Analyse de données de Single Molecule

Timothé Ringeard

30 mars 2026

Dans ce document on détaille plusieurs façons d'analyser les données provenant du cadre de Single Molecule. On explique les concepts mathématiques derrière les traitements de données, et donne du code en Python pour faire ces traitements de données. On part du modèle smPRESS, on explique comment les données correspondent au modèle, mais aussi les limites que l'on peut lui trouver.

Le code pour générer les figures ce ce document est disponible dans le fichier `Figures.py`.

Ce n'est pas un rapport à proprement parler, donc je me permet parfois des remarques peu formelles.

Table des matières

Les Sections 1, 2 et 3 sont presque entièrement dédiées à expliquer les outils mathématiques et informatiques qui servent à l'étude des données de Singe Molecule. La Section 4 confronte ce modèle avec les données.

1 Format des données

A partir des films, il y a essentiellement deux types de données que l'on récupère
— La première est le nombre de molecules visibles en fonction du temps.
— La deuxième est, pour chaque molécule visible, son temps de vie et son temps de mort.

La deuxième donnée est beaucoup moins robuste, car elle nécéssite de déterminer si deux tâches sur des frames consécutives correspondent à la même molécule, ou à des molécules différentes. On appelle *trace* d'une molécule l'intervalle entre son temps de naissance et son temps de mort.

Voici quelques packages Python standard :

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.optimize as sco
```

— Le package `numpy` sert à la manipulation de tableaux.
— Le package `matplotlib.pyplot` sert à créer des figures.
— Le package `scipy.optimize` contient deux fonctions utiles : `scipy.optimize.minimize`, qui implémente un algorithme de minimisation de fonction, et `scipy.optimize.curve_fit`, qui implémente un algorithme qui cherche les meilleurs paramètres pour qu'une courbe paramétrée colle aux données.

Les données que l'on manipule se trouvent sous trois formes différentes.
— Pour le smPRESS, les données consistent en un tableau à deux colonnes et N lignes. La première colonne correspond au temps écoulé, et la deuxième au nombre de molécules présentes à ce temps. En Python on peut y accéder comme ceci (en modifiant le chemin d'accès dans le PC, si les fichiers sont au même endroit que le code il est possible qu'il ne soit pas nécessaire de spécifier tout le chemin)

```
data = np.loadtxt("C:/Users/timor/Desktop/Stage_RobinLab/Data/fbr92/
    NN_vs_t_avg1.txt")
t, N = data[:, 0], data[:, 1]
```

— Pour le Tracking, les données consistent en un tableau à quatres colonnes et N lignes. La première colonne est un identifiant de la particule, la deuxième est son temps de naissance (numéro de la première frame où il apparaît), la troisième est son temps de mort (numéro de la dernière frame où elle apparaît), et la dernière est la durée d'une frame (c'est une constante). En Python, on peut y accéder comme ceci :

```
tracking = np.loadtxt("C:/Users/timor/Desktop/Stage_RobinLab/Data/
    fbr92_C1/tau_avg2_fbr92_C1.txt")
```

```
# First frame in which the particle is seen
birth_frame = tracking[:,1]
# First frame in which the particle is NOT seen
death_frame = tracking[:,2] + 1
# Length of the particle lifetime, in seconds
resitime = tracking[:,3] * (tracking[:,2] - tracking[:,1] + 1)
```

— Enfin, une troisième forme, plus générale consiste en un tableau avec 9 colonnes et `N_lines` lignes. Chaque ligne correspond à un spot detecté sur une frame. La liaison entre les spots (de manière à créer des traces) a déjà été faites. Dans les 9 colonnes on trouve : position x, position y, intensité, (?), (?), numéro de la frame depuis la première frame de la trace, temps de la frame, identifiant de la trace, durée de la trace en nombre de frames. Si l'on autorise les traces a sauter certaines frames, alors la colonne "numéro de la frame depuis la première frame de la trace" prendra par exemple les valeurs 1,2,3,5,6,8,9 le long d'une trace où les frames 4 et 7 ont été sautées.

```
data = np.loadtxt("C:/Users/timor/Desktop/Stage_RobinLab/Data/
    Film_2_pop/table.txt")
N_lines = len(data[:,0]) # number of lines
x_pos = data[:,0]
y_pos = data[:,1]
intensity = data[:,2]
frame_dur = data[:,5]
t_pos = data[:,6]
identifier = data[:,7]
duration = data[:,8]
```

Remarque : Pour importer des fichier de la forme `.mat`, on pourra faire comme ceci :

```
import scipy.io
data = scipy.io.loadmat("C:/Users/timor/Desktop/Stage_RobinLab/Data/
    Formins_Marjolaine/Nformins_raw.mat")
N = data['Temps202'][:,0]
```

2 smPRESS

2.1 Le modèle

Cette Section est un retranscription du Papier de François [1], Supplementary Material Note 1 "smPReSS equations : Measuring turn-over from single molecule loss during photobleaching".

On considère que les moélcules peuvent être dans deux compartiments : le cortex (R), et le cytoplasme (Y). Il y a trois réactions en jeu :

— Déplacement des molécules du cytoplasme vers le cortex, à taux k_{on}.
— Déplacement des molécules du cortex vers le cytoplasme, à taux k_{off}.
— Disparition (par photobleaching) des molecules du cortex, à taux k_{ph}.

Le modèle stipule que les densités de présence $Y(t), R(t)$, dans les deux compartiments sont

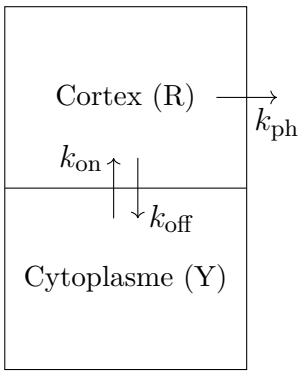

FIGURE 1 – Schématisation du modèle.

régies par

$$R'(t) = k_{\text{on}}Y(t) - (k_{\text{off}} + k_{\text{ph}})R(t), \tag{1}$$
$$Y'(t) = k_{\text{off}}R(t) - k_{\text{on}}Y(t). \tag{2}$$

On fait l'hypothèse qu'à l'instant 0, les densités sont à l'équilibre pour le même système, sans le photobleaching. On aurait donc $k_{\text{on}}Y(0) = k_{\text{off}}R(0)$. Quelques calculs nous amènent alors aux expressions suivantes :

$$R(t) = R(0)\left(\frac{k_{\text{ph}} + r_2}{r_2 - r_1}e^{r_1 t} + \frac{k_{\text{ph}} + r_1}{r_1 - r_2}e^{r_2 t}\right), \tag{3}$$

et

$$Y(t) = Y(0)\left(\frac{r_2}{r_2 - r_1}e^{r_1 t} + \frac{r_1}{r_1 - r_2}e^{r_2 t}\right), \tag{4}$$

où r_1, r_2 satisfont

$$r_1 = -\frac{1}{2}\left(k_{\text{on}} + k_{\text{off}} + k_{\text{ph}} + \sqrt{(k_{\text{on}} + k_{\text{off}} + k_{\text{ph}})^2 - 4k_{\text{on}}k_{\text{ph}}}\right), \tag{5}$$
$$r_2 = -\frac{1}{2}\left(k_{\text{on}} + k_{\text{off}} + k_{\text{ph}} - \sqrt{(k_{\text{on}} + k_{\text{off}} + k_{\text{ph}})^2 - 4k_{\text{on}}k_{\text{ph}}}\right), \tag{6}$$

ou, de manière équivalente,

$$k_{\text{on}} = \frac{r_1 r_2}{k_{\text{ph}}}, \tag{7}$$
$$k_{\text{off}} = -(r_1 + r_2) - (k_{\text{on}} + k_{\text{ph}}). \tag{8}$$

Typiquement, les courbes des valeurs de $R(t)$ au cours du temps ressemblent à celle de la Figure 2.

2.2 Déterminer les paramètres $k_{\text{off}}, k_{\text{ph}}, k_{\text{on}}$ à partir des données

On explique ici comment estimer les paramètres $k_{\text{off}}, k_{\text{ph}}, k_{\text{on}}$ à partir des données expérimentales. Pour cela, on détaille le fichier de code `smPRESS_functions.py`.

La fonction `sin_mol_fun` modélise le nombre de molécule visibles au cours du temps dans ce modèle. La fonction `sin_mol_fun_k` fait de même lais en prenant en paramètres k_{on}, k_{off} au lieu de r_1, r_2.

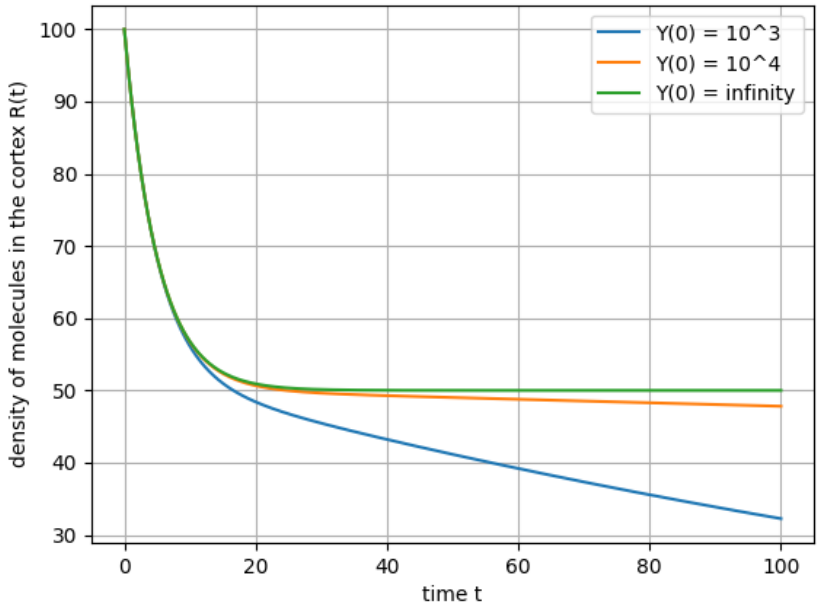

FIGURE 2 – Exemple de courbe smPRESS et comparaison avec une exponentielle simple, correspondant au cas d'un cytoplasme infini. Les paramètres sont ici $k_{\text{ph}} = 0.1, k_{\text{off}} = 0.1, R(0) = 100, Y(0) = 10^3, 10^4, \infty$.

Algo 1 – sin_mol_fun

```
def sin_mol_fun(tdata, y0, r1, r2, kph):
        return [y0*(kph+r2)/(r2-r1)*exp(r1*t) - y0*(kph+r1)/(r2-r1)*exp(r2*t
            ) for t in tdata]

def sin_mol_fun_k(tdata, y0, k_on, k_off, k_ph):
    f = k_on + k_off + k_ph
    g = sqrt(f*f - 4*k_on*k_ph)
    r1 = (-f - g)/2
    r2 = (-f + g)/2
    return sin_mol_fun(tdata, y0, r1, r2, k_ph)
```

On se donne une courbe expérimentale $N(t_1), \ldots, N(t_p)$, avec $t_1 = 0$, et tel que le système soit à l'équilibre à ce temps 0. La courbe N_{fit} de la forme (3) avec $N_{fit}(0) = N(0)$ qui colle le mieux à la courbe expérimentale est calculée par `get_fit_3_params`. Cette fonction prend facultativement en plus en argument un entier `avg_first`, qui correpond à prendre plutôt $N_{fit}(0) = \frac{1}{\text{avg\_first}} \sum_{i=1}^{\text{avg\_first}} N(t_i)$. Cela permet de réduite le bruit dû à la valeur en zéro de N.

Algo 2 – get_fit_3_params

```
def get_fit_3_params(t, N, avg_first=1):
    def sin_mol_fun3N(tdata, r1, r2, kph):
        # The function sin_mol_fun but with fixed value at zero
        N0 = np.mean(N[:avg_first])
        return sin_mol_fun(tdata, N0, r1, r2, kph)
```

```
    r10 = -10 * random()
    r20 = -10 * random()
    kph0 = 0.1
    rjump = 0.01
    sto = [[],[],[]]
    for i in range(10):
        r1, r2, kph = sco.curve_fit(sin_mol_fun3N, t, N, p0 = [r10, r20,
            kph0], bounds = ([-np.inf, -np.inf, 0], [0, 0, np.inf]))[0]
        kon = r1*r2/kph
        koff = -(r1+r2)-(kon+kph)
        sto[0].append(kon)
        sto[1].append(koff)
        sto[2].append(kph)
        r10 = r1 * exp(rjump*(random()-0.5))
        r20 = r2 * exp(rjump*(random()-0.5))
        kph0 = kph * exp(rjump*(random()-0.5))
        rjump = rjump*0.99
    return kon, koff, kph, r1, r2, sto
```

Notons que la signification de "coller mieux à la courbe initiale" est cachée dans le code par la fonction `scipy.curve_fit`, mais cela signifie (à peu de choses près) que c'est la courbe N_{fit} qui minimise 2

$$\sum_{i=1}^{p}(N(t_i)-N_{fit}(t_i))^2. \tag{9}$$

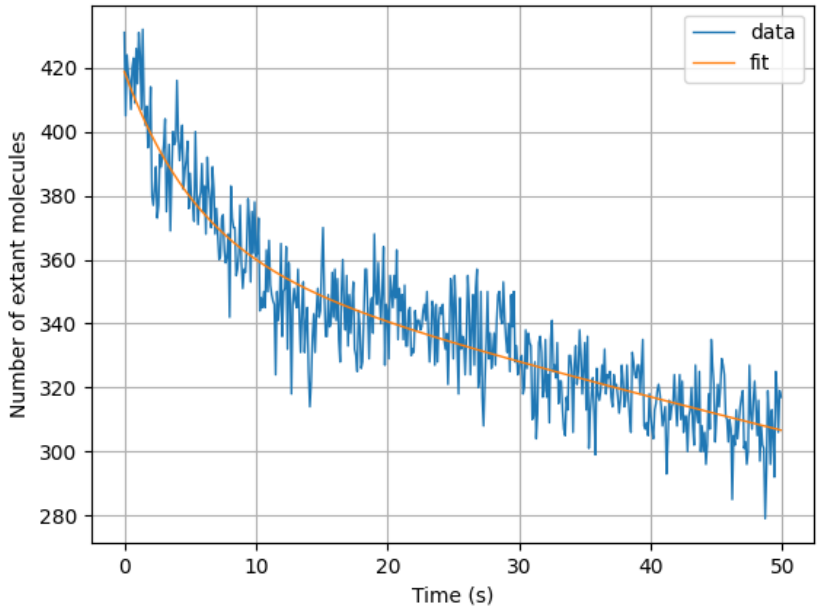

FIGURE 3 – La courbe orange est, parmi celles de la forme (3), celle qui colle le mieux aux données en bleu (en terme de somme des écarts au carré).

Pour avoir une idée de la robustesse des estimations de k_{on}, k_{off} et k_{ph}, on peut procéder comme suit. Etant donné $N(t_1),\dots,N(t_p)$ la courbe expérimentale, et $N_{fit}(t_1),\dots,N_{fit}(t_p)$ la courbe fittée,

on estime l'erreur moyenne autour de la courbe fittée par

$$\sigma = \sqrt{\frac{1}{p}\sum_{i=1}^{p}(N(t_i) - N_{fit}(t_i))^2}. \tag{10}$$

On cherche ensuite les paramètres $k_{\text{on}}^{up/down}$, $k_{\text{off}}^{up/down}$, $k_{\text{ph}}^{up/down}$ qui fittent au mieux les courbes $N^{up/down}(t) = N(t) \pm \sigma$. On a alors une idée de la variation de l'estimation de ces paramètres par une petite modification de la courbe. La fonction `get_up_down_estimate_k` implémente cette approche, elle est écrite en B.1.

Note : Je ne suis pas convaincu par la pertinence de cette méthode. Je pense que le biais induit par une translation de toute la courbe est assez faible sur l'estimation de k_{ph} par exemple, puisque k_{ph} correspond surtout à la pente en zéro de la courbe.

2.3 Simulations stochastiques

Le modèle décrit en 2.1 possède un analogue stochastique naturel, qui est le suivant :

On définit deux processus $R(t), Y(t)$ pour $t \geq 0$, à valeur dans $\mathbb{N}$, modélisant le nombre de molécule respectivement dans le cortex et le cytoplasme. Chaque molecule de Y saute dans R avec taux k_{on}, chaque molécule de R saute dans Y avec taux k_{off}, et chaque molécule de R disparaît avec taux k_{ph}. Ces trois processus sont indépendants.

Un algorithme de type Gillepsie permet de simuler efficacement ce processus. Cela est fait dans le fichier `Simulations_Gillepsie_smPRESS.py`. Puisque le système est supposément à l'état d'équilibre sans photobleaching dans l'état initial, on effectue la simulation en faite en deux étapes : la première étape consiste à éteindre le photobleaching et à attendre l'équilibre, et la deuixème étape consiste à allumer le photobleaching et simuler son effet. La vérfication que l'équilibre est atteint au bout de la première étape se fait en regardant les courbes du nombre de molécules dans le cortex et en s'assurant qu'elle sont stables. Il est possible de calculer explicitement les valeurs à l'équilibre pour les fournir en condition initiale, mais notre approche permet en plus de modéliser l'incertitude sur la condition initiale.

L'algorithme est donné en B.2

On utilisera `get_counts_simu` pour récupérer les données de la simulation seulement dans la deuxième phase, mais on peut d'abord utiliser `simul_2_phases_Gillepsie` pour vérifier que l'équilibre est atteint durant la première phase. Le paramètre `cyto_pool_0` correspond au nombre total de molécule mise dans le système à l'instant zéro

L'algorithme calcule en même temps les données de tracking, c'est à dire les temps de naissance et de mort de chaque particule. L'algorithme de Gillepsie à la particularité de ne pas être discrétisé en temps (à part par des contraintes de représentation numérique des réels en Python), donc ces temps de naissances et de mort peuvent être très proches.

Le nombre de molécules dans le cortex est de l'ordre de $\frac{k_{\text{off}}}{k_{\text{off}}+k_{\text{on}}}$`cyto_pool_0`. Lorsque cette quantité tend vers l'infini, la solution de l'équation stochastique converge vers la courbe déterministe (3). Cependant, si l'on fait tendre `cyto_pool_0` vers l'infini en maintenant $\frac{k_{\text{off}}}{k_{\text{off}}+k_{\text{on}}}$`cyto_pool_0` environ constant (par exemple en prenant $k_{\text{on}} \propto$ `cyto_pool_0` et k_{off} constant), alors il restera toujours une composante stochastique au système, même à la limite.

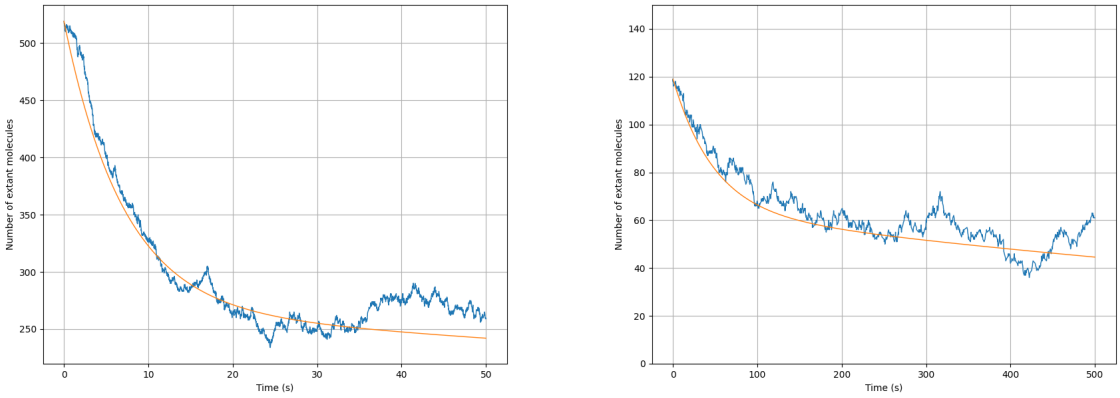

FIGURE 4 – Simulations stochastiques effectuées avec une algorithme de Gillepsie pour différentes valeur des paramètres $k_{\text{off}}, k_{\text{on}}, k_{\text{ph}}$. La limite lisse correspondant à (3) est indiquée en orange.

3 Tracking

Dans cette section, on donne plusieurs manières d'analyser les données provenenat du Tracking. On dispose des temps naissance $t_b(i)$ et de mort $t_d(i)$ pour n particules ("b" pour "birth", "d" pour "death"). On en déduit le temps de résidence $t_r(i) = t_d(i) - t_b(i)$ de chacune de ces particules.

3.1 Visualisation des temps de naissance, de mort, et de résidence

Une première façon de visualiser les données est de regarder l'histogramme des temps de résidence. Pour cela, il faut spécifier une granulosité τ, qui correspond à un temps. Comme les temps de naissance et de mort sont toujours multiples de la durée d'une frame (notée τ_{frame}), c'est aussi le cas des temps de résidence, et donc la granulosité la plus fine est celle de la durée d'une frame. On évitera de prendre une granulosité qui n'est pas un multiple de τ_{frame}, pour éviter les biais.

Les fonctions `bins_mid` et `bins_border` prennnent en argument une valeur minimale, une valeur maximale, un pas, et retournent les boîtes associées à cette découpe.

Algo 3 – get_bins_mid, get_bins_border

```
def get_bins_mid(mini, maxi, step):
    return np.linspace(mini, maxi, round((maxi-mini)/step)+1)

def get_bins_border(mini, maxi, step):
    return np.linspace(mini-step/2, maxi+step/2, round((maxi-mini)/step)+2)

# Exemples :
# bins_mid(0.5, 3, 0.5) = [0.5, 1.0, 1.5, 2.0, 2.5, 3]
# bins_border(0.5, 3, 0.5) = [0.25, 0.75, 1.25, 1.75, 2.25, 2.75, 3.25]
```

On peut créer l'histogramme, après avoir calculé `resitime`, avec la fonction `np.histogram`. On peut ensuite l'afficher en échelle logarithmique avec `plt.semilogy`, en spécifiant les milieux des boîtes `bins_mid(...)` pour les abscisses :

Algo 4 – Histogram plotting

```
hist = np.histogram(resitime, bins = bins_border(0.25, 50, 0.25))[0]
plt.semilogy(bins_mid(0.25, 50, 0.25), hist)
plt.grid(visible=True)
plt.xlabel('Time (s)')
plt.ylabel('Number of occurences')
plt.show()
```

J'ai rajouté quelques options au plot pour qu'il contienne des noms d'axes x et y, une grille. Voir la Figure 5 pour un exemple.

Pour comparer plusieurs échantillons qui n'ont pas forcément la même taille, on peut renormaliser cet histogramme en imposant que la valeur à zéro soit 1, ou bien en imposant que l'intégrale soit 1 (voir (16) pour plus de détails). Dans les sections suivantes, la fonction f correspondra à celle représentée dans cette Figure 5. Cette fonction est discrète, on ne connait que $f(\tau)$, $f(2\tau)$, ..., $f(N\tau)$, où N et τ dépendent des boîtes que l'on a choisie. On ne peut pas choisir τ arbitrairement petit car il faut avoir $\tau \geq \tau_{\text{frame}}$.

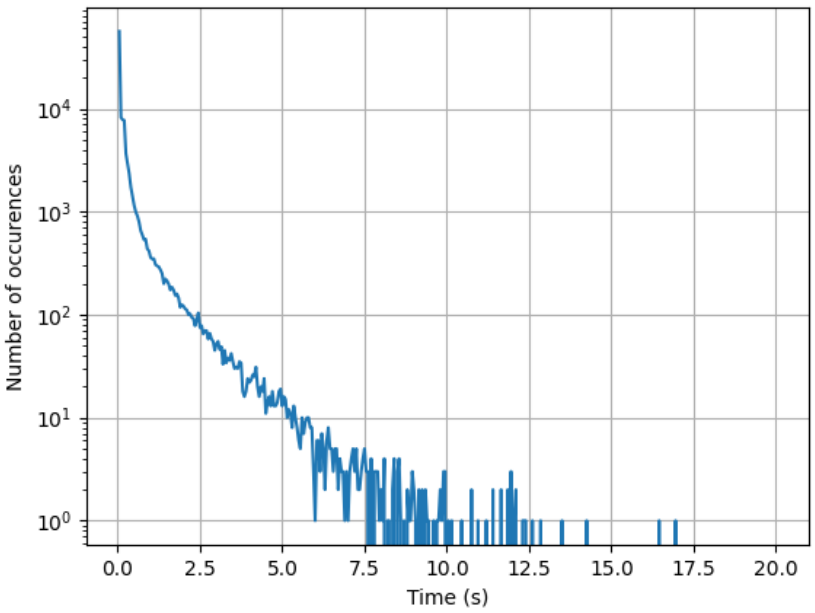

FIGURE 5 – Exemple d'histogramme de la durée de vie de molécules dans le cortex.

Une deuxième manière de visualiser les temps de résidence des molécules est d'en faire un nuage de point où un point correspond à une molécule, et a deux coordonnées : son temps de naissance et son temps de mort. voir la Figure 6

Algo 5 – Birth/Death point cloud

```
birth_time = birth_frame * 0.05 # A frame is 50ms
death_time = death_frame * 0.05
plt.scatter(birth_time, death_time, s=1) # The s parameter corresponds to
    the size of the dots
plt.xlabel('birth time')
plt.ylabel('death time')
```

```
plt.axis('square') # So that the plot is a square
plt.show()
```

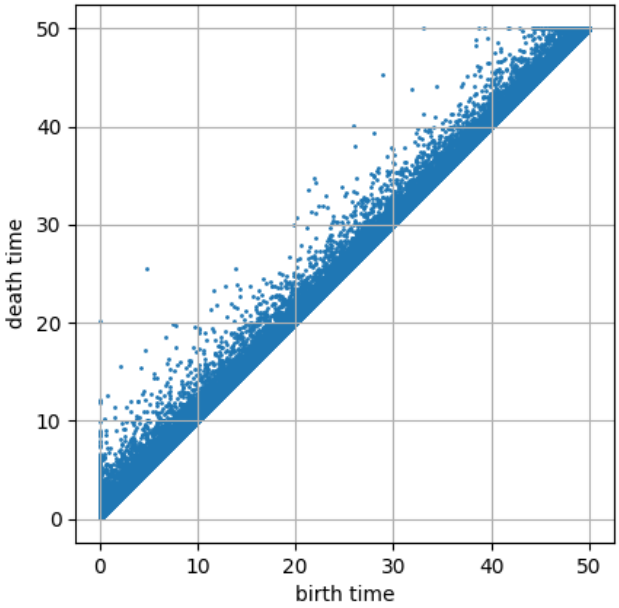

FIGURE 6 – Exemple de visualisation du temps de vie des molécules dans le cortex.

Cela permet bien de visualiser le souci au bord du domaine : les particules qui naissent à la première frame ou meurent à la dernière frame sont anormalement plus nombreuses que celles qui naissent ou meurent à une frame au milieu.

Ce genre de graphique est commun en Analyse Topologique de Données (un domaine des maths relativement nouveau), mais je ne m'y connais pas assez pour savoir s'il y avait des choses à explorer de ce côté.

3.2 Estimation du paramètre dans le cas d'une unique population exponentielle

On fait l'hypothèse dans cette section que chacune des molécule à un temps de vie indépendant, et suit une loi exponentielle de paramètre k_{off}. Alors l'histogramme des durées de vie converge (au facteur multiplicatif près) vers $f(t) = e^{-k_{\text{off}}t}$, ce qui correspond à une droite en échelle semi-log.

On peut alors estimer le paramètre k_{off} par l'inverse de la moyenne des temps de résidences :

$$\hat{k}_{\text{off}} = \frac{N}{\sum_{i=1}^{N} t_r(i)}. \tag{11}$$

Il faut cependant faire attention à la discrétisation des temps, car les temps de résidence sont toujours multiples de la durée d'une frame τ_{frame}, voir même d'une valeur encore plus grande τ (si l'on a regroupée les traces pour limiter le bruit). Si $k_{\text{off}}\tau << 1$, alors la discrétisation a peu d'effet et l'estimateur précédent est valide. Lorsque $k_{\text{off}}\tau \sim 1$, il y a plusieurs corrections possibles

en foncion des hypothèses de mesure, mais dans un certain nombre de cas l'estimateur

$$\hat{k}_{\text{off}}^{\text{discr}} = -\frac{1}{\tau}\log\left(1-\hat{k}_{\text{off}}\tau\right) \tag{12}$$

est plus pertinent. Notons que la formule est toujours bien définie car $\hat{k}_{\text{off}} > \frac{1}{\tau}$ (sauf si toutes les traces sont de 1 frame...). La formule reste valide dans le cas $k_{\text{off}}\tau >> 1$, mais je pense que dans ce cas là il faut questionner l'acquisition des données (car cela signifie que la plupart des traces ne durent que 1 frame).

On réfère au fichier dédié pour plus de détails.

Sur nos données, l'histogramme est proche d'une droite lorsque l'on moyenne sur une grand nombre de frames (20 ou 40). On a alors $\tau \sim 1s$, et $1/k_{\text{off}} \sim 2s$, donc l'effet de la discrétisation n'est pas négligeable (même si en réalité, il n'est pas très grand).

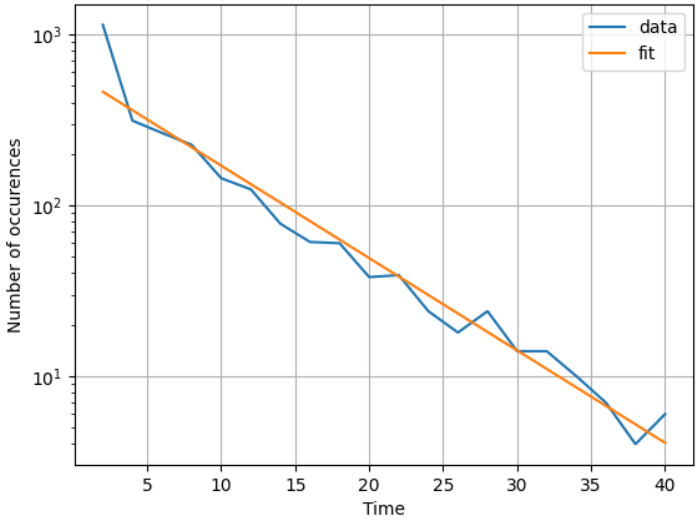

FIGURE 7 – Exemple d'histogramme qui correspond à une distribution exponentielle. L'inverse de la moyenne donne $\hat{k}_{\text{off}} = 0.13$, la correction due à la discrétisation donne $\hat{k}_{\text{off}}^{\text{discr}} = 0.15$, et la regression linéaire donne $\hat{k}_{\text{off}}^{\text{regr}} = 0.12$. C'est donc un mauvais exemple pour illustrer mon propos...

3.3 Mélange de populations : deux populations

On fait maintenant l'hypothèse qu'il y a deux populations de molécules, qui suivent des lois exponentielles de paramètres $k_{\text{off}}^1 > k_{\text{off}}^2$. On suppose qu'il y a une proportion p_i de population de la population avec un taux de mort k_{off}^i pour $i = 1, 2$, et donc que $p_1 + p_2 = 1$. Alors l'histogramme des durées de vie converge (au facteur multiplicatif près) vers

$$g(t) = p_1 k_{\text{off}}^1 e^{-k_{\text{off}}^1 t} + p_2 k_{\text{off}}^2 e^{-k_{\text{off}}^2 t}. \tag{13}$$

On a ici choisi la normalisation pour g qui satisfait $\int_0^\infty g(t)dt = 1$.

La fonction $\log(g)$ admet des asymptotes en 0 et en $+\infty$, d'équations

$$\log(g_0(t)) = \log\left(p_1 k_{\text{off}}^1 + p_2 k_{\text{off}}^2\right) - \frac{p_1 (k_{\text{off}}^1)^2 + p_2 (k_{\text{off}}^2)^2}{p_1 k_{\text{off}}^1 + p_2 k_{\text{off}}^2} t \tag{14}$$

$$\log(g_\infty(t)) = \log\left(p_2 k_{\text{off}}^2\right) - k_{\text{off}}^2 t, \tag{15}$$

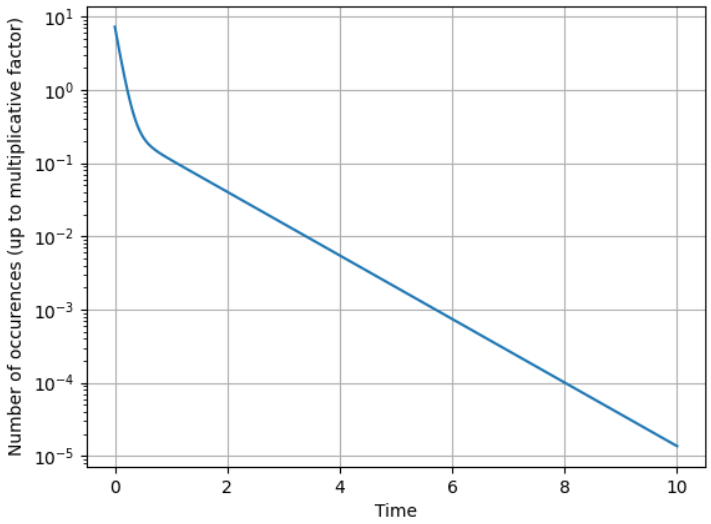

FIGURE 8 – Densité de la loi d'un mélange de deux populations exponentielles, pour laquelle une proportion $p_1 = 0.7$ a un taux $k^1_{\text{off}} = 10$, et une proportion $p_2 = 0.3$ a un taux $k^2_{\text{off}} = 1$.

Il est donc possible de retrouver à la main les paramètres $p_1, p_2, k^1_{\text{off}}, k^2_{\text{off}}$ (On a quatre équations et quatre paramètres. La condition $p_1 + p_2 = 1$ rajoute une équation mais le fait que l'on ne connait g qu'à une constante multiplicative près rajoute également une inconnue, donc en réalité on est à cinq équations et cinq inconnues). Je le ferai pas de cette manière, puisque je trouve que la tangente en zéro est trop peu fiable car elle contient trop peu de points sur nos données (voir la Figure 9).

On va plutôt utiliser un algorithme de minimisation pour trouver la fonction de la forme (13) qui colle le mieux à l'entièreté de la courbe. Cela est fait dans `Tracking_functions.py`, dans la fonction `print_plot_two_exp_fit`. Ici, on s'est débarrassé du facteur multiplicatif en normalisant la courbe expérimentale f de manière à ce que son intégrale soit 1. Pour être précis, puisque f n'est donnée qu'en des valeurs discrètes $f(\tau), f(2\tau), \ldots, f(N\tau)$, son intégrale ici est donnée par

$$\int f = \tau \sum_{i=1}^{N} f(i\tau). \tag{16}$$

(notamment, il ne faut pas oublier de multiplier par τ (je parle d'expérience)). On fera en sorte de prendre N suffisamment grand pour que les hypothétiques valeurs $f(i\tau)$ avec $i > N$ ne contribuent que très peu à la somme. En général, ce n'est pas un problème car f décroit exponentiellement vite.

On note qu'il est important de fitter le log de f avec le log d'une fonction de la forme (13) (et pas la même chose sans les log), car f prend des valeurs sur des échelles très différentes.

On peut également essayer de retrouver à la main deux populations sur le graphe de tracking, de la manière suivante : On choisit un intervalle de la courbe sur lequel la courbe $\log(f)$ est proche d'une droite. On effectue une régression linéaire sur cette portion de droite, et l'on obtient de cette manière k^2_{off}. Ensuite, on soustrait notre droite obtenue à la courbe (attention, comme on est en échelle log, cela ne revient pas seulement à translater la courbe). On espère alors tomber sur un graphe, qui, aux temps inférieurs à la droite que l'on a fittée, se rapproche d'une droite. Si c'est

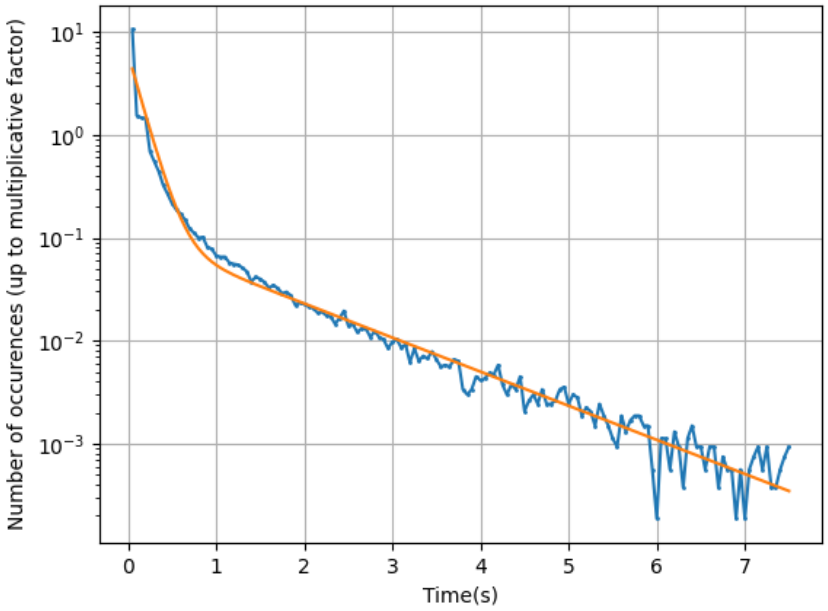

FIGURE 9 – Fit d'un histogramme de longueurs de traces avec un mélange de deux exponentielles avec `print_plot_two_exp_fit`. Ici on obtient $k^1_{\text{off}} = 7.06$, $k^2_{\text{off}} = 0.76$, $p_2 = 0.14$. Le fit me semble satisfaisant, même à la transition entre les deux droites

le cas, on peut faire une deuxième régression linéaire sur cette partie de la courbe, et obtenir k^1_{off}. Le processus est repésenté en Figure 10. Il faut ensuite vérifier que l'intégrale de la courbe fittée correspond environ à l'intégrale de la courbe expérimentale.

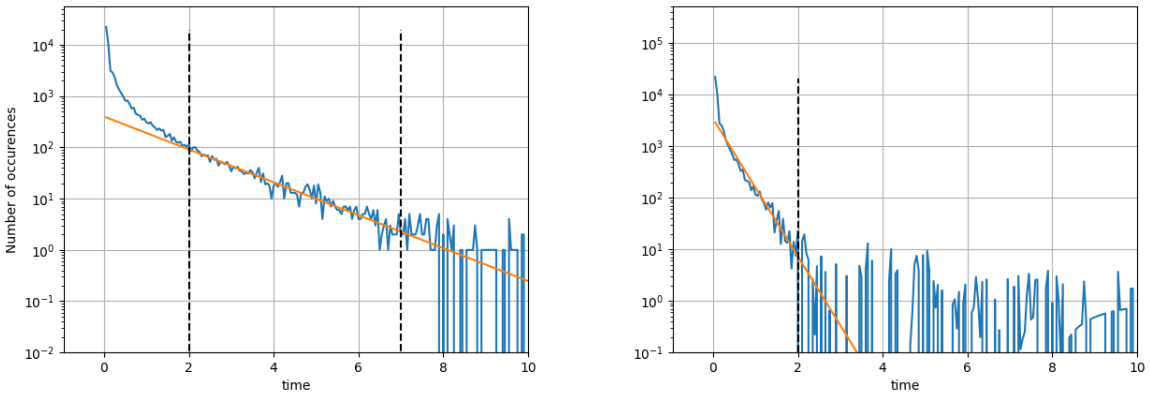

FIGURE 10 – Détection en deux étapes des deux populations. A gauche : historgamme original et regréssion linéaire sur l'intervalle $[2,7]$. A droite : Histogramme dont on a soustrait la première régréssion linéaire. On obtient plus ou moins une droite sur l'intervalle $[0,2]$, et on peut faire une régréssion linéaire dessus.

Il faut faire attention au sens des proportions p_1 et p_2. On réfère à la remarque à ce propos dans la section suivante (3.4.1).

3.4 Mélange de populations : méthode GRID

On peut se demander s'il n'y aurait pas encore plus que deux populations exponentielles dans les données. On détaille dans cette section l'algorithme du papier [2] (code Matlab sur Github : `https://gitlab.com/GebhardtLab/GRID`), qui explique comment décomposer un mélange d'un nombre quelconque de lois exponentielles. Mathématiquement, cela revient à faire une transformée de Laplace inverse. L'algorithme prend en entrée un fonction (qui correspondra à notre histogramme, donc je me permet de l'appeler f) de la forme

$$f(t) = \int_a^b S(k)e^{-kt}dk, \tag{17}$$

où $0 < a < b$ sont des bornes fixées, et infère au mieux possible la fonction S. Bien sûr, tout est discrétisé, donc on considère plutôt une "grille" choisie de valeurs possibles de k : $k \in \{k_1, k_2, \dots, k_p\}$. On prendra typiquement les k_i sur plusieurs ordres de grandeurs, par exemple régulièrement espacés en échelle log entre 10^{-3} et 10^3. L'hypothèse est alors que f s'écrit plutôt sous la forme

$$f(t) = \sum_{i=1}^{p} S_i e^{-k_i t}, \tag{18}$$

De plus, l'algorithme prend en réalité seulement des valeurs $f(\tau), f(2\tau), \dots, f(N\tau)$.

Quitte à multiplier par une constante, on suppose dans la suite que $f(0) = 1$.

L'algorithme lui-même consiste en une minimisation d'un coût (qu'il n'est pas complètement nécessaire de comprendre). On fixe un paramètre $E \geq 0$ (en général $E = 10^{-2}$ fonctionne pas trop mal dans mes exemples). On cherche alors les valeurs $S_1, \dots, S_p$ qui minimisent

$$L(S_1, \dots, S_p) = \sum_{j=3}^{N} \left(\frac{h(j\tau)}{h(2\tau)} - \frac{f(j\tau)}{f(2\tau)} \right)^2 + \frac{E}{2}\Big(q(2\tau) - q(\tau)\Big)^2, \tag{19}$$

où

$$h(t) = \sum_{i=1}^{p} S_i e^{-k_i t}, \qquad\qquad q(t) = \frac{\sum_{i=1}^{p} k_i S_i e^{-k_i t}}{\sum_{i=1}^{p} S_i e^{-k_i t}}. \tag{20}$$

Dans le fichier `GRID_method.py`, la fonction de coût à minimiser est implémentée par `lsqobj`, et le résultat de la minimisation est donnée par la fonction `spectrumfitwrapper`. Avec des données sous la forme $t = (\tau, 2\tau, \dots, N\tau)$, curve $= (f(\tau), f(2\tau), \dots, f(N\tau))$, $k = (k_1, k_2, \dots, k_p)$, $E \in \mathbb{R}_+$, on retrouve la décomposition de la courbe comme en (18) en appliquant `plot_inv_Lapl_transf(t, curve, k, E)`. Des exemples sont fournis à la fin du fichier `GRID_method.py`. On se réfère à B.4 pour le code (si jamais le fichier `.py` est perdu).

Notons que la précision du résultat n'est pas toujours excellente. L'algorithme arrive bien à retrouver S lorsque seulement peu d'entre eux sont non-nuls. Mais lorsque beacoup d'entre eux sont du même ordre de grandeur (et proches), alors l'algorithme a plus de mal. De plus, j'ai l'impression que l'on peut faire confiance à l'algorithme pour savoir s'il y a deux (ou trois) populations, mais la valeur du k pour chacune des populations n'est pas très robuste.

Autre remarque : Il n'est pas nécessaire de renormaliser f (de sorte à avoir $f(0) = 1$ par exemple) pour appliquer l'algorithme, étant donné que l'on n'utilise f qu'à travers des rapports $\frac{f(j\tau)}{f(2\tau)}$. Cepedant, la minimisation se fait sous la contrainte $\sum_{i=1}^{p} S_i = 1$ (ainsi que $S_i \geq 0$), ce qui entraîne que la fonction fittée f_{fit} satisfait toujours $f_{\text{fit}}(0) = 1$.

3.4.1 Application de cet algorithme à des données de mélange de population exponentielles

Supposons que la courbe f soit l'histogramme des temps de résidence d'une certaine molécule. On suppose également que ces molécules sont divisées en $K \geq 2$ populations, et que dans la i-ème population, les molécules ont un temps de vie qui suit une exponentielle de taux $k_i > 0$. On considère qu'il y a une porportion $p_i \in [0,1]$ de molécules dans la population i (et donc $\sum_{i=1}^{K} p_i = 1$). Alors une molécule tirée avec probabilité p_i dans la population i aura une durée de vie t_v qui satisfait

$$\mathbb{P}(t_v \in [t, t+dt]) = \Big(\sum_{i=1}^{K} p_i k_i e^{-k_i t} \Big) dt. \tag{21}$$

Autrement dit, l'histogramme de la longueur des traces va s'approcher de la fonction (à un facteur multiplicatif près)

$$f(t) = \sum_{i=1}^{K} p_i k_i e^{-k_i t}. \tag{22}$$

La méthode GRID nous fournira les poids (à un facteur multiplicatif commun près) $S_i = p_i k_i$, donc pour retrouver les proportions de populations, il ne faut pas oublier de diviser la sortie de l'algorithme par k_i. Pour retrouver le facteur multiplicatif, on utilise que $\sum_{i=1}^{K} p_i = 1$, et donc

$$p_i = \frac{S_i/k_i}{\sum_{j=1}^{K} S_j/k_j}. \tag{23}$$

Le calcul des p_i est implémenté dans `plot_decomp_expo`, voir B.4.

Attention à l'interprétation des proportions p_i ! Ici, on considère l'ensemble des traces sur l'ensemble des temps. Si l'on fixe une frame, et que l'on regarde l'ensemble des molécules présentes sur cette frame, alors la proportion de la population i sera alors proportionnelle à $p_i/k_i \propto S_i/k_i^2$ (sous l'hypothèse que les molécules apparaissent et disparaissent uniformément en temps).

3.5 Nombre de molécules nées/mortes à chaque frame

Il existe encore une manière de vérifier l'adéquation du modèle aux données, et d'estimer $k_{\text{off}} + k_{\text{ph}}$, que je n'ai pas beaucoup explorée, mais que je détaille ici.

On regarde le nombre de traces nées par frames, ainsi que le nombre de traces mortes chaque frame. Selon le modèle à deux boîtes, le nombre de traces mortes chaque frame devrait être proportionnelle au nombre de molécules présentes. C'est vaguement le cas, mais il y a quand même une tendance décroissante de ce rapport, voir la Figure **??**.

De même, sous l'hypothèse que le nombre de molécules dans le cytoplasme varie peu, le nombre de traces nées par frame devrait être constant.

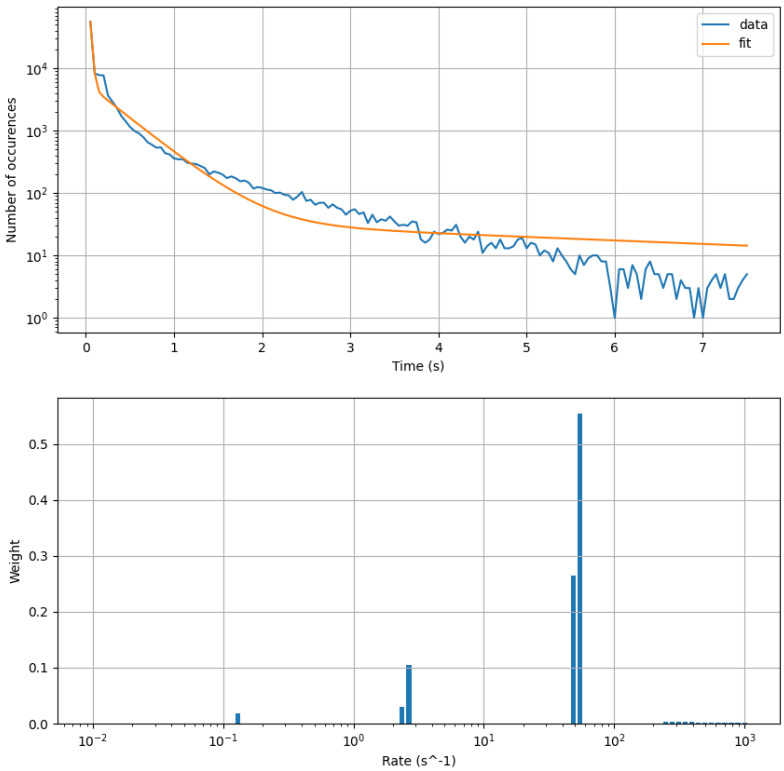

FIGURE 11 – Application de l'algorithme GRID à un histogramme. On a utilisé une grille de 100 valeurs de k entre 10^{-2} et 10^3, régulièrement espacé logarithmiquement, ainsi que $E = 10^{-3}$. Le fit semble moins bon qu'avec seulement deux exponentielles, et le modèle contient plus de paramètres, donc son score BIC (ou chose du genre ratio bon fit sur généralité du modèle) est beaucoup moins bon.

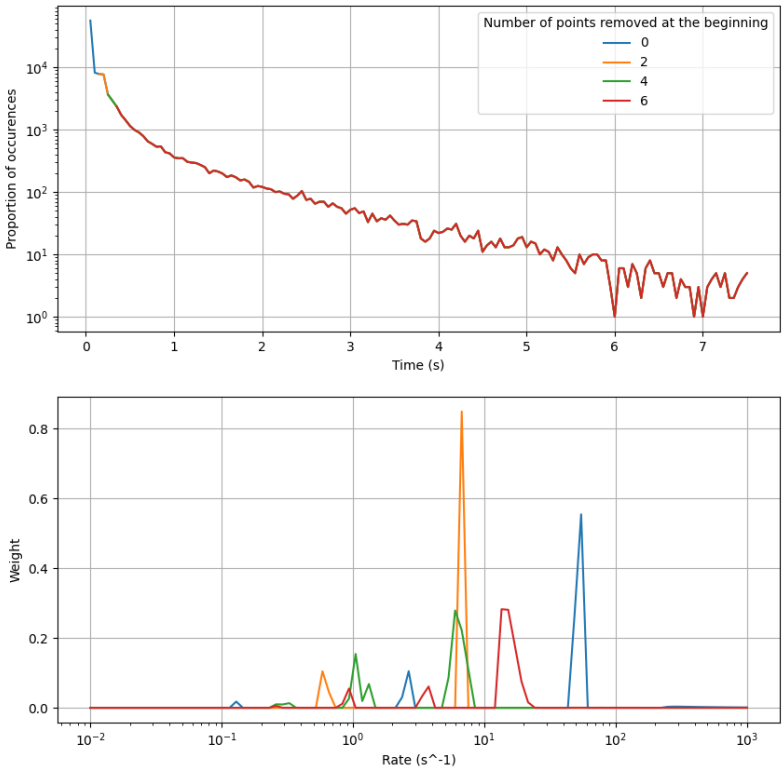

FIGURE 12 – Application de l'algorithme GRID à plusieurs histogrammes quasi identiques (on a seulement retiré quelques points au début de la courbe). La décompositon en population exponentielles n'est pas robuste.

4 Application et limites du modèle sur nos données

4.1 Comparaison des estimations des paramètres par les deux méthodes sur des données expérimentales

Dans cette section, on compare les deux manières de mesurer l'adéquation des données au modèles, notamment en estimant les paramètres k_{on}, k_{off} et k_{ph} à partir du comptage de molécules, et à partir du Tracking. Pour smPRESS, on estime les paramètres k_{off}, k_{ph} et k_{on} à partir de `get_fit_3_params` (voir Algo 2). Pour le tracking, selon le modèle 2.1, la distribution des temps de vie des particules suit une loi exponentielle de paramètre $k_{\text{off}} + k_{\text{ph}}$. On peut alors estimer cette valeur par l'inverse de la moyenne des longueurs des traces, c'est à dire comme en (11)

$$\hat{k}^{\text{track}}_{\text{off+ph}} = \frac{N}{\sum_{i=1}^{N} t_r(i)}. \tag{24}$$

On peut aussi utiliser la correction donnée par la discrétisation comme en (12)

$$\hat{k}^{\text{discr}}_{\text{off+ph}} = -\frac{1}{\tau_{\text{frame}}} \log\Big(1 - \hat{k}^{\text{track}}_{\text{off+ph}} \tau_{\text{frame}}\Big) \tag{25}$$

Cependant, ces deux formules ont du sens lorsque la population suit une unique loi exponentielle. Cela ne semble pas être le cas sur nos données, donc on peut avoir recours à d'autres manières d'estimer $k_{\text{off}} + k_{\text{ph}}$. Si l'histogramme de la longueur des traces correpond bien à une somme de deux populations exponentielles, on a accès à deux valeurs de turnover, une pour chacune des deux populations. La valeur la plus petite de ces deux turnover est plus facilement mesurable : c'est la pente que l'on observe aux temps longs (souvent, >2s suffit). Cette valeur est donc relativement robustement mesurable. On la note $\hat{k}^{\text{slope}}_{\text{off+ph}}$.

Comparons finalement tous ces estimateurs sur un même graphe. On dispose de plusieurs échantillons, et on fait également varier la manière dont le film a été modifié avant que l'algorithme de tracking n'ait été implémenté : on peut faire de moyennes sur plusieurs frames pour réduire le bruit, et ces moyennes peuvent être glissantes ("sa" pour "sliding average"), ou non. Noter que le rapport signal-sur-bruit change losrqu'on modifie le moyennage. On obtient les Figures 13, 14 et 15.

4.2 Les variations stochastiques sont elles trop grandes pour que le modèle colle aux données ?

On peut alors comparer les simulations stochastiques aux données réelles. Il apparaît que les variations des simulations semblent parfois plus grandes que les variations des données. Ce n'est pas normal, parce que les variations des données comportent à la fois les variations stochastiques intrinsèques à la dynamique, plus un bruit de mesure. On voit bien ce bruit de mesure, puisque que les variations des données sont à plus hautes fréquence que les variations des simulations.

A faire : réussir à quantifier cette différence de variations.

4.3 Y a-t-il réellement deux populations ?

Le modèle prédit que les temps de vie des molécules observées suit une loi exponentielle de paramètre $k_{\text{off}} + k_{\text{ph}}$. Cela signifierait que l'histogramme de la longueur des traces soit une droite

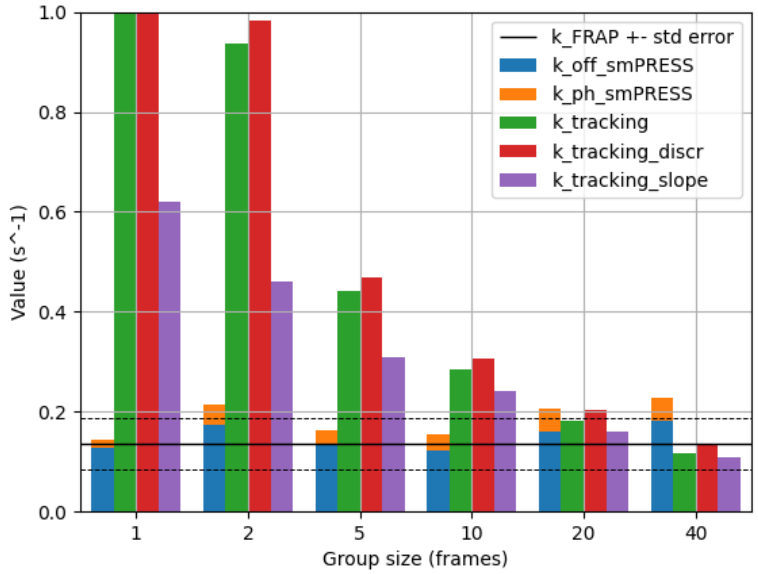

FIGURE 13 – Comparaison des différentes estimations de $k_{\text{off}} + k_{\text{ph}}$. L'estimation FRAP a été effectuée avec 11 échantillons, mais les autres estimations ne proviennent que d'un seul échantillon (Utrophin::GFP, fbr92).

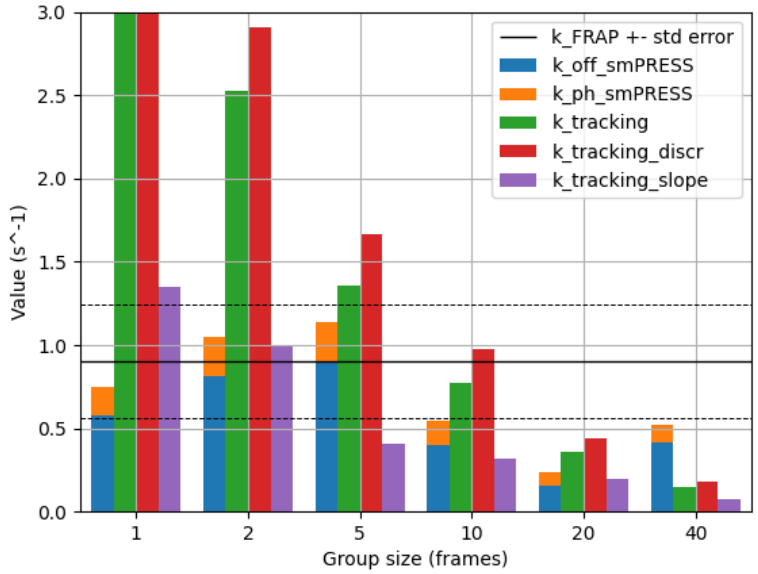

FIGURE 14 – Comparaison des différentes estimations de $k_{\text{off}} + k_{\text{ph}}$. L'estimation FRAP a été effectuée avec 15 échantillons, mais les autres estimations ne proviennent que d'un seul échantillon (LifeAct::GFP, fbr181). Lorsque le temps d'une frame effective dépasse le temps de turover (ici, 20 frames correspond à 1s), les estimateurs ne collent plus à la valeur détectée par le FRAP. Je me serai attendu à ce que la correction de la discrétisation soit utile ici, mais en fait pas vraiment... Le smPRESS reste encore la méthode la plus robuste.

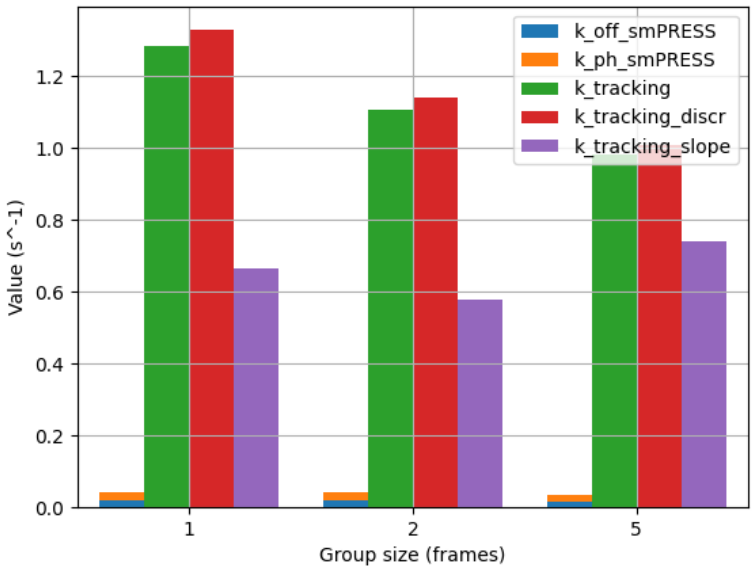

FIGURE 15 – Comparaison des différentes estimations de $k_{\text{off}} + k_{\text{ph}}$ sur les données de Marjolaine. Les fluorophores qu'elle utilise (HaloTag si je ne trompe pas) se lient à la formine, qui a des trajectoires ballistiques, contrairement aux monomères d'actine qui sont à peu près immobiles.

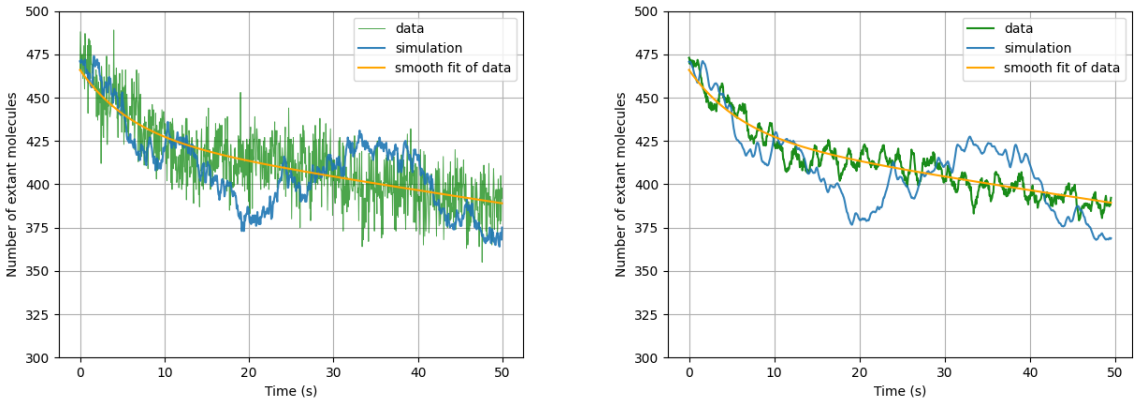

FIGURE 16 – La simulation a été effectuée avec les paramètres retournés par le fit des donnée. A gauche : données brutes. A droite : Les données, ainsi que la simulation, ont été moyennée sur 10 frames (glissantes), pour diminuer le bruit haute fréquence. Sur ce deuxième graphe les grandes variations de la simulation stochastique sont plus claires. Ce graphe a été réalisé avec une seule réalisation stochastique, mais le constant reste le même sur la plupart des simulations.

(en échelle log en ordonnée). Ce n'est pas ce que l'on observe : la plupart du temps, il y a une sur-représentation des traces courtes (voir par exemple la Figure 17). Comment expliquer cela ? Y aurait-il une séparation des molécules en deux (ou plus) populations, ou est-ce que cette anomalie provient d'un bruit ? Dans le premier cas, à quoi correpondent ces populations ? Dans le deuxième cas, quel est la source de ce bruit (mesure, algorithme de tracking, ...) ? Nous n'avons pas de réponse claire à ces questions.

La qualité du tracking diffère en fonction de la densité totale des molécules. Elle est par exemple

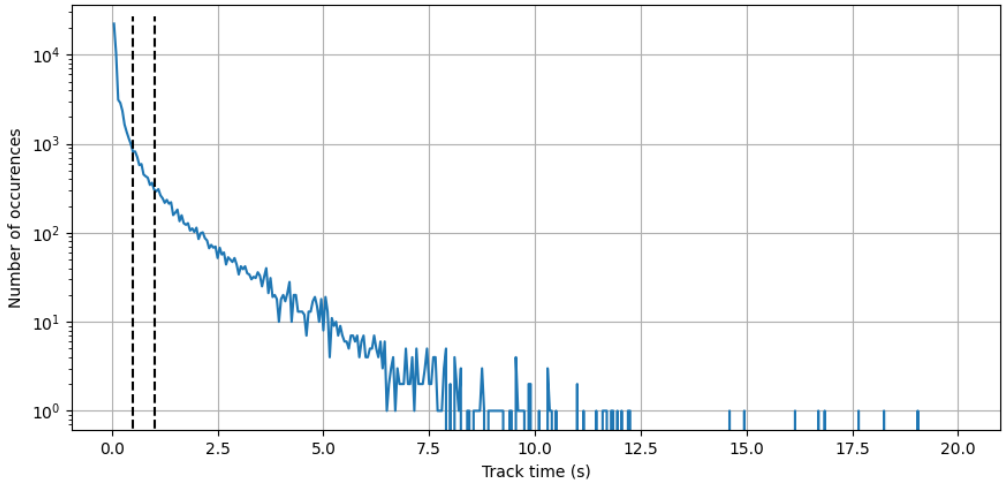

FIGURE 17 – Séparation en deux population. On crée une population avec les molécules dont les traces sont plus longues que 2s (2ème ligne verticale), et une autre population avec les molécules dont les traces sont plus courtes que 1s. On ne s'intéresse pas aux molécules de traces intermédiaires, qui pourraient être un mélange des deux populations.

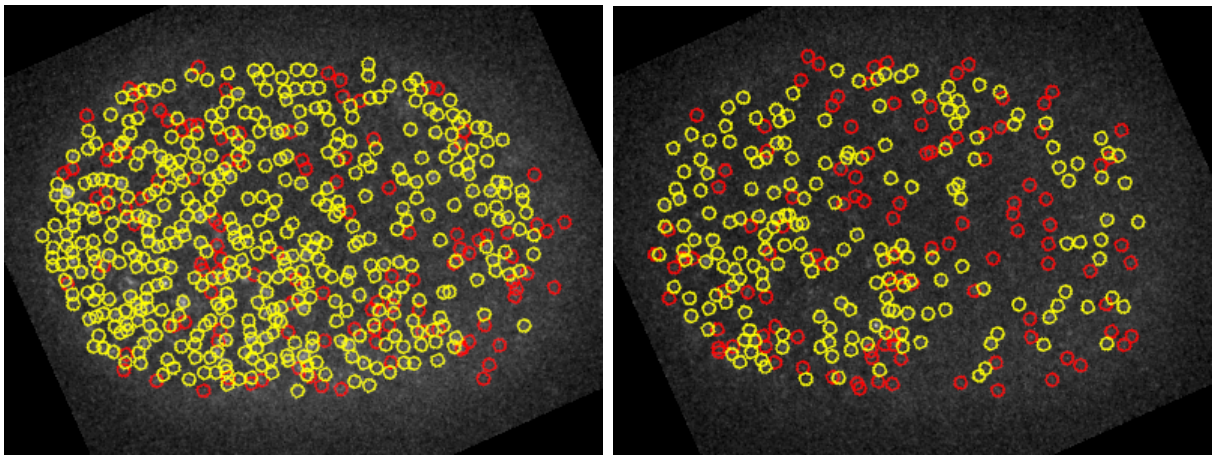

FIGURE 18 – Tentative de séparation en deux populations. En rouge, les molécules de traces courtes (<1s), en jaune, les molécules de traces longues (>2s). Les deux images correspondent à deux instants dans le même film, dans lesquels la densité de molécules présente est différente. On ne distingue pas de différence de localisation spatiale entre ces deux populations. PS : Je ne suis pas sûr de savoir différencier le postérieur et l'antérieur sur le sens de la figure.

meilleure sur l'image droite de la Figure 18 que sur l'image gauche. On peut se demander si le surplus de traces courtes provient d'un tracking fait lorsque la population de molécules détectées est trop dense. Pour cela, on compare sur un même film l'histogramme des traces des molécules nées au début du film (grande densité), avec l'histogramme des molécules mortes à la fin du film (petite densité). On n'observe pas de réduction de traces courtes.

Note : il est important de prendre les traces des molécules *mortes* dans un intervalle proche de la fin plutôt que nées dans un intervalle proche de la fin, pour éviter un biais dû au fait que les traces ne peuvent pas dépasser la fin du film. On pourra penser à la Figure 6.

On repère tout de même une différence entre les traces courtes et les traces longues : L'intensité

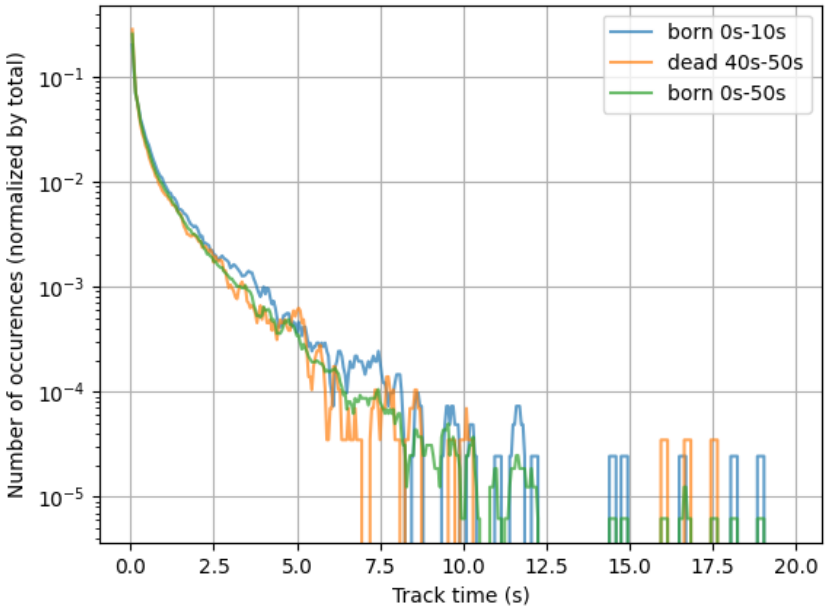

FIGURE 19 – Histogramme de la longueur des traces dans trois populations : Les traces nées vers le début du film (tracking dense, supposément plus mauvais), celle mortes vers la fin du film (tracking épars, supposément meilleur), et la population totale. Il n'apparaît pas de différence significative entre les trois courbes.

des spots des traces longues est en moyenne plus grande que celle des traces courtes.

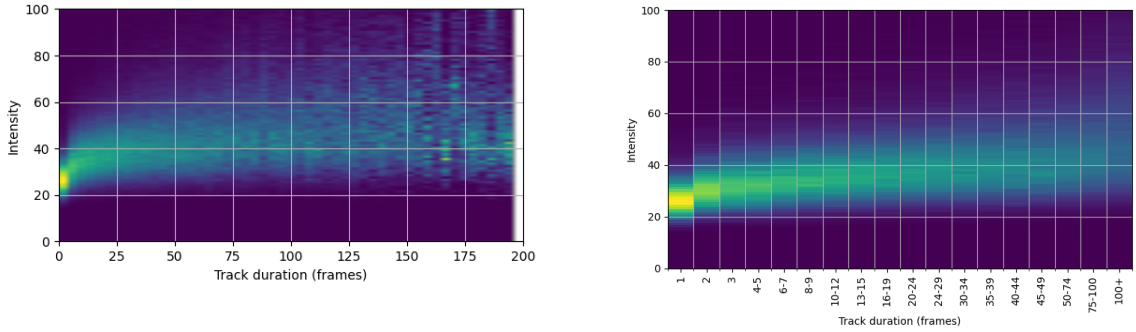

FIGURE 20 – Intensité des traces en fonction de leur temps de résidence. Chaque colonne, correspondant aux traces entre deux temps donnés, a été renormalisée de manière à avoir un poids total de 1. Il apparaît que les traces courtes ont une intensité plus faible que les traces longues. Les deux figures représentent la même chose, seulement dans la deuxième l'axe des abscisses n'est pas linéaire, pour montrer plus de détails autour des traces courtes.

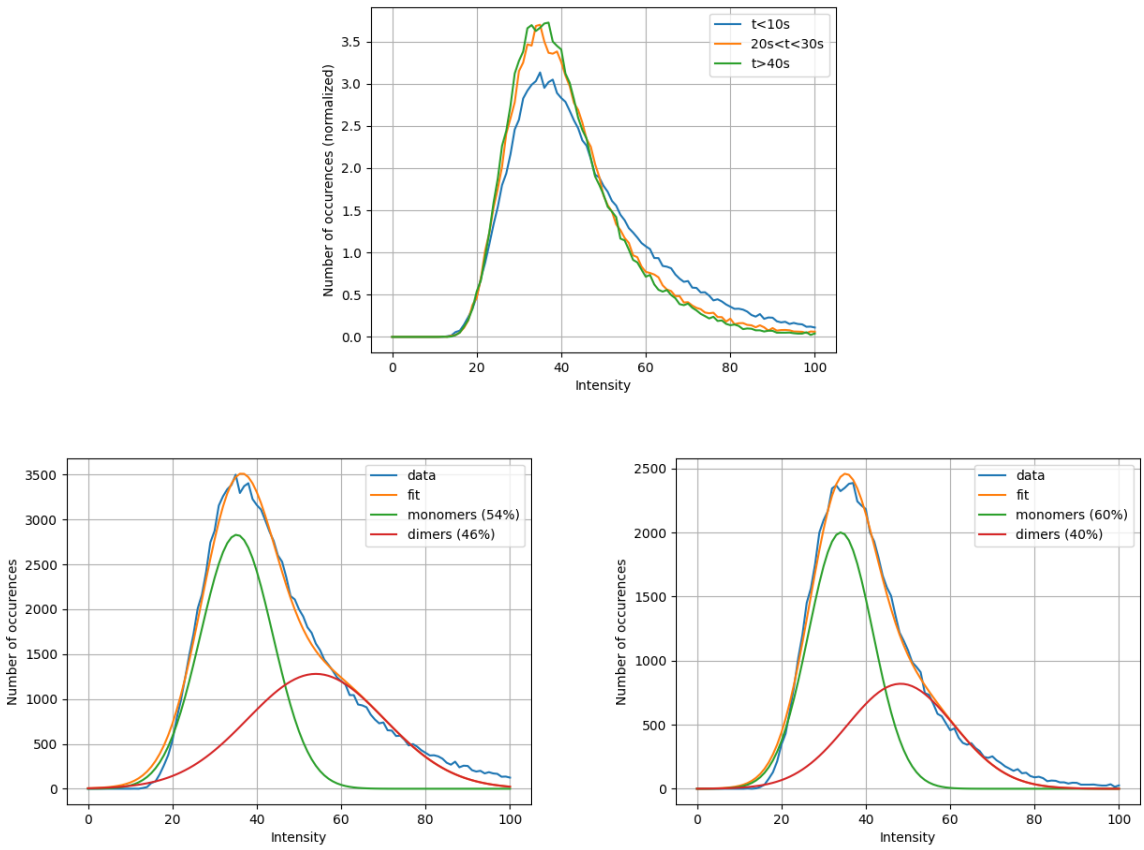

FIGURE 21 – Distribution de l'intensité des spots. En haut, on compare les distributions faites au début, au milieu et à la fin du film, pendant lequel la densité de détection de molécule diminue. En bas, on fit cette distribution avec un mélange de deux exponentielles, correspondant probablement aux monomères et aux dimères. On trouve une grande proportions de dimères (>40%), que ca soit au début (à gauche) ou à la fin (à droite) du film, ce qui suggère que le tracking est mauvais dans tous les cas ? ? ? Alors qu'à faible densité (i.e. à la fin du film), la densité à l'air correcte pour un bon tracking. Sinon, cela peut être un indice de plus vers la présence de deux populations ? On peut aussi imposer que la variance de la courbe des dimères soit deux fois plus grande que celle dela courbe des monomères. Cela ne change pas grand chose (car c'est déjà presque le cas que le rapport des variances vaut 2).

4.4 Une solution possible : Le modèle à trois boîtes

4.4.1 Boites supérieures en série

Dans ce modèle, on rajoute une couche intermédiaire entre le cortex et le cytoplasme, que l'on appelera ici couche intermédiaire. Cette couche représente des molécules que l'on observe, et qui sont donc photobleachées. On peut imaginer que cette couche est la surface inférieure du cortex, ou bien la surface supérieure du cyotplasme. Ce n'est pas très important ici, car on ne fait qu'une analyse mathématique du modèle. On donne en Figure 22 un schématisation du modèle.

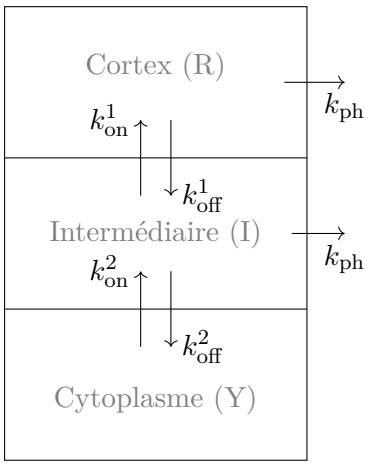

FIGURE 22 – Schématisation du modèle à trois boîtes

Les densités de molecules dans chacune des boîtes satisfait alors

$$R'(t) = -(k_{\text{ph}} + k^1_{\text{off}})R(t) + k^1_{\text{on}} I(t), \tag{26}$$

$$I'(t) = k^1_{\text{off}} R(t) - (k_{\text{ph}} + k^1_{\text{on}} + k^2_{\text{off}})I(t) + k^2_{\text{on}} Y(t), \tag{27}$$

$$Y'(t) = k^2_{\text{off}} I(t) - k^2_{\text{on}} Y(t). \tag{28}$$

En supposant qu'au temps 0 le système est à l'équilibre pour $k_{\text{ph}} = 0$ (c'est à dire $-k^1_{\text{off}}R(0) + k^1_{\text{on}}I(0) = k^1_{\text{off}}R(0) - (k^1_{\text{on}} + k^2_{\text{off}})I(0) + k^2_{\text{on}}Y(0) = k^2_{\text{off}}I(0) - k^2_{\text{on}}Y(0) = 0$), on peut trouver une expression plus ou moins explicite de la solution :

$$R(t) = R(0)\left[\frac{(k_{\text{ph}}+r_2)(k_{\text{ph}}+r_3)}{(r_1-r_2)(r_1-r_3)}e^{r_1 t} + \frac{(k_{\text{ph}}+r_1)(k_{\text{ph}}+r_3)}{(r_2-r_1)(r_2-r_3)}e^{r_2 t} + \frac{(k_{\text{ph}}+r_1)(k_{\text{ph}}+r_2)}{(r_3-r_1)(r_3-r_2)}e^{r_3 t}\right], \tag{29}$$

$$\begin{aligned} I(t) = I(0)\Bigg[&\left(1 + \frac{k_{\text{ph}}+r_1}{k^1_{\text{off}}}\right)\frac{(k_{\text{ph}}+r_2)(k_{\text{ph}}+r_3)}{(r_1-r_2)(r_1-r_3)}e^{r_1 t} + \left(1 + \frac{k_{\text{ph}}+r_2}{k^1_{\text{off}}}\right)\frac{(k_{\text{ph}}+r_1)(k_{\text{ph}}+r_3)}{(r_2-r_1)(r_2-r_3)}e^{r_2 t} \\ &+ \left(1 + \frac{k_{\text{ph}}+r_3}{k^1_{\text{off}}}\right)\frac{(k_{\text{ph}}+r_1)(k_{\text{ph}}+r_2)}{(r_3-r_1)(r_3-r_2)}e^{r_3 t}\Bigg] \end{aligned} \tag{30}$$

où

$$r_1 r_2 r_3 = -k^2_{\text{on}} k_{\text{ph}}(k_{\text{ph}} + k^1_{\text{off}} + k^1_{\text{on}}), \tag{31}$$

$$r_1 r_2 + r_1 r_3 + r_2 r_3 = k^1_{\text{off}}(k_{\text{ph}} + k^2_{\text{off}}) + k_{\text{ph}}(k^1_{\text{on}} + k_{\text{ph}} + k^2_{\text{off}}) + k^2_{\text{on}}(k^1_{\text{off}} + k^1_{\text{on}} + 2k_{\text{ph}}), \tag{32}$$

$$r_1 + r_2 + r_3 = -(k^1_{\text{off}} + k^1_{\text{on}} + 2k_{\text{ph}} + k^2_{\text{off}} + k^2_{\text{on}}). \tag{33}$$

(Dans l'hypothèse peu probable où ces formules seront utilisées, merci de ne pas faire confiance aveuglément à l'absence d'erreur.) (Les $\cdot^2$ correspondent toujours à un indice, jamais à un carré.)

La quantité observée serait $R(t) + I(t)$, ou éventuellement juste $R(t)$, dans le cas où la couche intermédiaire appartient au cytoplasme et où les molécules sont trop rapides pour être observées.

Noter que l'on pourrait retrouver $k_{\text{ph}}, k^1_{\text{off}}, k^1_{\text{on}}, k^2_{\text{off}}, k^2_{\text{on}}$ à partir de $r_1, r_2, r_3, k_{\text{ph}}$ et k^1_{off}. par contre, si l'on ne mesure que $R(t)$, alors on a seulement accès à $r_1, r_2, r_3, k_{\text{ph}}$, et il manque un paramètre pour retrouver tous les paramètres originaux. On remarque également que l'on a encore, comme dans le modèle à deux boîtes,

$$R'(0) = -k_{\text{ph}} R(0). \tag{34}$$

Ce modèle permettrait d'expliquer

— la présence de deux populations (Figure 17),
— la non-gaussianité de l'intensité des spots même à faible densité (Figure 21) (car les molecules de R et I peuvent apparaître avec des intensités différentes),
— (potentiellement) la faible variation intrinsèque du nombre de particule en fonction du temps (Figure 16).

Malheuresement, le modèle est trop complexe pour faire des estimation robustes de ses paramètres.

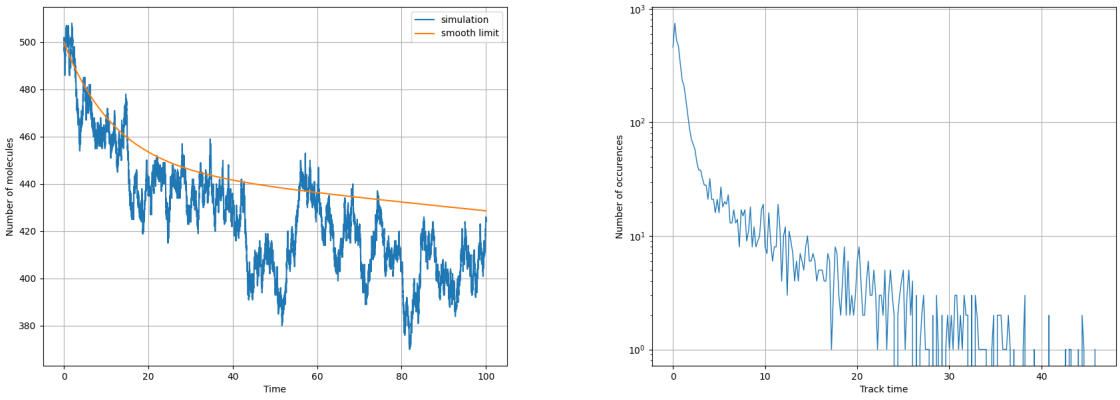

FIGURE 23 – A gauche : Nombre de molecules dans R+I en fonction du temps sur une simulation du modèle à trois boîtes. Comparer avec la Figure 16. Les variations stochastiques intrinsèques ont l'air moins grandes que celles du modèle à deux boîtes, et donc plus cohérent avec les données. A droite : Histogramme de la longueur des traces pour une simulation du modèle à trois boîtes. Une trace commence quand une molécule entre dans R+I, et se termine lorsque cette molécule revient dans Y. On retrouve (plus ou moins) la forme de nos données, qui ressemble à un mélange de deux exponentielles. Les deux graphes proviennent de la même simulation, qui a été effectuée avec $k^1_{\text{off}} = 0.1$, $k^2_{\text{off}} = 1$, $k_{\text{ph}} = 0.01$, $k^1_{\text{on}} = 0.4$, $k^2_{\text{on}} = 0.01$, et $R(0) + I(0) + Y(0) = 10^4$.

La forme théorique de la courbe de la Figure à droite 23 (i.e. la loi du temps de vie d'une molécule) est probablement calculable explicitement. Mais ce n'est pas un mélange parfait de deux exponentielles.

A faire : Calculer à quoi ressemblerait le FRAP. i.e. la solution si on impose $R(0) = I(0) = 0$ (cas idéal), ou $R(0) = \frac{k_{\text{on}}}{k_{\text{off}}} I(0) \ll R_{eq}$ la valeur à l'équilibre sans photobleaching.

4.4.2 Boites supérieures en parallèle

On pourrait aussi penser à un modèle à trois boîtes de la forme de la Figure 24. Ce modèle me semble étrange d'un point de vue modélisation, car je ne vois pas de raison pour laquelle les molécules ne pourraient pas passez du Cortex 1 au Cortex 2. Ou alors il faut ajouter un échange entre le cortex 1 et le cortex 2, mais ca devient encore plus complexe.

Ce modèle est plus contraignant à justifier biologiquement : Il faut qu'il y ait vraiement deux molécules différentes. L'avantage de ce modèle est que la loi de la durée d'une trace est exactement un mélange de deux exponentielles, dont on connait les taux (ce sont k^1_{off} et k^2_{off}).

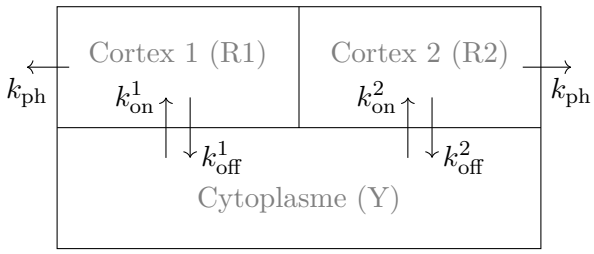

FIGURE 24 – Schématisation du modèle à trois boîtes, version parallèle

Les densités de molecules dans chacune des boîtes satisfait alors

$$R_1'(t) = -(k_{\text{ph}} + k^1_{\text{off}})R_1(t) + k^1_{\text{on}}Y(t), \tag{35}$$
$$R_2'(t) = -(k_{\text{ph}} + k^2_{\text{off}})R_2(t) + k^2_{\text{on}}Y(t), \tag{36}$$
$$Y'(t) = k^1_{\text{off}}R_1(t) + k^2_{\text{off}}R_2(t) - (k^1_{\text{on}} + k^2_{\text{on}})Y(t). \tag{37}$$

Je n'ai pas calculé la solution explicite, mais ca doit être faisable. Idem pour la courbe théorique du FRAP dans ce cas.

A Figures bonus

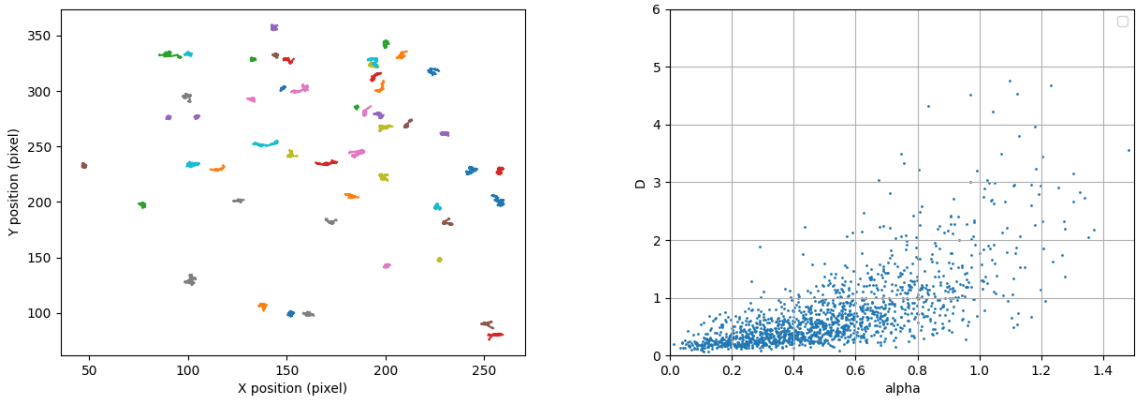

FIGURE 25 – A gauche : 50 trajectoires de longueur au moins 100 frames. Le caractère sous-diffusif apparaît : Les trajectoires sont presque immobiles. A droite : MSD de toutes les trajectoires de longueur au moins 50 frames.

B Bouts de code

B.1 Estimation des l'erreur sur l'estimation des paramètres de smPRESS

Algo 6 – get_up_down_estimate_k

```
def get_std_dev(N, N_fit):
    N_diff = np.array(N) - np.array(N_fit)
    return sqrt(np.mean(N_diff*N_diff))

def get_std_dev_k(t, N, k_on, k_off, k_ph):
    N_fit = sin_mol_fun_k(t, N[0], k_on, k_off, k_ph)
    return get_std_dev(N, N_fit)

def get_up_down_estimate_k(t, N):
    k_on, k_off, k_ph, _, _, _ = get_fit_3_params(t, N)
    sigma = get_std_dev_k(t, N, k_on, k_off, k_ph)
    k_on_up, k_off_up, k_ph_up, _, _, _ = get_fit_3_params(t, N + sigma)
    k_on_down, k_off_down, k_ph_down, _, _, _ = get_fit_3_params(t, N -
        sigma)
    print("k_on : ", round_sig(k_on_down, 3), round_sig(k_on, 3), round_sig(
        k_on_up, 3))
    print("k_off : ", round_sig(k_off_down, 3), round_sig(k_off, 3),
        round_sig(k_off_up, 3))
    print("k_ph : ", round_sig(k_ph_down, 3), round_sig(k_ph, 3), round_sig(
        k_ph_up, 3))
    return [k_on_down, k_on, k_on_up], [k_off_down, k_off, k_off_up], [
        k_ph_down, k_ph, k_ph_up]
```

B.2 Algorithme de Gillespie pour smPRESS

Algo 7 – simul_2_phases_Gillepsie, get_counts_simu

```
def update_state_Gillepsie(birth_times, death_times, alive, cyto_pool, t,
    t_max, k_on, k_off, k_ph):
    # Modifies birth_times, death_times, alive and cyto_pool IN PLACE
    # Takes the state of the system at time t, update the state at the time
        of the next reaction, returns this time
    nb_tracks = len(birth_times)
    total_rate = k_on*cyto_pool[0] + (k_off+k_ph) * len(alive)
    dt = np.random.exponential(1/total_rate)
    if t + dt > t_max :
        return t_max
    choose_reaction = np.random.random()
    # Birth
    if choose_reaction < k_on*cyto_pool[0]/total_rate:
        birth_times.append(t+dt)
        death_times.append(t_max)
        alive.append(nb_tracks)
        cyto_pool[0] -= 1
    # Death by going into the cytoplasm
    elif choose_reaction < (k_on*cyto_pool[0] + k_off*len(alive))/total_rate
        :
    particle_id = np.random.choice(np.array(alive))
    death_times[particle_id] = t+dt
```

```
            alive.remove(particle_id)
            cyto_pool[0] += 1
        # Death by photobleaching
        else:
            particle_id = np.random.choice(np.array(alive))
            death_times[particle_id] = t+dt
            alive.remove(particle_id)
        return t + dt

def simul_2_phases_Gillepsie(t_final, cyto_pool_0, k_on, k_off, k_ph):
    birth_times = []
    death_times = []
    alive = []
    t = 0
    reaction_times = [0]
    counts = [0]
    cyto_pool = [cyto_pool_0]
    cyto_counts = [cyto_pool_0]
    percent_done = 0
    print("Phase 1 commencee")
    while t < t_final/2:
    t = update_state_Gillepsie(birth_times, death_times, alive, cyto_pool, t
        , t_final/2, k_on, k_off, 0)
    counts.append(len(alive))
            cyto_counts.append(cyto_pool[0])
            reaction_times.append(t)
            if t/t_final >= percent_done + 0.1 :
            percent_done += 0.1
                print(round(100*percent_done), "% done", sep = "")
                print("Phase 2 commencee")
    while t < t_final:
            t = update_state_Gillepsie(birth_times, death_times, alive,
               cyto_pool, t, t_final, k_on, k_off, k_ph)
            counts.append(len(alive))
            cyto_counts.append(cyto_pool[0])
            reaction_times.append(t)
            if t/t_final >= percent_done + 0.1 :
            percent_done += 0.1
            print(round(100*percent_done), "% done", sep = "")
            return np.array(counts), np.array(birth_times), np.array(death_times
               ), np.array(cyto_counts), np.array(reaction_times)

            def get_counts_simu(t_final, k_on, k_off, k_ph, cyto_pool_0, plot=
               False):
    counts, birth_times, death_times, cyto_counts, reaction_times =
       simul_2_phases_Gillepsie(t_final, cyto_pool_0, k_on, k_off, k_ph)
    birth_times_ph = birth_times[birth_times>t_final/2] - t_final/2
    death_times_ph = death_times[birth_times>t_final/2] - t_final/2
    resitimes = get_resitimes_distrib(birth_times_ph, death_times_ph)
```

```
    reaction_times_ph = reaction_times[reaction_times>=t_final/2] - t_final
        /2
    counts_ph = counts[reaction_times>=t_final/2]
    if plot:
    t_steps = np.linspace(t_final/2, t_final, 100)
    N_part2 = sin_mol_fun_k(t_steps-t_final/2, counts[np.where(
        reaction_times==t_final/2)[0]], k_on, k_off, k_ph)
    fig, ax = plt.subplots(2,1)
    ax[0].vlines(t_final/2, ymin=0, ymax = max(counts), linestyle = 'dashed'
        , lw=1, color='black')
    ax[0].plot(reaction_times, counts)
    ax[0].plot(t_steps, N_part2)
    ax[1].plot(reaction_times, cyto_counts)
    plt.show()
    return counts_ph, birth_times_ph, death_times_ph, cyto_counts,
        reaction_times_ph, resitimes
```

B.3 Algorithme de Gillespie pour smPRESSle modèle à trois boîtes, version série

```
    '''
␣␣␣␣------␣3␣boxes␣:
␣␣␣␣cortex␣␣|␣␣surface␣(cytoplasm)␣␣|␣␣(deep)␣cytoplasm
␣␣␣␣------␣5␣reactions␣:
␣␣␣␣k_on␣␣:␣surf␣->␣cortex
␣␣␣␣k_off␣:␣cortex␣->␣surf
␣␣␣␣k_ph␣␣:␣cortex␣&␣surf␣->␣/
␣␣␣␣k_on_cy␣:␣cyto␣->␣surf
␣␣␣␣k_off_cy␣:␣surf␣->␣cyto
␣␣␣␣---␣In␣our␣setup␣we␣will␣have␣␣k_on_cy,␣k_off_cy␣>>␣k_on,␣k_off
␣␣␣␣'''

    def update_state_Gillepsie_3_boxes(birth_times, death_times, alive_ctx,
        alive_surf, cyto_pool, t, t_max, k_on_cy, k_off_cy, k_on, k_off, k_ph
        ):
    # Modifies birth_times, death_times, alive_surf, alive_ctx and cyto_pool
         IN PLACE
    # Takes the state of the system at time t, update the state at the time
        of the next reaction, returns this time
    nb_tracks = len(birth_times)
    total_rate = k_on_cy*cyto_pool[0] + (k_off_cy + k_on + k_ph) * len(
        alive_surf) + (k_off + k_ph) * len(alive_ctx)
    dt = np.random.exponential(1/total_rate)
    if t + dt > t_max :
    return t_max
    choose_reaction = np.random.random()
    # cyto -> surf
    if choose_reaction < k_on_cy*cyto_pool[0]/total_rate:
```

```
        birth_times.append(t+dt)
        death_times.append(t_max)
        alive_surf.append(nb_tracks)
        cyto_pool[0] -= 1
    # surf -> cyto
    elif choose_reaction < (k_on_cy*cyto_pool[0] + k_off_cy*len(alive_surf))
        /total_rate:
        particle_id = np.random.choice(np.array(alive_surf))
        death_times[particle_id] = t+dt
        alive_surf.remove(particle_id)
        cyto_pool[0] += 1
    # surf -> cortex
    elif choose_reaction < (k_on_cy*cyto_pool[0] + (k_off_cy+k_on)*len(
        alive_surf))/total_rate:
        particle_id = np.random.choice(np.array(alive_surf))
        alive_ctx.append(particle_id)
        alive_surf.remove(particle_id)
    # surf -> photobleach
    elif choose_reaction < (k_on_cy*cyto_pool[0] + (k_off_cy+k_on+k_ph)*len(
        alive_surf))/total_rate:
        particle_id = np.random.choice(np.array(alive_surf))
        death_times[particle_id] = t+dt
        alive_surf.remove(particle_id)
    # cortex -> surf
    elif choose_reaction < (k_on_cy*cyto_pool[0] + (k_off_cy+k_on+k_ph)*len(
        alive_surf) + k_off*len(alive_ctx))/total_rate:
        particle_id = np.random.choice(np.array(alive_ctx))
        alive_surf.append(particle_id)
        alive_ctx.remove(particle_id)
    # cortex -> photobleach
    else:
        particle_id = np.random.choice(np.array(alive_ctx))
        death_times[particle_id] = t+dt
        alive_ctx.remove(particle_id)
    return t + dt

def simul_2_phases_Gillepsie_3_boxes(t_final, cyto_pool_0, k_on_cy,
    k_off_cy, k_on, k_off, k_ph):
    birth_times = []
    death_times = []
    alive_ctx = []
    alive_surf = []
    t = 0
    reaction_times = [0]
    counts_ctx = [0]
    counts_surf = [0]
    counts = [0]
    cyto_pool = [cyto_pool_0]
    cyto_counts = [cyto_pool_0]
    print("Computing␣Phase␣1...")
```

```
percent_done = 0
while t < t_final/2:
t = update_state_Gillepsie_3_boxes(birth_times, death_times, alive_ctx,
    alive_surf, cyto_pool, t, t_final/2, k_on_cy, k_off_cy, k_on, k_off,
    0)
counts_ctx.append(len(alive_ctx))
counts_surf.append(len(alive_surf))
counts.append(len(alive_ctx) + len(alive_surf))
cyto_counts.append(cyto_pool[0])
reaction_times.append(t)
if t/t_final >= percent_done + 0.1 :
percent_done += 0.1
print(round(100*percent_done), "% done", sep = "")
print("Computing Phase 2...")
while t < t_final:
t = update_state_Gillepsie_3_boxes(birth_times, death_times, alive_ctx,
    alive_surf, cyto_pool, t, t_final, k_on_cy, k_off_cy, k_on, k_off,
    k_ph)
counts_ctx.append(len(alive_ctx))
counts_surf.append(len(alive_surf))
counts.append(len(alive_ctx) + len(alive_surf))
cyto_counts.append(cyto_pool[0])
reaction_times.append(t)
if t/t_final >= percent_done + 0.1 :
percent_done += 0.1
print(round(100*percent_done), "% done", sep = "")
return np.array(counts), np.array(counts_ctx), np.array(counts_surf), np
    .array(birth_times), np.array(death_times), np.array(cyto_counts), np
    .array(reaction_times)

def get_resitimes_distrib(birth_times, death_times):
return np.array(death_times) - np.array(birth_times)

def get_counts_simu_3_boxes(t_final, k_on_cy, k_off_cy, k_on, k_off,
    k_ph, cyto_pool_0, plot=False):
counts, counts_ctx, counts_surf, birth_times, death_times, cyto_counts,
    reaction_times = simul_2_phases_Gillepsie_3_boxes(t_final,
    cyto_pool_0, k_on_cy, k_off_cy, k_on, k_off, k_ph)
birth_times_ph = birth_times[birth_times>t_final/2] - t_final/2
death_times_ph = death_times[birth_times>t_final/2] - t_final/2
resitimes = get_resitimes_distrib(birth_times_ph, death_times_ph)
reaction_times_ph = reaction_times[reaction_times>=t_final/2] - t_final
    /2
counts_ph = counts[reaction_times>=t_final/2]
counts_ctx_ph = counts_ctx[reaction_times>=t_final/2]
counts_surf_ph = counts_surf[reaction_times>=t_final/2]
if plot:
plt.plot(reaction_times_ph, counts_ph, label = 'cortex + surface')
plt.plot(reaction_times_ph, counts_ctx_ph, label = 'cortex')
plt.plot(reaction_times_ph, counts_surf_ph, label='surface')
```

```
    plt.xlabel("Time")
    plt.ylabel("Number of molecules")
    plt.legend()
    plt.show()
    return counts_ph, counts_ctx_ph, counts_surf_ph, birth_times_ph,
        death_times_ph, cyto_counts, reaction_times_ph, resitimes

# Theoretical curve
def sin_mol_fun_3_boxes(tdata, N0, k_on_cy, k_off_cy, k_on, k_off, k_ph):
    R0 = k_on / (k_on+k_off) * N0
    I0 = k_off / (k_on+k_off) * N0
    [r1, r2, r3] = np.roots([1, k_off+k_on+2*k_ph+k_off_cy+k_on_cy, k_off*(
        k_ph+k_off_cy)+k_ph*(k_on+k_ph+k_off_cy)+k_on_cy*(k_off+k_on+2*k_ph),
         k_on_cy*k_ph*(k_ph+k_off+k_on)])
    return (R0+I0*(1+(k_ph+r1)/k_off))*((k_ph+r2)*(k_ph+r3))/((r1-r2)*(r1-r3
        ))*np.exp(r1*tdata) + (R0+I0*(1+(k_ph+r2)/k_off))*((k_ph+r1)*(k_ph+r3
        ))/((r2-r1)*(r2-r3))*np.exp(r2*tdata) + (R0+I0*(1+(k_ph+r3)/k_off))
        *((k_ph+r1)*(k_ph+r2))/((r3-r1)*(r3-r2))*np.exp(r3*tdata)
```

B.4 Méthode GRID

Algo 8 – get_up_down_estimate_k

```
def spectrumfitwrapper(time, data, k, E):
    """
    ---- Inputs
    time : A list of times
    data : data[i] is the number of remaining molecules at time time[i]
    k : the list of decay rates that are tested; typically regularly spaced
        points between 10^-3 and 10^3 (regularly spaced in log scale)
    E : The regularisation weight. 0.01 seems to be working fine in my
        examples
    ---- Output
    paraopt : the optimal weights associated with each rate in k. It is a
        list with the same length as k, and its sum is one
    output : more detailled output
    """
    N = len(k)
    S = np.zeros(N)

    # Bounds for amplitudes
    lb = S
    ub = S+1

    # Initial guess, uniform
    x0 = S + 1.0/N

    # Equality constraint (sum of amplitudes = 1)
    Aeq = np.ones(N)
```

```
    eq_constraint = {
        "type": "eq",
        "fun": lambda x: np.dot(Aeq, x) - 1,
        "jac": lambda x: Aeq
    }

    # Objective/Cost function to minimize
    def obj_wrapper(para):
        val, grad = lsqobj(para, k, time, data, E)
        return val, grad

    def obj_fun(para):
        val, _ = obj_wrapper(para)
        return val

    def obj_jac(para):
        _, grad = obj_wrapper(para)
        return grad

    # ------------------------------------------------------------------
    # Run optimization
    # ------------------------------------------------------------------
    res = minimize(
        fun = obj_fun,
        x0 = x0,
        jac = obj_jac,
        bounds = list(zip(lb, ub)),
        constraints = [eq_constraint],
        method = 'SLSQP',
        options = {"maxiter": 1000, "disp": False}
    )

    paraopt = res.x

    # Compute final error (with E = 0)
    error_val, _ = lsqobj(paraopt, k, time, data, 0)

    output = {
        "scipy_output": res,
        "error": error_val
    }

    return paraopt, output

# Cost function to minimize
def lsqobj(S, k, time, data, E):
    """
    ----Input
    S : weights associated with each rate in k
    k : List of decay rates
```

```
    time : A list of times
    data : data[i] is the number of remaining molecules at time time[i]
    E : Regularization weight
    ---- Output
    d : Function value
    grad : Gradient function value
    """

    # ------------------------------------------------------------------
    # Cost from difference between data and model
    # ------------------------------------------------------------------

    p = np.asarray(time)
    ftarg = np.asarray(data)

    # Model functions
    h = calch(S, k, p)

    # squared error equation
    eq0 = h / h[1] - ftarg / ftarg[1]
    ceqt = eq0**2

    # gradients
    gradh1 = gradh(eq0, h, p, k)
    gradceqt = np.concatenate([gradh1])

    # Cost and cost gradient
    ceq = np.sum(ceqt)
    gradceq = gradceqt

    # ------------------------------------------------------------------
    # Regularization
    # ------------------------------------------------------------------
    tau = np.array([time[0], time[1]])
    A = Transmat(tau, k)
    kquer = (A @ (k * S)) / (A @ S)

    reg = 0.5 * (kquer[0] - kquer[1])**2

    # gradient of regularization wrt S
    temp1 = (A[0] / (A[0] @ S)) * (k - kquer[0])
    temp2 = (A[1] / (A[1] @ S)) * (k - kquer[1])
    gradreg = (kquer[0] - kquer[1]) * (temp1 - temp2)

    # ------------------------------------------------------------------
    # Final cost and gradient
    # ------------------------------------------------------------------
    d = ceq + E * reg
    grad = gradceq + E * gradreg
```

```
    return d, grad

# ------------------------------------------------------------------------
# Helper functions
# ------------------------------------------------------------------------

def calch(S, k, t):
    """ Compute h = A*S' where A is Laplace transform matrix. """
    A = Transmat(t, k)
    return A @ S

def gradh(eq0, h, t, k):
    """ Gradient wrt S (decay spectrum amplitudes). """
    A = Transmat(t, k).T
    term = 2 * eq0 * (A / h[1] - (A[:, 1:2] * (h / h[1]**2)))
    return term.sum(axis=1)

def Transmat(t, k):
    """
    returns the Laplace transform matrix A with A[m,n] = exp(-k[n]*t[m]).
    t: vector of times
    k: vector of decay rates
    """
    return np.exp(-np.tensordot(t, k, axes=0))

def get_fit_Laplace(t, k, S):
    return np.sum(S * Transmat(t, k), axis=1)

# The function spectrumfitwrapper computes the inverse Laplace transform
# If we want to give as input data = a histogram of residence times, then
   there is a biais in the size of the populations:
# The weight of a population of rate k and computed weight S is actually
   proportionnal to S/k
# We return this specific value (renormalized so that the sum is one) here:
# (pdf stands for probability distribution function here)
def spectrumfitwrapper_pdf(time, data, k, E):
    S, _= spectrumfitwrapper(time, data, k, E)
    total = np.sum(S/k)
    return S/(k*total)

# ------------------------------------------------------------------------
# Plotting the curve and the inverse Laplace transform
# ------------------------------------------------------------------------

def plot_inv_Laplace_transf(time, data, k, E):
    res = spectrumfitwrapper(time, data, k, E)
    S = res[0]
    fig, [decay, Lapl] = plt.subplots(2, 1)
    decay.semilogy(time, data, label = 'data')
    decay.grid(visible=True)
```

```
    decay.set_xlabel('Time')
    decay.set_ylabel('Proportion of occurences')
    decay.semilogy(time, data[1] * np.sum(S * Transmat(time, k), axis=1)/np.
       sum(S * np.exp(-time[1]*k)), label = 'fit')
    decay.legend()
    Lapl.bar(k, S, width = 0.7*k*(k[1]/k[0]-1)) # Widths: in order to have a
        approximately constant width in log-scale
    Lapl.set_xscale('log')
    Lapl.grid(visible=True)
    Lapl.set_xlabel('Rate (time^-1)')
    Lapl.set_ylabel('Weight')
    plt.show()

def plot_decomp_expo(time, data, k, E):
    S = spectrumfitwrapper_pdf(time, data, k, E)
    fig, [decay, Lapl] = plt.subplots(2, 1)
    decay.semilogy(time, data, label = 'data')
    decay.grid(visible=True)
    decay.set_xlabel('Time (s)')
    decay.set_ylabel('Number of occurences')
    decay.semilogy(time, data[1] * np.sum(S * k * Transmat(time, k), axis=1)
       /np.sum(S * k * np.exp(-time[1]*k)), label = 'fit')
    decay.legend()
    Lapl.bar(k, S, width = 0.7*k*(k[1]/k[0]-1)) # Widths: in order to have a
        approximately constant width in log-scale
    Lapl.set_xscale('log')
    Lapl.grid(visible=True)
    Lapl.set_xlabel('Rate (s^-1)')
    Lapl.set_ylabel('Weight')
    plt.show()
```

Références

[1] Robin FB, McFadden WM, Yao B, Munro EM. *Single-molecule analysis of cell surface dynamics in Caenorhabditis elegans embryos*. Nature Methods. 2014 Jun ;11(6) :677-82. https://doi.org/10.1038/nmeth.2928.

[2] Matthias Reisser, Johannes Hettich, Timo Kuhn, Achim P. Popp, Andreas Große-Berkenbusch1, J. Christof M. Gebhardt. *Inferring quantity and qualities of superimposed reaction rates from single molecule survival time distributions*. Scientific Reports 10, 1758, (2020). https://doi.org/10.1038/s41598-020-58634-y.