrid(np.linspace(0, 10, 100),np.linspace(0, 10, 100))
Z = target_pi_gmm(X, Y)
ax1.contour(X, Y, Z)
ax1.set_title('MH non adaptatif')
ax1.scatter(sampled_multimod[:,0], sampled_multimod[:,1],c='r')

ax2.contour(X, Y, Z)
ax2.set_title('MH adaptatif')
ax2.scatter(sampled_multimod_adapt[:,0], sampled_multimod_adapt[:,1],c='r')
plt.show()
```



In [37]:

```
fig,(ax1, ax2)=plt.subplots(2)
fig.suptitle(f'Taux d\'acceptation global pour chaque composante, {fenetre=}')
ax1.set_title('Test')
im1=ax1.imshow(accept_multimod_adapt,aspect='auto', interpolation='nearest')
divider1=make_axes_locatable(ax1)
```

```
cax1 = divider1.append_axes("right", size="20%", pad=0.05)
cbar1 = plt.colorbar(im1, cax=cax1, ticks=[0, 0.24, 0.5, 1],
                format="%.2f",
                extend='both')
ax1.xaxis.set_visible(False)
ax1.xaxis.set_visible(False)


ax2.set_title('Non adaptatif')
im2=ax2.imshow(accept_multimod,aspect='auto', interpolation='nearest')
divider2=make_axes_locatable(ax2)
cax2 = divider2.append_axes("right", size="20%", pad=0.05)
cbar2 = plt.colorbar(im2, cax=cax2, ticks=[0, 0.24, 0.5, 1],
                format="%.2f",
                extend='both')
ax2.xaxis.set_visible(False)
plt.tight_layout()
# Make space for title
plt.subplots_adjust(top=0.85)
plt.show()
```



On voit bien que dans le cas des distributions multimodales, aucune des deux méthodes n'est satisfaisante. La méthode adaptative fonctionne particulièrement peu.

On a beaucoup de mal à changer de mode. De plus pour le mode adaptatif, comme le taux d'acceptation est bas par nature on a tendance à faire encore

plus baisser ce taux en compensation, nous laissant coincés dans un seul mode.

Ici, les performances plus que correctes de l'algo adaptatif sont dues à un σ_{prop} assez élevé, qui pousse à visiter de nouveaux états.

2.B – Parallel Tempering¶

In [38]:

```
# Pour échantillonner le couple (i,j) uniformément entre 1 et K
def random_unif_couple(n_iter, radius):
    #on échantillonne sur le disque de dimension 2
    r_squared=np.random.uniform(0, np.square(radius), size=n_iter)
    theta=np.random.uniform(0, 2*np.pi, size=n_iter)
    a=np.abs([1+np.sqrt(r_squared)*np.cos(theta), 1+np.sqrt(r_squared)*np.sin(theta)])
    #Puis on récupère l'entier inférieur le plus proche (transformée inverse)
    return np.floor(a)
```

In [39]:

```
T=np.array([ 60, 21.6, 7.7, 2.8, 1])
def pi_temperature(X, t, pi=pi_gmm):
    return pi(X)**(1/t)

@np.vectorize
def target_pi_temperature(x, y, T):
    return pi_temperature([x, y], T)

def swap_temperatures(X, T, K, pi=pi_gmm,n=100):
    found=False
    while not found:
        list=random_unif_couple(n, K)
        for i in range(n):
            distance=np.abs(list[0][i]- list[1][i])
            if distance ==1:
                indexs=list[:,i]
                found=True
                break
    indexs=indexs.astype(int)
    num=pi_temperature(X[indexs[1]], numba.float64(T[indexs[0]]))*pi_temperature(X[indexs[0]
    denom=pi_temperature(X[indexs[0]], numba.float64(T[indexs[0]]))*pi_temperature(X[indexs[
    alpha=num/denom
    prob_choose=1 if 1<= alpha else alpha
    u=np.random.rand()
    if prob_choose>=u:
        X[[indexs[0], indexs[1]]]=X[[indexs[1], indexs[0]]]
    return X
```

```
def parallel_tempering(x, sigma, noyau, pi, T, max_iter):
    K = T.shape[0]
    a=None
    sampled = np.zeros((max_iter, T.shape[0], x.shape[0]))
    xk = np.zeros((T.shape[0], x.shape[0]))
    xk[:, :] = x
    for k in range(max_iter):
        for i in range(K):
            log_pi_temp=lambda x: np.log(pi_temperature(x, t=T[i]))
            #HM
            tau_i = 0.25 * np.sqrt(T[i])
            proposed, *_ = noyau(numba.float64(xk[i]),a,se=tau_i*np.ones(2),pi=log_pi_temp)
            xk[i]=proposed
        #Swap
        xk=swap_temperatures(xk, T, numba.int16(K))
        sampled[k] = xk
    return sampled
```

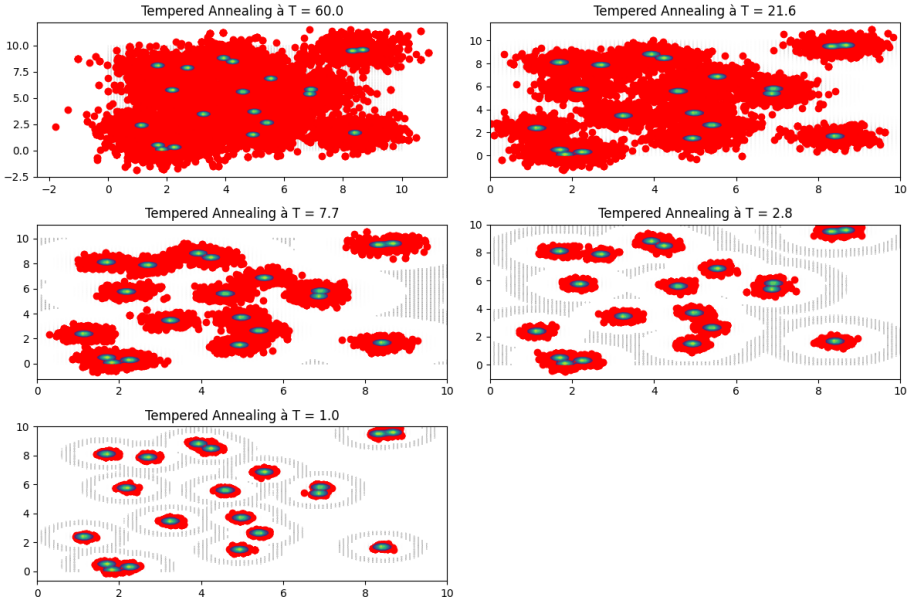
In [40]:

```
max_iter=10000
x = parallel_tempering(mu[0], sigma=Sigma_multimod, noyau=noyau_SRW, pi=log_pi_gmm, T=T, max
```

```
/tmp/ipykernel_235419/3636034072.py:38: RuntimeWarning: divide by zero encountered in log
  log_pi_temp=lambda x: np.log(pi_temperature(x, t=T[i]))
```

In [41]:

```
fig, axs=plt.subplots(nrows=3, ncols=2)
X, Y = np.meshgrid(np.linspace(0, 10, 100),np.linspace(0, 10, 100))
Z = target_pi_gmm(X, Y)
for i in range(T.shape[0]):
    local=target_pi_temperature(X,Y, T[i])
    ax=axs.flat[i]
    ax.scatter(X,Y,local,alpha=0.4,color='0.5')
    ax.contour(X, Y, Z)
    ax.set_title(f'Tempered Annealing à T = {T[i]}')
    ax.scatter(x[:,i,0], x[:,i,1],c='r')

for ax in axs.flat:
    if not bool(ax.has_data()):
        fig.delaxes(ax)
plt.tight_layout()
plt.show()
```



On voit bien que l'algorithme d'annealing fonctionne beaucoup mieux pour les distributions multimodales

Exercice 3 - Analyse bayésienne d'un modèle à effets aléatoires univariés¶

Question 1¶

On cherche écrire a une constante de normalisation près la vraisemblance $q(X, \mu, \sigma^2, \tau^2|Y)$.

En utilisant le théorème de Bayes et en passant au log : $$\begin{aligned} \log q(X, \mu, \sigma^2, \tau^2|Y) &\propto q(X, \mu, \sigma^2,\tau^2, Y)\\ &= \log q(Y|X, \mu, \sigma^2, \tau^2) + \log q(X| \mu, \sigma^2, \tau^2) + \log \underbrace{q}_{\pi_{prior}}(\mu, \sigma^2, \tau^2) \end{aligned}$$

On peut reformuler les 3 termes :

$$\begin{aligned} &\log q(Y|X, \mu, \sigma^2, \tau^2) &&\propto \sum_{i=1}^N \sum_{j=1}^{k_i} -\dfrac{1}{2} \log\tau^2 - \dfrac{1}{2\tau^2}(x_i - y_{i,j})^2 \quad\quad {\rm \text{car les X sont i.i.d, et le bruit}}~\varepsilon_{i,j}~{\rm \text{centré en 0, donc}}~y_{i,j}\sim\mathcal{N}(x_i, \tau^2) }\\ & &&\propto -\dfrac{k}{2} \log\tau^2 - \dfrac{1}{2\tau^2}\sum_{i=1}^N \sum_{j=1}^{k_i}(x_i - y_{i,j})^2\\ &\log q(X|\mu, \sigma^2, \tau^2) &&\propto -\dfrac{N}{2} \log \sigma^2 - \dfrac{1}{2\sigma^2} \sum_{i=1}^N \left(x_i - \mu\right)^2\\ &\log \pi_{prior}(\mu, \sigma^2, \tau^2)

&&\propto -\beta(\sigma^{-2}+\tau^{-2}) -(1+\alpha)\log \sigma^2 - (1+\gamma)\log \tau^2\\ \end{aligned}$$

D'où, en sommant les 3 log-vraisemblances et en rassemblant les termes : $$\log q(X, \mu, \sigma^2, \tau^2|Y) \propto -(\dfrac{N}{2}+1+\alpha)\log \sigma^2 -(\dfrac{k}{2}+\gamma+1)\log \tau^2 -\beta(\sigma^{-2}+\tau^{-2}) - \dfrac{1}{2\tau^2}\sum_{i=1}^N \sum_{j=1}^{k_i}(x_i - y_{i,j})^2- \dfrac{1}{2\sigma^2} \sum_{i=1}^N \left(x_i - \mu\right)^2$$

Question 2¶

Pour l'échantillonneur de Gibbs on doit connaître la distribution de chaque variable sachant toutes les autres. Pour cela on va partir de la loi postérieure calculée question 1 : $$\begin{aligned} & \log q(X~|~\mu, \sigma^2, \tau^2, Y) &&\propto - \dfrac{1}{2\tau^2}\sum_{i=1}^N \sum_{j=1}^{k_i}(x_i - y_{i,j})^2- \dfrac{1}{2\sigma^2} \sum_{i=1}^N \left(x_i - \mu\right)^2\\ & \log q(\mu~|~X, \sigma^2, \tau^2, Y) &&\propto- \dfrac{1}{2\sigma^2} \sum_{i=1}^N \left(x_i - \mu\right)^2\\ & \log q(\sigma^2 ~|~ X,\mu, \tau^2,Y) &&\propto -(\dfrac{N}{2}+1+\alpha)\log \sigma^2 - \dfrac{\beta}{\sigma^2} - \dfrac{1}{2\sigma^2} \sum_{i=1}^N \left(x_i - \mu\right)^2\\ & &&\propto -(\dfrac{N}{2}+\alpha +1)\log \sigma^2 - \dfrac{\beta + 0.5+ \sum_{i=1}^N \left(x_i - \mu\right)^2}{2\sigma^2} \end{aligned}$$ $\text{On reconnaît la log-distribution d'une Inverse Gamma} ~(\dfrac{N}{2}+\alpha, \beta + 0.5+ \sum_{i=1}^N \left(x_i - \mu\right)^2)$

$$\begin{aligned} & \log q(\tau^2 ~|~ X,\mu, \sigma^2,Y) &&\propto - (\dfrac{k}{2}+1+\gamma)\log \tau^2 - \dfrac{\beta}{\tau^2} - \dfrac{1}{2\tau^2} \sum_{i=1}^N \sum_{j=1}^{k_i} \left(x_i - y_{i,j}\right)^2\\ & &&\propto -(\dfrac{k}{2}+\gamma +1)\log \tau^2 - \dfrac{\beta + 0.5+ \sum_{i=1}^N \sum_{j=1}^{k_i} \left(x_i - y_{i,j}\right)^2}{2\sigma^2} \\ \end{aligned}$$ $\text{De même, on reconnaît la log-distribution d'une Inverse Gamma} ~(\dfrac{k}{2}+\gamma, \beta + 0.5+ \sum_{i=1}^N \sum_{j=1}^{k_i} \left(x_i - y_{i,j}\right)^2)$

Concernant les deux premières distributions :

- En passant à l'exponentielle, on peut remarquer que $$\begin{aligned} &q(X~|~\mu, \sigma^2, \tau^2, Y) &&\propto \exp \left(- \dfrac{1}{2\tau^2}\sum_{i=1}^N \sum_{j=1}^{k_i}(x_i - y_{i,j})^2- \dfrac{1}{2\sigma^2} \sum_{i=1}^N \left(x_i - \mu\right)^2 \right)\ \newline & &&\propto \prod_{i=1}^N \exp \left(- \dfrac{1}{2\tau^2}\sum_{j=1}^{k_i}(x_i - y_{i,j})^2- \dfrac{1}{2\sigma^2}\left(x_i - \mu\right)^2 \right) \quad \text{Qui semble Gaussien, on développe :}\ & &&\propto \prod_{i=1}^N \exp \left(
- \dfrac{1}{2\tau^2}\sum_{j=1}^{k_i}(x_i^2 - 2x_i y_{i,j} +y_{i,j}^2)- \dfrac{1}{2\sigma^2}\left(x_i^2 - 2x_i \mu + \mu^2\right) \right) \quad \text{On réunit les termes en x}\ & &&\propto \prod_{i=1}^N \exp \left(

-x_i^2\underbrace{(\dfrac{k_i}{2\tau^2}+\dfrac{1}{2\sigma^2})}{\dfrac{\sigma^2 k_i+\tau^2}{2\tau^2\sigma^2}} + 2x_i \left(\dfrac{\sum_{j=1}^{k_i} y_{i,j}}{2\tau^2} + \dfrac{\mu}{2\sigma^2}\right)- \left(\dfrac{\sum_{j=1}^{k_i} y_{i,j}^2}{2\tau^2} + \dfrac{\mu^2}{2\sigma^2} \right) \right)\ \newline & &&\propto \prod_{i=1}^N \exp \left(- \dfrac{\dfrac{2\tau^2\sigma^2}{\sigma^2 k_i+\tau^2}}{\dfrac{2\tau^2\sigma^2}{\sigma^2 k_i+\tau^2}} \left(x_i^2(\dfrac{k_i}{2\tau^2}+\dfrac{1}{2\sigma^2}) - 2x_i \left(\dfrac{\sum_{j=1}^{k_i} y_{i,j}}{2\tau^2} + \dfrac{\mu}{2\sigma^2}\right)+ \left(\dfrac{\sum_{j=1}^{k_i} y_{i,j}^2}{2\tau^2} + \dfrac{\mu^2}{2\sigma^2} \right) \right) \right)\ \newline & && \propto \prod_{i=1}^N \exp \left(- \dfrac{ x_i^2 - 2 x_i \left(\dfrac{\sigma^2 \sum_{j=1}^{k_i} y_{i,j} + \tau^2 \mu}{\sigma^2 k_i + \tau^2} \right) + \overbrace{\left(\dfrac{\sigma^2 \sum_{j=1}^{k_i} y_{i,j}^2 + \tau^2 \mu^2}{\sigma^2 k_i + \tau^2} \right)}^{\text{Indépendant de }~x_i} } {2\times\dfrac{\tau^2\sigma^2}{\sigma^2 k_i+\tau^2}} \right) \end{aligned}$$

À constante près indépendante de x_i, on peut écrire que $$ \left(x_i- \dfrac{\sigma^2 \sum_{j=1}^{k_i} y_{i,j} + \tau^2 \mu}{\sigma^2 k_i + \tau^2} \right)^2 \propto x_i^2 - 2 x_i \left(\dfrac{\sigma^2 \sum_{j=1}^{k_i} y_{i,j} + \tau^2 \mu}{\sigma^2 k_i + \tau^2} \right) $$

D'où, en reprenant cette relation de droite à gauche $$ \begin{aligned} &q(X~|~\mu, \sigma^2, \tau^2, Y) &&\propto \prod_{i=1}^N \exp \left(\dfrac{ \left(x_i- \dfrac{\sigma^2 \sum_{j=1}^{k_i} y_{i,j} + \tau^2 \mu}{\sigma^2 k_i + \tau^2} \right)^2 } { 2\times\dfrac{\tau^2\sigma^2}{\sigma^2 k_i+\tau^2} } \right) \end{aligned} $$

Les x_i étant i.i.d on voit ici la loi de X est proportionnelle au produit de lois gaussiennes $\mathcal{N}_i\left(\dfrac{\sigma^2 \sum_{j=1}^{k_i} y_{i,j} + \tau^2 \mu}{\sigma^2 k_i + \tau^2},\dfrac{\tau^2\sigma^2}{\sigma^2 k_i+\tau^2}\right)$

- On peut remarquer que
- $$\begin{aligned} &q(\mu~|~X, \sigma^2, \tau^2, Y) &&\propto \exp \left(- \dfrac{1}{2\sigma^2} \sum_{i=1}^N \left(x_i - \mu\right)^2\right)\\ \newline & &&\propto \exp \left(- \dfrac{1}{2\sigma^2} \sum_{i=1}^N \left(x_i^2 - 2x_i\mu + \mu^2\right)\right)\\ & &&\propto \exp \left(- \dfrac{1}{2\sigma^2} \sum_{i=1}^N \left(x_i^2 - 2x_i\mu\right) + N\mu^2)\right)\\\newline & &&\propto \exp \left(- \dfrac{1}{2\sigma^2} N \left[\overbrace{\sum_{i=1}^N x_i^2}^{\text{Indépendant de }~\mu} - 2\mu \left(\sum_{i=1}^Nx_i \right)+ \mu^2\right]\right)\\\newline & &&\propto \exp \left(- \dfrac{\left(\dfrac{\sum_{i=1}^Nx_i}{N}- \mu\right)^2}{2\times \dfrac{\sigma^2}{N} }\right) \quad \text{Avec le même raisonnement que plus haut} \end{aligned}$$

Il vient donc que $\mu \sim \mathcal{N}\left(\dfrac{\sum_{i=1}^Nx_i}{N},

\dfrac{\sigma^2}{N}\right)$

In [42]:

```
def gibbs_sampler_random(Y,alpha,beta,gamma,n_iter):
    N,K=Y.shape
    k_total=N*K
    mu=np.zeros(n_iter)
    #Sigma et Tau sont les sigma et tau carré
    sigma=np.ones(n_iter)
    tau=np.ones(n_iter)
    X=np.zeros((n_iter,N))
    Y_sum=Y.sum(axis=1)
    for n in range(n_iter):
        #On update d'abord les invgamma
        a=alpha+N/2
        b=beta+.5+.5*np.sum(np.square(X[n]-mu[n]))
        sigma[n]=stats.invgamma.rvs(a,scale=b)
        a=gamma+k_total/2
        b=beta+.5+.5*np.sum(np.square(X[n].reshape(-1,1)-Y))
        tau[i]=stats.invgamma.rvs(a,scale=b)

        #Puis les normales
        mu[n]=np.random.normal(loc=np.mean(X[n]), scale=np.sqrt(sigma[n]/N))

        new_var=np.sqrt(sigma[n]*tau[n]/(sigma[n]*K+tau[n]))
        new_mean=(sigma[n]*np.sum(Y, axis=1)+tau[n]*mu[n])/(sigma[n]*K+tau[n])
        X[n]=np.random.normal(loc=new_mean, scale=new_var)

    return sigma,tau,mu,X
```

Question 3¶

On introduit la distribution jointe de X et μ pour l'échantillonnage par bloc. D'après la question 1, on a : $$ \begin{aligned} \log q(X,\mu~|~\sigma^2, \tau^2, Y) &\propto - \dfrac{1}{2\tau^2}\sum_{i=1}^N \sum_{j=1}^{k_i}(x_i - y_{i,j})^2- \dfrac{1}{2\sigma^2} \sum_{i=1}^N \left(x_i - \mu\right)^2\\\newline &\propto \log q(X~|~\mu,\sigma^2, \tau^2, Y)\\ &\propto \exp \left(-\dfrac{\sum_{i=1}^N \sum_{j=1}^{k_i} (x_i^2 -2 x_i y_{i,j} + y_{i,j}^2)}{2\tau^2} -\dfrac{\sum_{i=1}^N(x_i^2 -2x_i \mu+ \mu^2)}{2\sigma^2}\right) \quad \text{On regroupe comme plus haut:}\\ & \propto \exp \left(-\sum_{i=1}^Nx_i^2\dfrac{\sigma^2 k_i+\tau^2}{2\tau^2\sigma^2} + 2\sum_{i=1}^N x_i \left(\sum_{j=1}^{k_i} \dfrac{y_{i,j}}{2\tau^2} + \dfrac{\mu}{2\sigma^2}\right)- \sum_{i=1}^N\left(\overbrace{ \dfrac{\sum_{j=1}^{k_i} y_{i,j}^2}{2\tau^2} }^{\text{Indépendant de }~x_i,\mu}+ \dfrac{\mu^2}{2\sigma^2} \right) \right)\\ \newline & \propto \exp \left(- \left(\dfrac{\sigma^2 k_i+\tau^2}{2\tau^2\sigma^2}\sum_{i=1}^Nx_i^2

- 2\dfrac{\sum_{j=1}^{k_i} y_{i,j}}{2\tau^2}\sum_{i=1}^N x_i - 2\dfrac{\mu}{2\sigma^2}\sum_{i=1}^Nx_i + \sum_{i=1}^N\dfrac{\mu^2}{2\sigma^2} \right) \right)\\ \newline & \propto \exp \left(- \dfrac{1}{2} \left(\dfrac{\sigma^2 k_i+\tau^2}{\tau^2\sigma^2}\sum_{i=1}^Nx_i^2 - 2\dfrac{\sum_{j=1}^{k_i} y_{i,j}}{\tau^2}\sum_{i=1}^N x_i - 2\dfrac{\mu}{\sigma^2}\sum_{i=1}^Nx_i + \sum_{i=1}^N\dfrac{\mu^2}{\sigma^2} \right) \right)\\ \newline \end{aligned} $$

Comme x et μ suivent une loi normale, on reconnaît ici un produit proportionnel au produit de log-densités de loi normales multivariées, de dimension $n+1$, soit $dim(x)+dim(\mu)=n+1$.

On peut aussi identifier les coefficients de la matrice de Σ^{-1} : pour $\Sigma^{-1} \in S_{n+1}^{++}$ on a $V^T\Sigma^{-1}V = \sum_i\sum_j \Sigma^{-1}_{ij}V_iV_j$.

Si on prend $V = (X,\mu) $, et en supposanant que la loi multivariée est centrée en 0, on a : $$ \begin{aligned} \Sigma^{-1}_{ii} &= \dfrac{\sigma^2 k_i x+ \tau^2}{\tau^2 \sigma^2} ~\forall ~i \leq n \quad &&\text{En identifiant les } ~x_i^2\\ \Sigma^{-1}_{n+1,n+1}&=\dfrac{N}{\sigma^2} &&\text{En identifiant}~\mu^2\\ \Sigma^{-1}_{i,n+1}&=-\dfrac{1}{\sigma^2} ~\forall ~i \leq n &&\text{En identifiant}~\mu\sum_{i=1}^Nx_i\\ \Sigma^{-1}_{n+1,i}&=-\dfrac{1}{\sigma^2} ~\forall ~i \leq n &&\text{Par symétrie}\\ \end{aligned}$$

En résumant, on a quelque chose comme cela : $$ \Sigma^{-1}(V,V) = \begin{pmatrix} \dfrac{\sigma^2 k + \tau^2}{\tau^2 \sigma^2} & 0 & \cdots & 0 & \dfrac{-1}{\sigma^2} \\ 0 & \ddots & 0 & 0 & \vdots \\ \vdots & 0 & \ddots& 0 & \vdots \\ 0 & 0 & 0 & \dfrac{\sigma^2 k + \tau^2}{\tau^2 \sigma^2} & \dfrac{-1}{\sigma^2} \\ -\dfrac{1}{\sigma^2} & \cdots & \cdots & -\dfrac{1}{\sigma^2} & \dfrac{N}{\sigma^2} \\ \end{pmatrix} $$

Il ne nous manque que le terme en $- 2\dfrac{\sum_{j=1}^{k_i} y_{i,j}}{\tau^2}\sum_{i=1}^N x_i$ qu'on va faire rentrer dans la moyenne de la loi. On note $\mathcal{C}$ la moyenne de $q(X,\mu~|~\sigma^2, \tau^2, Y)$, et on a : $$ \Sigma^{-1} \mathcal{C}= [\dfrac{\sum_{j=1}^{k_1} y_{1j}}{\tau^2}, ... ,\dfrac{\sum_{j=1}^{k_N} y_{Nj}}{\tau^2}, 0] $$

D'où

$$ \mathcal{C}= \Sigma [\dfrac{\sum_{j=1}^{k_1} y_{1j}}{\tau^2}, ... ,\dfrac{\sum_{j=1}^{k_N} y_{Nj}}{\tau^2}, 0] $$

On a donc que la distribution jointe de x et μ est : $$ q(X,\mu~|~\sigma^2, \tau^2, Y)\sim\mathcal{N}(\mathcal{C}, \Sigma) $$

In [43]:

```
def update_cov(sigma, tau, k, N):
    cov = np.zeros((N+1,N+1))
    diag_idx = np.diag_indices(N+1)
```

```
    cov[diag_idx] = (k*sigma + tau)/(sigma* tau)
    cov[:,N]= -1/sigma
    cov[N,:] = -1/sigma
    cov[N,N] = N/sigma
    return np.linalg.pinv(cov)

@njit
def update_mean(cov, Y, tau):
    return cov@Y/tau
```

In [44]:

```
def block_gibbs_sampler_random(Y,alpha,beta,gamma,n_iter):
    N,K=Y.shape
    k_total=N*K
    mu=np.zeros(n_iter)
    #Sigma et Tau sont les sigma et tau carré
    sigma=np.ones(n_iter)
    tau=np.ones(n_iter)
    X=np.zeros((n_iter,N))
    Y_sum=np.zeros(N+1)
    Y_sum[:N]=np.sum(Y, axis=1).reshape(N)
    for n in range(n_iter):
        #On update d'abord les invgamma
        a=alpha+N/2
        b=beta+.5+.5*np.sum(np.square(X[n]-mu[n]))
        sigma[n]=stats.invgamma.rvs(a,scale=b)

        a=gamma+k_total/2
        b=beta+.5+.5*np.sum(np.square(X[n].reshape(-1,1)-Y))
        tau[n]=stats.invgamma.rvs(a,scale=b)

        #Puis le bloc (x, mu)
        Sigma=update_cov(sigma[n], tau[n], K, N)
        Mean=update_mean(Sigma,Y_sum,tau[n])
        V=np.random.multivariate_normal(mean=Mean,cov=Sigma,size=1).reshape(N+1)
        X[n]=V[:N]
        mu[n]=V[-1]
    return sigma,tau,mu,X
```

L'échantillonneur de Gibbs par bloc devrait être plus performant car les blocs sont bien choisis (variables qui évoluent ensemble, calcul de la log_likelihood exactement pareil...). Par contre, on passe d'un échantillonnage en 1D (np.random.normal) à un échantillonnage en n+1 D, donc la rapidité d'exécution risque d'être impactée.

In [45]:

```
alpha = 1
beta =  1
gamma = 1

mu_true=5
sigma_true=1
tau_true=0.5

mu0=0.5
sigma0=2
tau0=0.5
#On considère k_i=cste=K \forall i
K = 1000
N = 500
Y = np.zeros((N,K))
for i in range(N):
    X_aux = np.random.normal(loc=mu_true,scale=sigma_true)
    eps_aux = np.random.normal(loc=0,scale=tau_true,size=K)
    Y[i,:] = X_aux + eps_aux

n_iter=20000
```

In [46]:

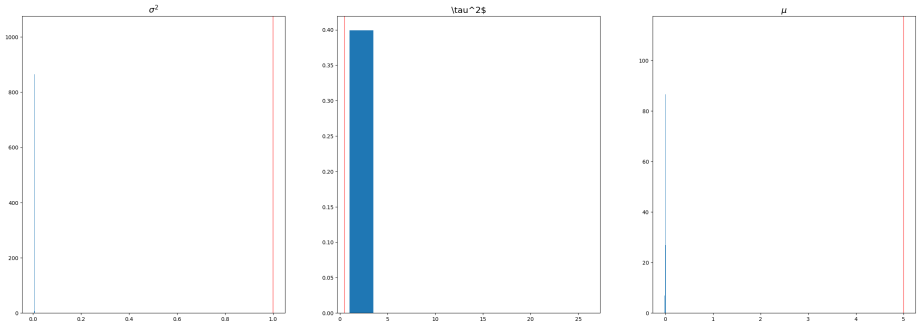
```
sigma_random,tau_random,mu_random,X_random=gibbs_sampler_random(Y,alpha,beta,gamma,n_iter)
```

In [47]:

```
sigma_adapt_random,tau_adapt_random,mu_adapt_random,X_adapt_random=block_gibbs_sampler_rando
```

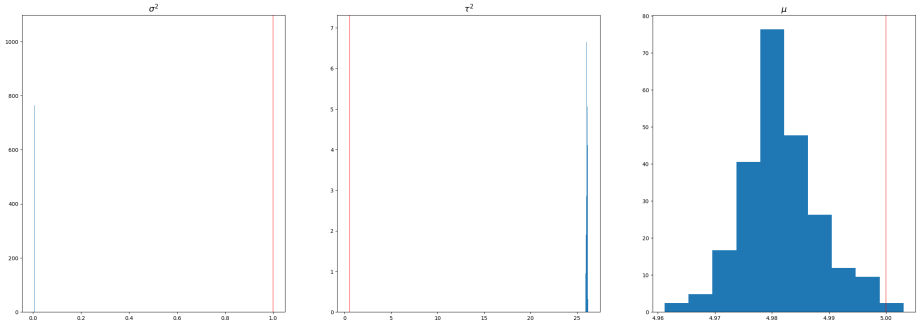
In [53]:

```
_,ax= plt.subplots(1,3,figsize=(30,10))

ax[0].hist(sigma_random,density=True)
ax[0].set_title('$\sigma^2$', fontsize =16)
ax[0].axvline(sigma_true, color='red',alpha=0.5)
ax[1].hist(np.array(tau_random),density=True)
ax[1].set_title(r'\tau^2$', fontsize =16)
ax[1].axvline(tau_true, color='red',alpha=0.5)
ax[2].hist(mu_random,density=True)
ax[2].set_title('$\mu$', fontsize =16)
ax[2].axvline(mu_true, color='red',alpha=0.5)
plt.show()
```



In [52]:

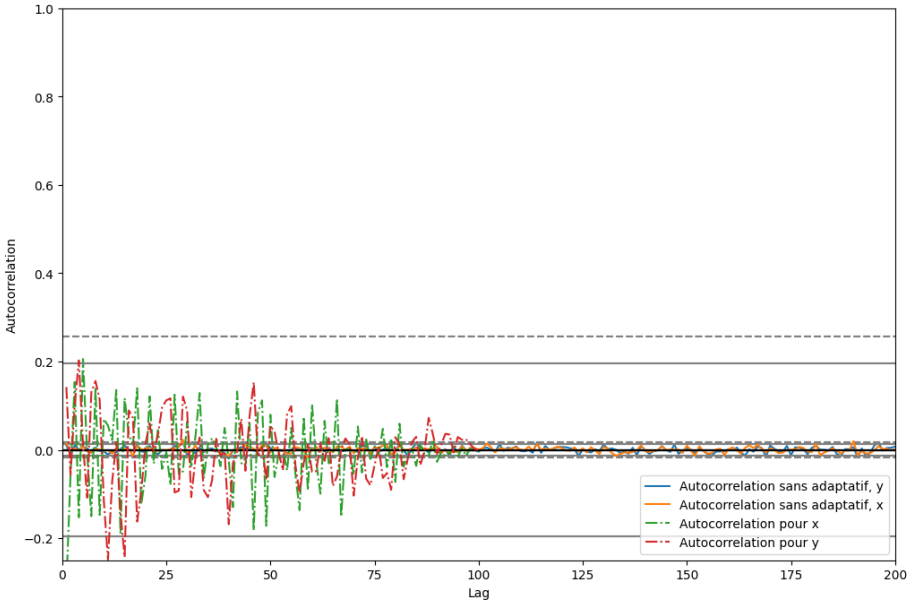
```
_,ax= plt.subplots(1,3,figsize=(30,10))

ax[0].hist(sigma_adapt_random,density=True)
ax[0].set_title('$\sigma^2$', fontsize =16)
ax[0].axvline(sigma_true, color='red',alpha=0.5)

ax[1].hist(np.array(tau_adapt_random),density=True)
ax[1].set_title(r'$\tau^2$', fontsize =16)
ax[1].axvline(tau_true, color='red',alpha=0.5)

ax[2].hist(mu_adapt_random,density=True)
ax[2].set_title('$\mu$', fontsize =16)
ax[2].axvline(mu_true, color='red',alpha=0.5)

plt.show()
```



In [55]:

```
x_test=pd.Series(X_random[:,0])
y_test=pd.Series(X_random[:,1])
autox_test = pd.plotting.autocorrelation_plot(x_test, label='Autocorrelation sans adaptatif
autoy_test = pd.plotting.autocorrelation_plot(y_test, label='Autocorrelation sans adaptatif
autox_test.plot(alpha=0.5)
```

```
autoy_test.plot()
x=pd.Series(X_adapt_random[:,0])
y=pd.Series(X_adapt_random[:,1])
autox = pd.plotting.autocorrelation_plot(x, label='Autocorrelation pour x ',ls='dashdot')
autoy = pd.plotting.autocorrelation_plot(y, label='Autocorrelation pour y',ls='dashdot')
autox.plot()
autoy.plot()
plt.xlim((0,200))
plt.ylim((-0.25,1))
plt.legend(loc='lower right')
plt.show()
```



Même avec 100 itérations, l'algorithme adaptatif semble diverger des bonnes valeurs de tau et sigma.

Comme pour les lois en 20D, l'autocorrélation diminue drastiquement par rapport à l'algorithme non adaptatif.