Université de Montréal

PHY 3075 – Modélisation numérique en physique

Projet 3 - Calcul sur réseau

---

par :

Patrice Béchard

20019173

14 mars 2017

1 Introduction

Le calcul sur réseau est une méthode numérique permettant d'effectuer une suite d'opérations sur un nombre d'éléments connectés entre eux et dépendant chacun d'eux. Ce réseau peut en théorie être autant un réseau périodique et géométrique qu'un réseau aléatoire.

L'étude du modèle d'Ising et de la gravure magnétique a été conduite en effectuant un calcul sur réseau pour connecter toutes les composantes du système. Il est ainsi possible de modéliser l'interaction d'un amalgame de particules ordonnées et liées de façon méthodique. En connaissant l'orientation des voisins de chacune des cellules d'un réseau, il est possible de modéliser numériquement le modèle d'Ising pour la magnétisation d'un matériau.

L'étude de la propagation d'un virus sur un réseau suivant le modèle de Barabási-Albert a aussi été étudiée en suivant divers modèles épidémiques, soit les modèles SIR, SEIR et SIRS.

Ce document présentera tout d'abord une validation du programme construit, suivi d'une présentation des résultats obtenus pour la simulation du modèle d'Ising et de la gravure magnétique, puis se concluera par une exploration numérique d'une propagation épidémique sur un réseau réel. Les différentes figures présentes dans ce document, ainsi que les codes utilisés et quelques animations sont fournies sur le dépôt public situé à l'adresse suivante : `goo.gl/FeIY1o`

2 Validation du code

La première étape sera de reproduire certaines figures des notes de cours pour ainsi valider le programme construit et s'assurer qu'il effectue bien ce qui est attendu de lui. Pour commencer, pour s'assurer que notre simulation numérique du modèle d'Ising est correcte, les figures 3.15 et 3.16 des notes de cours ont été répliquées. Celles-ci sont présentés aux figures 1 et 2.

Le comportement de l'énergie par spin en fonction de la température présenté à la figure 1 est identique à la figure 3.15 des notes de cours, mis à part les fluctuations reliées à la configuration initiale aléatoire du système. Chaque courbe tracée pour chaque température se stabilise aux mêmes valeurs qu'observé à la figure 3.15.

Le comportement de la magnétisation par spin en fonction de la température présenté à la figure 2 est encore une fois très similaire à la figure 3.16 des notes de cours. La courbe tracée pour la température $T = 5.00$ se stabilise à une valeur nulle, alors que celle tracée pour la température $T = 2.35$ oscille aléatoirement autour de cette valeur. La courbe tracée pour la température $T = 1.5$ est cependant renversée par-rapport à celle présentée à la figure 3.16. Cette situation est cependant tout-à-fait normale, mais cette fois-ci, le système s'est stabilisé dans l'état où tous les spins du réseau possèdent une valeur de

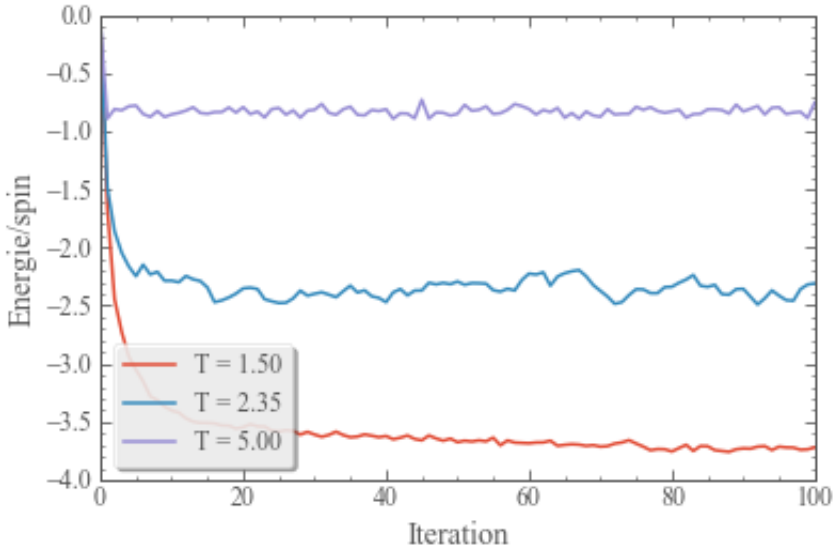

FIGURE 1 – Séquences temporelles de l'énergie moyenne par spin dans trois simulations sur réseau 64×64 avec $J = 1$ et $H = 0$, pour trois températures différentes, tel qu'indiqué. Un réseau où tous les spins sont parfaitement alignés aurait une énergie par spin de -4 ici.

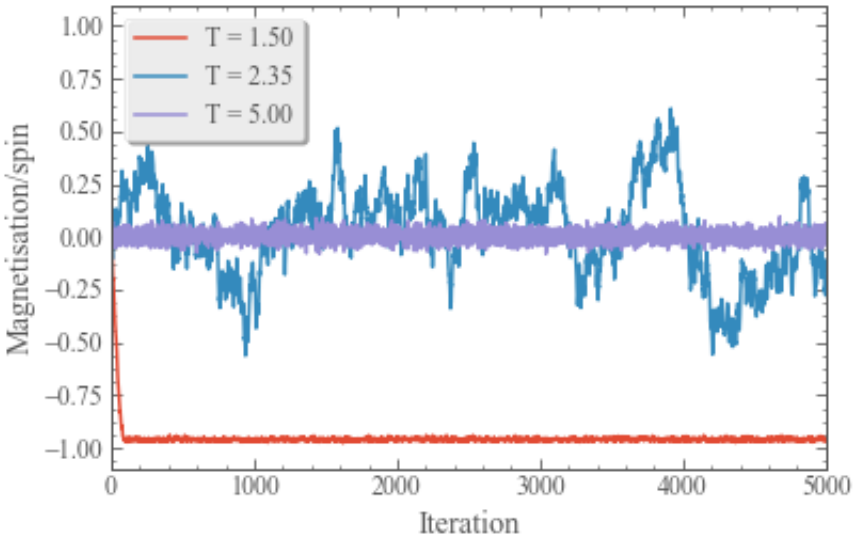

FIGURE 2 – Séquences temporelles de la magnétisation m pour les trois même simulations que ci-dessus. Notez l'échelle temporelle poussée ici à 5000 itérations, versus seulement 100 sur la Fig. 1.

-1 au lieu d'une valeur de 1. Ces deux situations possèdent la même énergie par spin, alors celles-ci sont équiprobables

Des animations de l'évolution du modèle d'Ising à ces températures ont été faites sur 500 itérations et sont disponibles sur le dépôt public dont l'adresse est donnée plus haut. Pour une température $T = 1.5$, toutes les cellules du réseau vont converger vers un même spin. Pour une température $T = 5.0$, les fluctuations thermiques sont trop grandes, et aucune domaine local n'apparaît. Le réseau est alors en régime presque complètement aléatoire. Finalement, pour une température $T = 2.35$, des régions où le spin est identique se développent et changent dans le temps sans jamais dominer complètement le réseau total. Des domaines de magnétisation locaux apparaissent et disparaissent au fil de la simulation.

La seconde partie de la vérification du programme consiste à répliquer la figure 3.20 des notes de cours, montrant le comportement du modèle d'Ising lorsqu'une magnétisation extérieure est introduite. Celle-ci sera tout d'abord une magnétisation variant sinusoïdalement sur une période de 500 itérations. Le résultat de cette simulation est présenté à la figure 3.

Le comportement de la magnétisation sur la première ainsi que la dernière figure sur la Fig.3 est identique au comportement de la magnétisation sur la figure 3.20 des notes de cours. Cependant, le comportement de la magnétisation sur les deux figures du centre présentent moins d'anomalies que la figure des notes de cours, signe que les algorithmes utilisés n'ont pas exactement le même comportement dans les deux cas. La simulation a été répétée plusieurs fois pour s'assurer qu'il ne s'agissait pas seulement d'une erreur due à la configuration aléatoire initiale, et aucune différence n'a été observée dans ces autres simulations.

La dernière étape de la validation consiste à chauffer une partie du domaine pendant un cours instant dans le but de graver un bit sur le réseau. La figure 3.22 des notes de cours a été répliquée pour montrer ce phénomène et est présentée à la figure 4

La figure produite ressemble très fortement à celle présentée dans les notes de cours. Quelques détails sont tout de même à soulever. Tout d'abord, la magnétisation totale pour la gravure réussie d'un bit est plus petite que celle présentée dans les notes de cours. Cela est tout simplement dû au fait que le bit gravé est de plus grande taille. Ce détail dépend de la séquence de nombres aléatoires utilisés. Ensuite, le cas où le bit gravé *envahit* le domaine au complet semble se propager plus rapidement que sur la figure 3.22 des notes de cours. La même raison est ici en cause. C'est cette raison qui cause aussi la convergence très rapide vers la configuration initiale du bit gravé dans le cas où il ne se propage pas, mais est plutôt effacé par le système. Plusieurs simulations du même système nous montre très clairement que la suite de nombres aléatoires choisis influence grandement le destin du bit gravé pour des températures critiques.

Une animation de la gravure du bit dans le cas où celle-ci est réussie et stable est fournie dans le dépôt

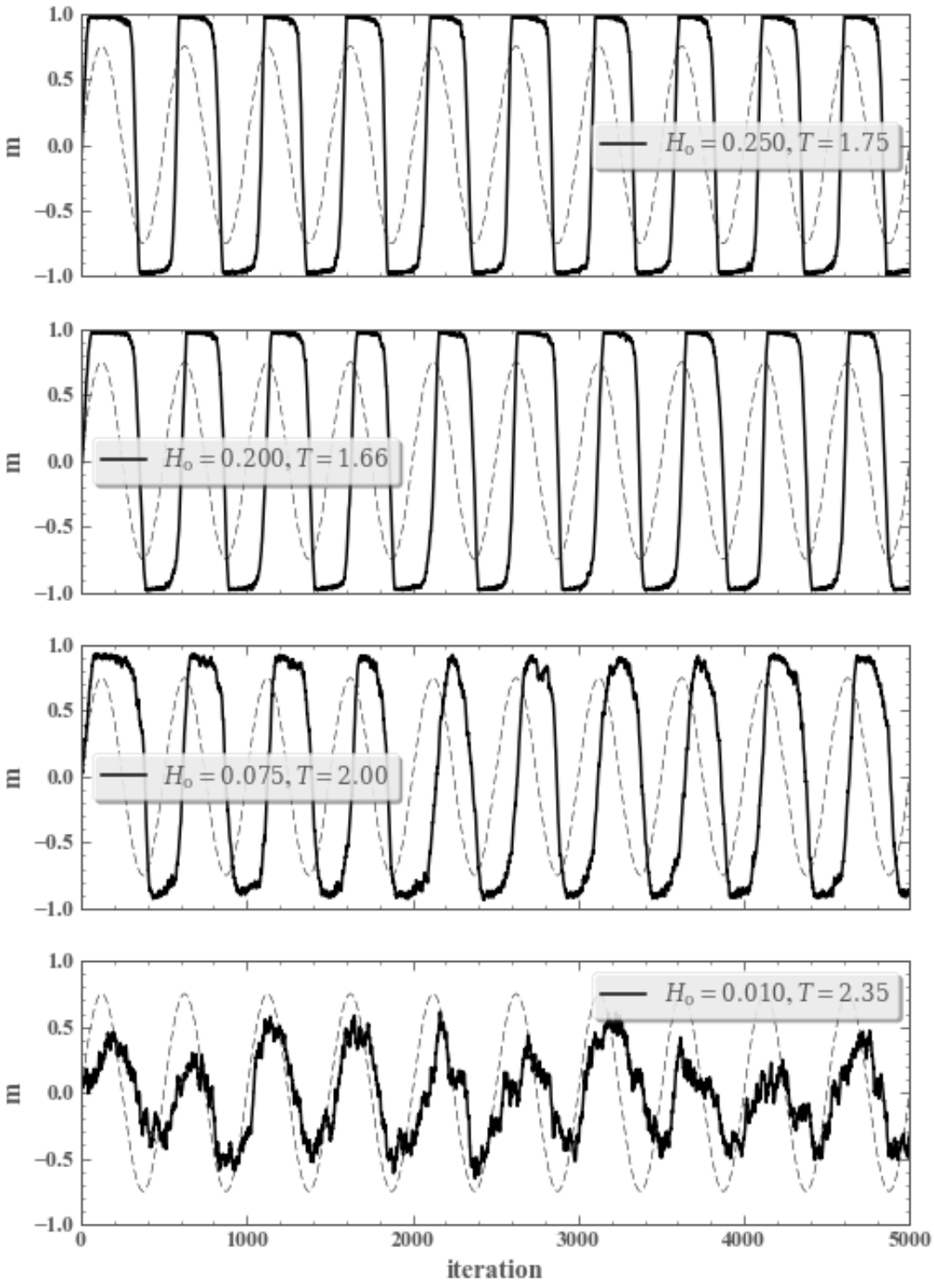

FIGURE 3 – Séquences temporelles de la magnétisation moyenne par spin (traits pleins) dans un réseau 64×64 avec J = 1, sujet à une magnétisation extérieure variant sinusoïdalement dans le temps (traits pointillés). Les quatre solutions sont obtenues avec des amplitudes de magnétisation extérieure H_0 et température T telles qu'indiquées. Dans tous les cas la condition initiale est aléatoire, et la période de la magnétisation extérieure est $P = 500$ itérations.

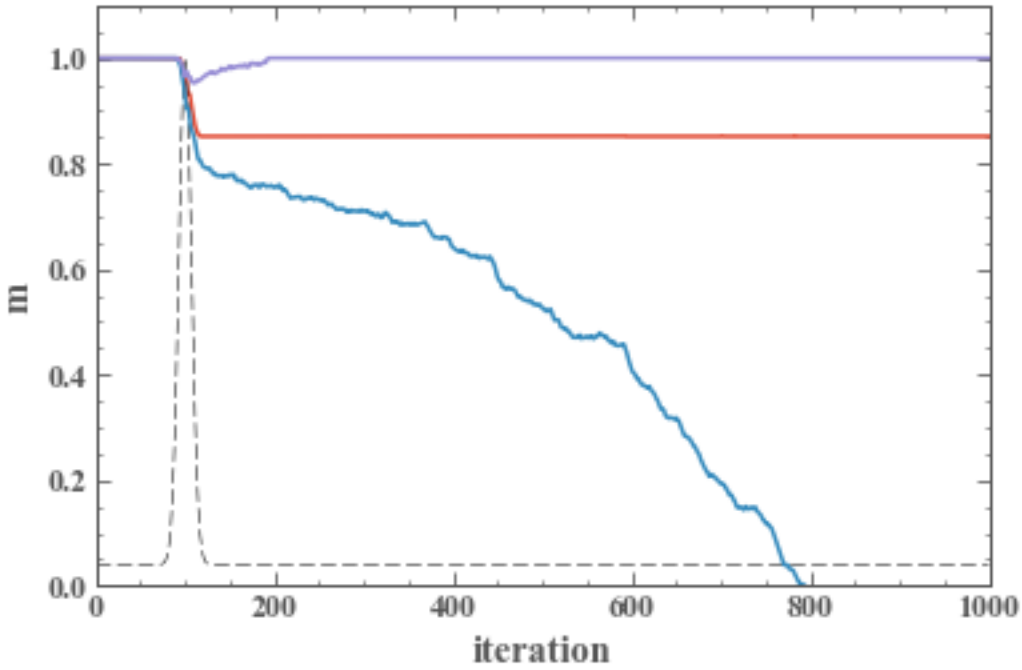

FIGURE 4 – Séquences temporelles de la magnétisation moyenne par spin sur un réseau 64×64 avec $J = 1$, initialement magnétisé positivement ($\uparrow$) partout, et sujet à un champ extérieur $H =?0.2$ et un pulse thermique de forme circulaire en son centre (ligne pointillée montre l'amplitude). Les paramètres du profil de chauffage (voir notes de cours) sont $(j_0, k_0) = (32, 32)$, $R = 10$, $t_0 = 100$, et $\sigma = 10$. La courbe s'immobilisant à ≈ 0.85 utilise $T_0 = 0.1$ et $\Delta T = 2.4$, tandis que pour les deux autres courbes utilisent $T_0 = 0.5$ et $\Delta T = 2.0$, ces deux simulations ne différant qu'au niveau de la séquence de nombres aléatoires utilisée.

public mentionné plus tôt. La figure 5 montre le bit tracé stable à la fin de la simulation.

3 Résultats et discussion

Pour l'étape de la vérification, un *stencil* de Von Neumann a été utilisée. Il est possible de répéter la simulation pour différents stencils, soit le stencil de Moore ainsi que le stencil triangulaire. Cette section présentera l'évolution du modèle d'ising selon ces deux différents stencils, ainsi que la gravure magnétique d'un bit grâce à ceux-ci.

Tout d'abord, il est possible de modifier l'algorithme d'évolution de la grille pour faire évoluer le système en prenant un différent stencil que le stencil de Von Neumann, n'utilisant que les 4 voisins immédiats de la cellule à l'étude. Le stencil de Moore prend aussi en compte les quatre voisins diagonaux de la cellule d'intérêt, et ce, avec un poids deux fois moins grand que celui associé aux quatres voisins de Von Neumann. Le stencil triangulaire prend en compte le voisin nord-est ainsi que le voisin sud-ouest de la cellule d'intérêt avec le même poids que pour les quatre voisins de Von Neumann. L'algorithme d'évo-

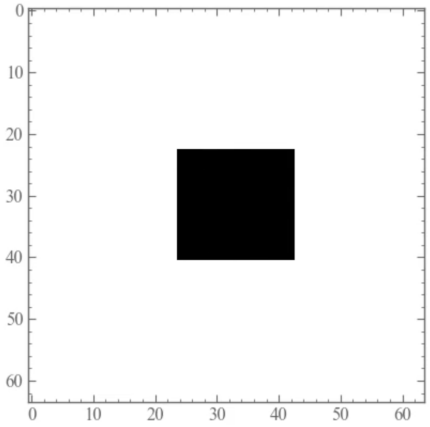

FIGURE 5 – Bit tracé par gravure magnétique sur le domaine avec une magnétisation extérieure $H = -0.2$ et un profil de température variant selon un profil gaussienne d'amplitude 2.4.

lution de type *damier* (voir notes) ne peut pas être appliquée avec ces stencils, puisque cela introduirait un biais dans notre système. Une méthode alternative est donc de copier la grille au complet avant l'évolution de celle-ci pendant une itération pour connaître l'alignement de chaque cellule avant l'évolution. Cette méthode n'est cependant pas tout-à-fait en accord avec l'algorithme de Métropolis responsable du changement d'orientation d'une cellule. En effet, dans le cas des trois stencils à l'étude, en partant d'une distribution des orientations aléatoire, la méthode d'évolution de *copie* tend vers un état *clignotant*, où chaque cellule change d'orientation à chaque itération dans la plage de températures où des domaines de magnétisation devraient apparaître. Un exemple de ce phénomène pour le stencil de Moore est présenté à la figure 7. Une animation de cette situation est fournie dans le dépôt public mentionné plus haut.

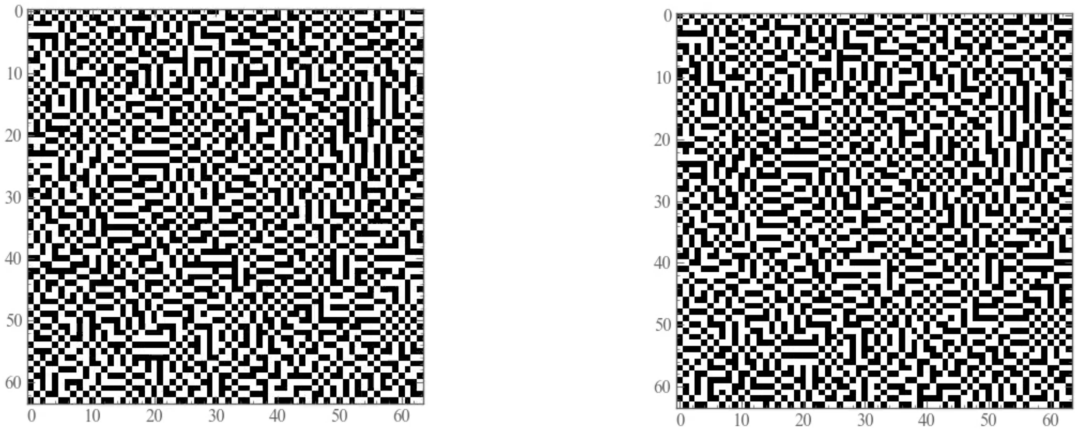

FIGURE 6 – Configuration de la grille pour deux itérations successives en utilisant le stencil de Moore avec la méthode d'évolution de *copie*. Chaque cellule possède des orientations inversées. Le modèle est dans une configuration *clignottante*. Ce problème survient avec tous les stencils utilisés.

Il est tout de même possible d'étudier la gravure d'un bit sur notre domaine en utilisant ces stencils. Cependant, les bits produits seront moins stables qu'ils l'étaient avec la méthode précédemment étudiée. Les bits obtenus avec la méthode d'évolution de copie pour les trois stencils utilisés sont présentés à la figure **??**. Des animations pour la formation de chacun de ces bits est disponible sur le dépôt public mentionné plus haut.

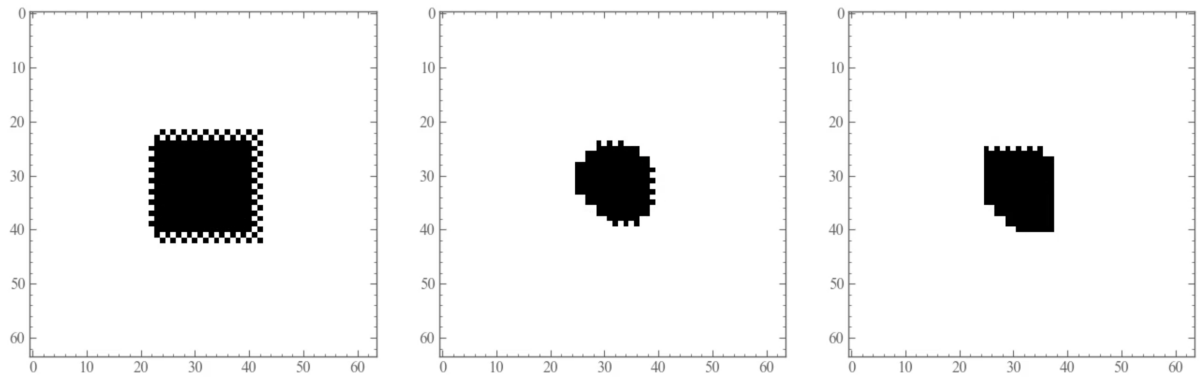

FIGURE 7 – Bits tracés par gravure magnétique pour la méthode d'évolution de copie pour les stencils de Von Neumann, Moore et triangulaire, respectivement. $\Delta t = 2.4$ pour le stencil de Von Neumann, et $\Delta t = 4.0$ pour les deux autres stencils.

Le bit tracé avec le stencil de Von-Neumann est carré et est entouré d'une région en régime clignotant (cellules alternées blanc/noir, voir animation). Le bit tracé avec le stencil de Moore possède plutôt une forme patatoïdale et ses bords sont constitués de cellules clignotantes elles-aussi. Finalement, le bit tracé par le stencil triangulaire n'est pas non plus rectangulaire et les bords sont aussi constitués de cellules clignotantes. Les bits sont donc tout de même tracés, mais leur stabilité est moins grande que pour le bit tracé dans la section de validation.

4 Exploration Numérique : Modèle de Barabási-Albert et propagation de virus

Plusieurs réseaux retrouvés dans la réalité ne sont pas que des réseaux symétriques comme étudiés jusqu'à présent : les liens entre les noeuds peuvent se partager aléatoirement ou selon d'autres règles mathématiques. À la fin des années 90, l'équipe du physicien Albert-László Barabási ont découvert, en mappant le domaine WWW de l'université Notre-Dame aux États-Unis, que la distribution de degré de chaque noeud (nombre de connections allant vers un noeud et partant de celui-ci) pouvait être représenté comme une loi de puissance. De plus, ils ont aussi découvert que plusieurs autres réseaux réels tels que le réseau électrique, le réseau des collaborations scientifiques, le réseau des acteurs d'Hollywood, les

interactions protéine-protéine et le métabolisme de la bactérie *E. Coli* suivaient aussi cette loi. Cette équipe a développé un modèle théorique de réseau invariant d'échelle possédant une distribution de degré allant en loi de puissance, appelé le *modèle de Barabási-Albert*, basé sur deux caractéristiques importantes : la *croissance* et l'*attachement préférentiel*, se basant donc sur l'évolution du réseau dans le temps pour faire émerger les caractéristiques s'apparentant aux réseaux réels[2]. Le document cité offre de plus amples détails sur les mathématiques reliés au modèle, ainsi que quelques algorithmes pour construire un réseau selon le modèle.

L'exploration numérique de ce chapitre portera donc sur la propagation d'information dans un tel réseau. Ce phénomène est facilement applicable à la réalité, par exemple lors de la propagation d'une maladie sur la planète lors d'une épidémie. Chaque noeud représentera un aéroport, lesquels seront reliés par des vols. Plusieurs noeuds ne possèdent qu'un petit degré (ex. aéroport Saguenay-Bagotville, aéroport Lyon Saint-Exupéry), mais un plus petit nombre d'aéroport possèdent un très grand degré (ex. aéroport LAX, aéroport Charles de Gaulle), appelés *hubs*. Il est facile d'imaginer que si un passager atteint d'un virus passe par un de ces aéroports très achalandés, les probabilités que celui-ci infecte d'autres passagers et que le virus se transmette à plusieurs autres endroits sur la planète devient très grand. Il est donc important de prévoir ce genre de scénario pour empêcher une catastrophe à grande échelle. Différents modèles épidémiques seront utilisés. Ceux-ci sont présentés plus bas[3][4]. La propagation des virus selon les différents modèles épidémiques se fera sur un réseau possédant 1000 noeuds que l'on a fait évoluer sur 100 itérations temporelles. Différentes animations de ces simulations sont fournies sur le dépôt public annoncé plus tôt.

Le premier modèle épidémique étudié sera le modèle *SIR*, développé par Kermack et McKendrick en 1927. Celui-ci inclue trois catégories de personnes :

- **S** : Les personnes *Susceptibles* sont celles qui peuvent contracter la maladie.
- **I** : Les personnes *Infectées* sont celles pouvant transmettre la maladie.
- **R** : Les personnes *Éliminées (Removed)* sont celles qui ont déjà contracté le virus et qui ne l'one plus. Ils ne peuvent pas la recontracter. Par exemple, ils ces personnes peuvent être mortes ou immunisés.

Ce modèle épidémique est plutôt simple, sans toutefois trop l'être pour devenir irréaliste. Ce genre de modèle épidémique peut être appliqué à la varicelle ou la mononucléose, par exemple, où il nous est impossible de le recontracter la maladie une seconde fois.

L'algorithme utilisé est le suivant :

1. On choisit un noeud au hasard dans le réseau et on l'infecte.
2. On regarde tous les noeuds voisins des noeuds infectés du réseau. Pour chaque noeud voisin, si un

nombre aléatoire tiré est plus petit qu'une certaine probabilité de transmission, le noeud est alors infecté ($S \to I$).

3. Si un noeud est infecté pendant un certain nombre d'itérations, on l'élimine des noeuds contaminés ($I \to R$).
4. On répète les étapes 2 et 3 pour un nombre d'itérations donné.

Tout d'abord, la probabilité d'infection d'un nouveau noeud par un noeud infecté a été variée entre 0 et 1 pour trouver comment la propagation d'un virus dépendait de celle-ci. Deux cas différents ont été tracés à la figure 8 et un graphique de la fraction de noeuds en santé à la fin de la simulation sur le nombre de noeuds total a été tracé en fonction de la probabilité d'infection à la figure 11

La figure 11 montre clairement un changement de phase du type "température de Curie" pour les différentes valeurs de probabilité d'infection. À basse probabilité ($\in [0, 0.2]$), à peu près toutes les simulations ne montrent aucune propagation du virus à grande échelle, alors que la proportion de la population en santé à la fin de la simulation est au-dessus de 90%. Le scénario opposé est vrai à haute probabilité ($\in [0.7, 1.0]$), alors qu'à peu près toutes les simulations montrent une population finale entièrement "morte". Un changement de phase apparaît pour les probabilités entre ces valeurs proposant que la propagation du virus dépend beaucoup de la structure aléatoire du réseau construit. En effet, si un virus passe par un *hub* avant l'atteinte à l'équilibre, la propagation du virus se fera beaucoup plus efficacement, puisqu'un plus grand nombre de personnes pourront contracter celui-ci.

Le second modèle épidémique étudié sera le modèle *SEIR*, où un nouveau paramètre apparaît :

— **E** : Après contraction du virus, la personne *Exposée* ne devient pas tout de suite contagieuse. C'est seulement après un certain temps qu'elle devient infectée et qu'elle peut transmettre la maladie.

Ce modèle est un peu plus réaliste que le dernier, puisqu'il prend en compte un certain temps où la personne ayant contracté le virus ne peut pas transmettre celui-ci.

L'algorithme utilisé précédemment changera donc un peu :

1. On choisit un noeud au hasard dans le réseau et on l'infecte.
2. On regarde tous les noeuds voisins des noeuds infectés du réseau. Pour chaque noeud voisin, si un nombre aléatoire tiré est plus petit qu'une certaine probabilité de transmission, le noeud est alors exposé ($S \to E$).
3. Si un noeud est exposé pendant un certain nombre d'itérations, il devient infecté ($E \to I$)
4. Si un noeud est infecté pendant un certain nombre d'itérations, on l'élimine des noeuds contaminés ($I \to R$).
5. On répète les étapes 2 à 4 pour un nombre d'itérations donné.

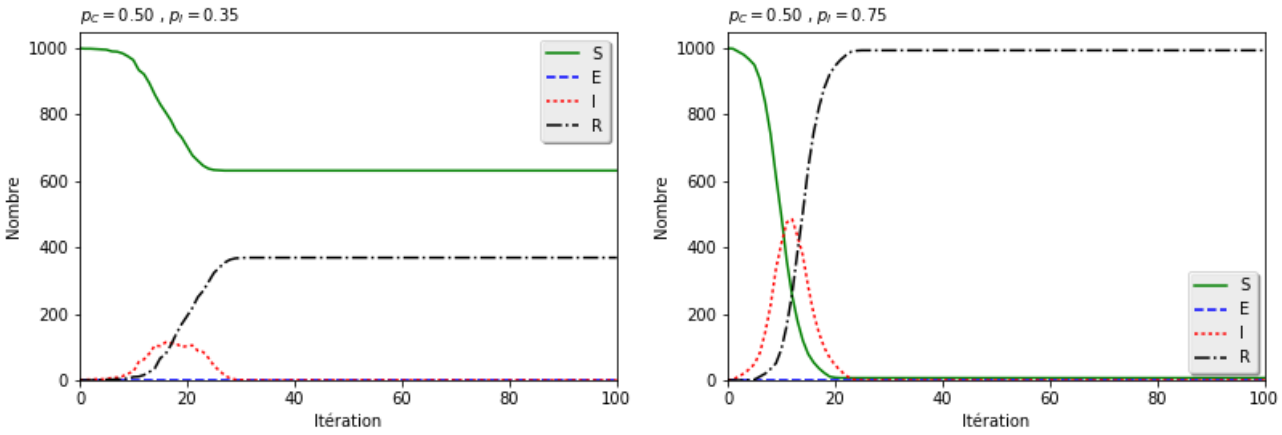

FIGURE 8 – Évolution des populations des différentes catégories de personnes pour des simulations avec des probabilités d'infection $p_I = 0.35$ et $p_I = 0.75$ pour le modèle épidémique SIR. La population de noeuds susceptibles de contracter la maladie (en santé) reste non-nulle après l'arrivée à l'équilibre dans la figure de gauche, alors que celle-ci est nulle lorsque la probabilité d'infection est trop élevée, comme dans la figure de droite. Le nombre maximal de personnes infectées en même temps est beaucoup plus grand pour la simulation à droite que celle à gauche. Une personne infectée reste dans cet état pendant 5 itérations, après quoi elle devient éliminée.

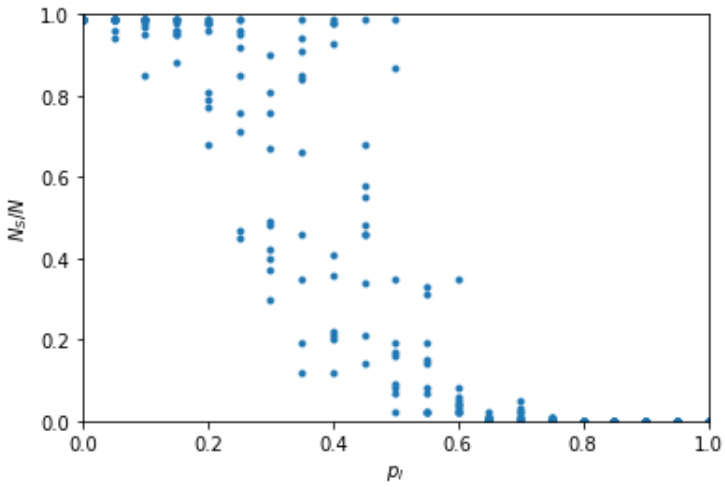

FIGURE 9 – Fraction de noeuds susceptibles de contracter la maladie (en santé) à la fin de plusieurs simulations en fonction de la probabilité d'infection pour 200 simulations.

L'effet de la probabilité d'infection sur la proportion de la population en santé ne sera pas à l'étude ici, à cause de la similarité avec le dernier modèle. Cependant, il vient intéressant de modifier le nombre d'itération après lesquels un noeud E deviendra un noeud I, ainsi que pour la transformation d'un noeud I en noeud R. La figure 10 montre deux cas où le seuil pour la transformation d'un noeud I en noeud R t_i était respectivement de 3 et 8 itérations pour des valeurs de $p_I = 0.5$ la probabilité d'infection et $t_E = 5$ le seuil pour la transformation d'un noeud E en noeud I.

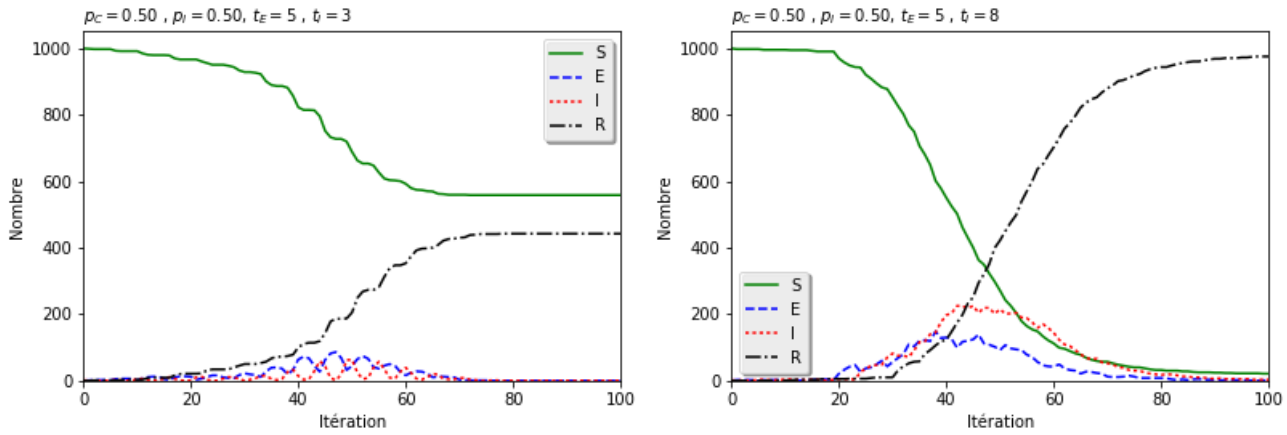

FIGURE 10 – Évolution des populations des différentes catégories de personnes pour des simulations avec des seuils de transformation $I \to R$ de $t_I = 3$ et $t_I = 8$ selon le modèle épidémique SEIR. La population de noeuds susceptibles de contracter la maladie (en santé) reste non-nulle après l'arrivée à l'équilibre dans la figure de gauche, alors que celle-ci est nulle lorsque la probabilité d'infection est trop élevée, comme dans la figure de droite. Le nombre maximal de personnes infectées ainsi que de personnes exposées en même temps est beaucoup plus grand pour la simulation à droite que celle à gauche. La probabilité p_I d'infection est gardée à 0.5 et le seuil de transformation $E \to I$ est gardé à $t_E = 5$.

La première chose à remarquer est le fait que l'arrivée à l'équilibre se produit beaucoup plus tard dans pour ce modèle épidémique que pour le modèle SIR étudié précédemment. De plus, le nombre total de personnes *enlevées* (R) à la fin est plus grand lorsque le seuil d'infection est plus élevé. Cela est prévisible, puisqu'un noeud contaminé aura plus de chances de contaminer ses noeuds voisins, vivant plus longtemps ainsi.

Le seuil de transformation $E \to I$ a ensuite été modifié en gardant $t_I = 5$ et $p_I = 0.5$ constants. La figure **??** montre la tendance des simulations pour en faisant varier t_E de 2 à 10.

La distribution de points engendré par les simulations en faisant varier le paramètre t_E ne semble pas faire varier significativement la fraction de noeuds en santé à la fin de la simulation. Cette fraction semble plutôt stable entre 0.2 et 0.4 pour la majorité des t_E testés, les points s'en écartant étant dû au caractère aléatoire de la structure du réseau.

Le dernier modèle épidémique étudié est le modèle *SIRS*. Cette fois-ci, la catégorie de population exposée n'est plus présente, mais une personne ayant déjà été infectée n'est plus immunisée pour le reste de la simulation. Une personne ayant été infectée, puis guérie (il n'y a pas de morts dans ce cas-ci) peut redevenir susceptible de contracter la maladie plus tard. Ce modèle épidémique représente bien une maladie tel qu'un rhume commun, pour lequel il n'est pas possible d'être immunisé.

L'algorithme utilisé pour cette simulation sera donc le suivant :

1. On choisit un noeud au hasard dans le réseau et on l'infecte.
2. On regarde tous les noeuds voisins des noeuds infectés du réseau. Pour chaque noeud voisin, si un nombre aléatoire tiré est plus petit qu'une certaine probabilité de transmission, le noeud est alors infecté ($S \to I$).
3. Si un noeud est infecté pendant un certain nombre d'itérations, on l'élimine des noeuds contaminés ($I \to R$).
4. Si un noeud est guéri depuis un certain nombre d'itérations, il redevient susceptible de contracter la maladie ($R \to S$).
5. On répète les étapes 2 et 3 pour un nombre d'itérations donné.

Contrairement aux autres modèles, ce modèle est cyclique. Le modèle est étudié en faisant varier les paramètres t_I, p_I et t_R séparément. La figure 12 montre deux simulations possédant tous les deux des probabilités d'infection différentes, mais avec des t_I et t_E identiques, montrant des comportements plutôt différents.

Les différentes simulations trouvent un état d'équilibre de sorte qu'il y a toujours une partie de la population dans chaque catégorie (en omettant la population *exposée* omise de ce modèle). Cependant, plus la probabilité d'infection est élevée, plus il y a une grande fluctuation de ces populations dans le temps, et ce, de façon périodique. En gardant la probabilité d'infection constante à 0.5 et en faisant varier le seuil $I \to R$, la figure 13 est obtenue, pour deux cas distincts. De plus, la figure 14 montre un graphique où une chaque pixel représente une moyenne sur 10 itération de la fraction de la population en santé sur la population totale à la fin de la simulation pour une certaine combinaison (t_I, t_R).

La figure 14 montre clairement le caractère observé sur les simulations présentés à la figure 13 : une simulation possédant un seuil k_I de 5 ou moins reviendra à la situation initiale avant la fin de la simulation. Cependant, pour $k_I \geq 6$, la simulation se retrouvera dans la plupart des cas dans une situation d'équilibre où la fraction de chaque population reste non-nulle, en fluctuant ou non. Cette caractéristique semble dépendre seulement du seuil k_I et non du seuil k_R, signe que le temps pour qu'une particule guérie redevienne susceptible de contracter le virus a peu d'importance sur la conclusion de la simulation.

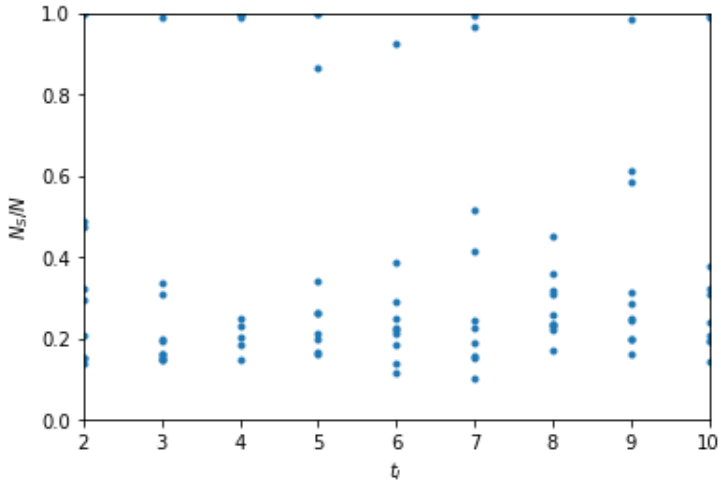

FIGURE 11 – Fraction de noeuds susceptibles de contracter la maladie (en santé) à la fin de plusieurs simulations en fonction du seuil t_E pour 90 simulations. Aucune tendance évidente ne semble ressortir du graphique.

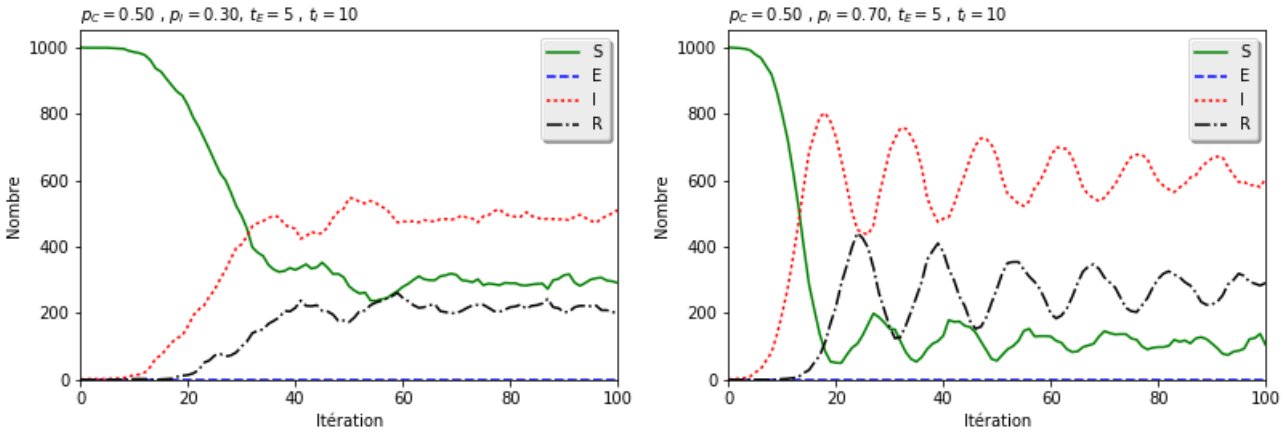

FIGURE 12 – Évolution des populations des différentes catégories de personnes pour des simulations avec des probabilités d'infection de 0.3 et 0.7, respectivement, selon le modèle épidémique SRIS. Lorsque la probabilité est basse, le système tend vers un état plutôt stable sans grande fluctuation de populations. Cependant, lorsque la probabilité d'infection est plus grande, le système rentre dans un état stationnaire. Pour ces simulations, les seuils sont fixés à $t_I = 10$ et $t_R = 5$.

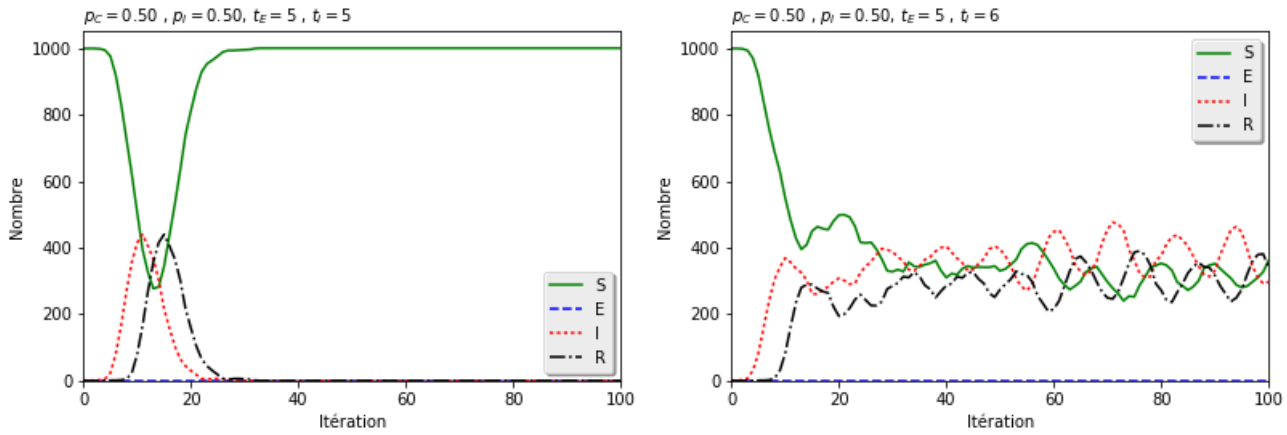

FIGURE 13 – Évolution des populations des différentes catégories de personnes pour des simulations avec des seuils t_I de 5 et 6, respectivement, selon le modèle épidémique SRIS. Lorsque le seuil de population infectée est de 5, le virus est rapidement irradié de la simulation, alors que la solution retrouve un état d'équilibre identique à son état initial. Cependant, lorsque le seuil est plutôt de 6, une situation d'équilibre différente est obtenue, encore une fois avec des fluctuations plutôt périodiques.

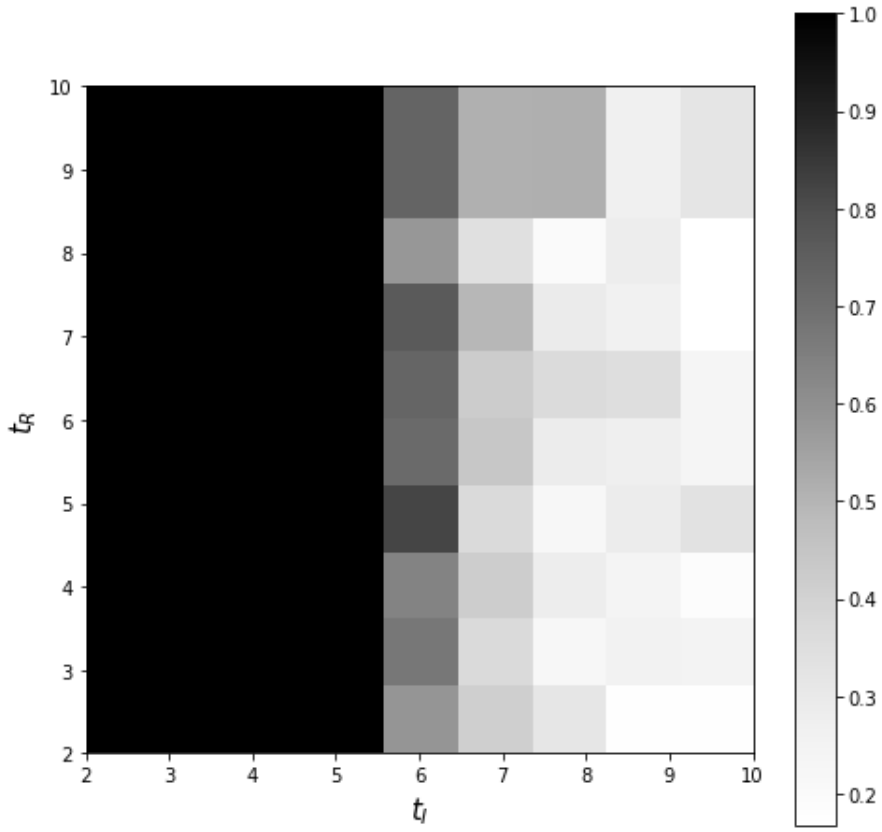

FIGURE 14 – Moyenne des fraction de la population en santé pour différentes combinaisons de seuil t_I et t_R.

Des animations de la propagation de virus dans un réseau suivant le modèle de Barabási-Albert sont disponibles pour les trois types de modèles épidémiques étudiés sur le dépôt public mentionné plus tôt. Il est intéressant de voir ce qui se passe lorsqu'un *hub* devient infecté, après quoi la propagation du virus se fait beaucoup plus rapidement, comme on pourrait l'imaginer si une personne atteinte d'une maladie comme Ebola passait par un aéroport important, pour revenir à l'exemple du début de cette section.

5 Programme

5.1 Programme principal

```
# -*- coding: utf-8 -*-
"""
Projet 3 - Gravure magnÃ©tique
Patrice BÃ©chard p1088418
mars 2017
"""
#------------------Modules-------------------------------------
import numpy as np
import matplotlib.pyplot as plt
import copy
import sys
import time
import scipy.constants as cst
from matplotlib import animation

plt.style.use('patrice')
print("Execution Start Time :",time.asctime())
start=time.time()            #time starts at beginning of execution
#--------------------Initial Conditions--------------------
"""
Grid will be size NxN (All are global variables)
config represents the initial configuration
    random : every cell is initialized randomly
    uniform-: every cell is initialized as '-1'
    uniform+: every cell is initialized as '+1'

evolmethod represents the way the grid is updated
    copy : all the cells are updated from an old version of the grid
    chess: cells are updates following a black/white cells oscillation
stencil represents the number of neighbors of each cell
    1 : von Neumann stencil (4 neighbors)
    2 : Moore stencil (8 neighbors)
    3 : Triangular stencil (6 neighbors)
"""
N=64                              #dimensions of grid
nIter=200                         #number of iterations
config='uniform-'
evolmethod='chess'
stencil=1

temperatures=[0.1]  #temperateures to loop over
source=[+0.2]              #and values of H_0
deltatemp=[2.4]
#--------------------Classes-------------------------------
class InputError(Exception):
    """Bad input for a given problem"""
```

```
    pass

class Parameters:
    """Default parameters for the simulation"""
    BOLTZMANN=1
    T0=0.1              #temperature of system
    DELTAT=2.4          #temperature pulse amplitude
    J=1
    R=10                #radius of temperature pulse area
    P=500               #period for oscillating H
    H0=-0.2             #outside magnetization term
    SIGMA=10
    X0,Y0=32,32         #where temperature pulse is centered
    time0=100           #time for temperature pulse

    GRAVURE=True
    VARH=False
    ANIMATION=True

class Grid(Parameters):
    """Grid containing each element of the network"""
    def __init__(self,dim,distribution):
        """Initialize dimensions and grid values depending on config"""
        if len(dim)!=2:
            raise InputError('Grid must be 2-D')
        self._dimensions=dim
        self._grid=[]
        if distribution=='uniform-':
            for i in range(self._dimensions[0]):#each node of the grid is a cell
                self._grid.append([-1 for j in range(self._dimensions[1])])
        elif distribution=='uniform+':
            for i in range(self._dimensions[0]):#each node of the grid is a cell
                self._grid.append([+1 for j in range(self._dimensions[1])])
        elif distribution=='random':
            for i in range(self._dimensions[0]):#each node of the grid is a cell
                self._grid.append([-1 if np.random.random()<0.5 else 1 for j in range(self._dimen
        elif distribution=='zeros':
            for i in range(self._dimensions[0]):
                self._grid.append([0 for j in range(self._dimensions[1])])
        else:
            raise InputError("Bad initial distribution. Please choose between 'uniform (+/-)' and

    def __getitem__(self,pos):
        """To access item via indexes"""
        if len(pos)!=2:
            raise IndexError("Grid is 2-D, enter index as [x,y]")
        if not (0<=pos[0]<self._dimensions[0] and 0<=pos[1]<self._dimensions[1]):
            raise IndexError("Out of bounds")
        return self._grid[pos[0]][pos[1]]
```

```
def evolution(self,cIter):
    """We make the grid evolve for one time step"""
    if evolmethod=='copy':
        self._copy_evol(cIter)
    elif evolmethod=='chess':
        self._chess_evol(cIter)
    else:
        raise InputError("Bad evolution method, please choose another one")
    ener=self.ener_spin()               #compute mean energy
    magnet=self.magnet_spin()           #compute mean magnetization
    return ener,magnet

def _copy_evol(self,cIter):
    """Evolving the system by copying old grid"""
    self._old=copy.deepcopy(self._grid)   #we are updating the grid, we deepcopy info in old
    self.H=self.define_H()
    for i in range(self._dimensions[0]):  #loop over all cells
        for j in range(self._dimensions[1]):
            self._count_neighbors([i,j])
            ener=-self.J*self._grid[i][j]*self._nb_neighbors\
                                    -self.H*self._grid[i][j]
            enerprime=-self.J*(-self._grid[i][j])*self._nb_neighbors\
                                    -self.H*(-self._grid[i][j])
            if self._metropolis(enerprime-ener,[i,j],cIter):
                self._grid[i][j]= -self._grid[i][j]

def _chess_evol(self,cIter):
    """Evolving the system with the chess method (see notes)"""
    self._old=copy.deepcopy(self._grid)   #we are updating the grid, we deepcopy info in old
    self.H=self.define_H()
    for i in range(N):          #white cells
        ij=(i%2)
        for j in range(ij,N,2):
            self._count_neighbors([i,j])
            ener=-self.J*self._grid[i][j]*self._nb_neighbors\
                                    -self.H*self._grid[i][j]
            enerprime=-self.J*(-self._grid[i][j])*self._nb_neighbors\
                                    -self.H*(-self._grid[i][j])
            if self._metropolis(enerprime-ener,[i,j],cIter):
                self._grid[i][j]= -self._grid[i][j]

    self._old=copy.deepcopy(self._grid)   #we are updating the grid, we deepcopy info in old
    for i in range(N):          #black cells
        ij=((i+1)%2)
        for j in range(ij,N,2):
            self._count_neighbors([i,j])
            ener=-self.J*self._grid[i][j]*self._nb_neighbors\
                                    -self.H*self._grid[i][j]
            enerprime=-self.J*(-self._grid[i][j])*self._nb_neighbors\
                                    -self.H*(-self._grid[i][j])
```

```
            if self._metropolis(enerprime-ener,[i,j],cIter):
                self._grid[i][j]= -self._grid[i][j]

    def _count_neighbors(self,pos):
        """Counts the neighbors of a cell"""
        self._nb_neighbors=0
        if stencil==1:
            self._von_neumann(pos)
        elif stencil==2:
            self._moore(pos)
        elif stencil==3:
            self._triangular(pos)
        else:
            raise InputError("Stencil doesn't exist, please choose another one")

    def _von_neumann(self,pos):
        """Von Neumann Stencil"""
        self.J=1
        for i in [-1,0,1]:
            for j in [-1,0,1]:            #loop over all neighbors of the cell
                if abs(i)+abs(j)!=1:      #excluding self and diagonals
                    continue
                elif (pos[0] + i) < 0 or (pos[1] + j) < 0:
                    continue              #avoiding negative index when on a side
                try:
                    self._nb_neighbors += self._old[pos[0]+i][pos[1]+j]
                except IndexError:        #out of bounds (cell on side)
                    continue

    def _moore(self,pos):
        """Moore Stencil"""
        self.J=1/3
        for i in [-1,0,1]:
            for j in [-1,0,1]:            #loop over all neighbors of the cell
                if abs(i)+abs(j)==0:      #excluding self
                    continue
                elif (pos[0] + i) < 0 or (pos[1] + j) < 0:
                    continue              #avoiding negative index when on a side
                elif abs(i)+abs(j)==1:
                    try:
                        self._nb_neighbors += 2*self._old[pos[0]+i][pos[1]+j]
                    except IndexError:        #out of bounds (cell on side)
                        continue
                elif abs(i)+abs(j)==2:
                    try:
                        self._nb_neighbors += self._old[pos[0]+i][pos[1]+j]
                    except IndexError:        #out of bounds (cell on side)
                        continue

    def _triangular(self,pos):
```

```
    """Triangular Stencil"""
    self.J=2/3
    for point in [[0,1],[0,-1],[1,0],[-1,0],[-1,-1],[1,1]]:
        if (pos[0] + point[0]) < 0 or (pos[1] + point[1]) < 0:
            continue              #avoiding negative index when on a side
        try:
            self._nb_neighbors += self._old[pos[0]+point[0]][pos[1]+point[1]]
        except IndexError:        #out of bounds (cell on side)
            continue

def _metropolis(self,deltaE,pos,cIter):
    """Probabilistic test"""
    prob=min(1,np.exp(-deltaE/(self.BOLTZMANN*self.temp_profile(pos,cIter))))
    if np.random.random()<=prob:
        return True
    return False

def define_H(self,param=False):
    """Outside magnetization term"""
    if self.VARH==True:
        return self.H0*np.sin(2*cst.pi*cIter/self.P)
    else:
        return self.H0

def ener_spin(self,param=False):
    """mean energy"""
    sum=0
    for i in range(N):
        for j in range(N):
            self._nb_neighbors=0
            if param:
                self._old=self._grid
            self._count_neighbors([i,j],)
            self.H=self.define_H()
            sum+=-self.J*self._grid[i][j]*self._nb_neighbors-self.H*self._grid[i][j]
    return (1/(N*N))*sum

def magnet_spin(self,param=False):
    """mean magnetization"""
    sum=0
    for i in range(N):
        for j in range(N):
            sum+=self._grid[i][j]
    return (1/(N*N))*sum

def space_profile(self,pos):
    """statial profile for the computation of the temperature pulse"""
    radius=(pos[0]-self.X0)**2+(pos[1]-self.Y0)**2
    return 1 if radius<=(self.R*self.R) else 0
```

```
    def temp_profile(self,pos,cIter):
        """Temperature profile for magnetic engraving"""
        if self.GRAVURE:
            return self.T0+self.DELTAT*self.space_profile(pos)*np.exp(-(cIter-self.time0)**2/self
        else:
            return self.T0

    def show_grid(self):
        """We show the playing grid cell by cell"""
        data=np.zeros([self._dimensions[0],self._dimensions[1]])
        for i in range(self._dimensions[0]):
            for j in range(self._dimensions[1]):
                data[i,j]=self._grid[i][j]
        plt.figure(figsize=(9,6))
        plt.imshow(data,cmap='Greys')
        plt.show()

#------------------------Functions--------------------------
def profile_H(grille):
    """Only used to plot sin wave"""
    return 0.75*np.sin(2*cst.pi*np.linspace(0,nIter,nIter+1)/grille.P)

def generate_evolution(grille,i,length):
    """
    Main part of the code
    Computing of energy, magnetization and animations is done here
    """
    cIter=0
    enermoy=[grille.ener_spin(True)]
    magnetmoy=[grille.magnet_spin(True)]
    profil=[grille.define_H(True)]
    Tprofil=[grille.temp_profile([grille.X0,grille.Y0],cIter)]
    if grille.ANIMATION:
        fig0=plt.figure(0)
        process=[]
        im=plt.imshow(grille._grid,animated=True,cmap='Greys')  #initial config
        process.append([im])
    for cIter in range(1,nIter+1):
        ener,magnet=grille.evolution(cIter)
        Tprofil+=[grille.temp_profile([grille.X0,grille.Y0],cIter)]
        enermoy.append(ener)
        magnetmoy.append(magnet)
        profil.append(grille.H)
        if grille.ANIMATION:
            im=plt.imshow(grille._grid,animated=True,cmap='Greys')
            process.append([im])
        if (cIter)%(nIter//10)==0:
            currenttime=time.time()-start
            print("%d  Elapsed time : %d h %d m %d s"\
                %(cIter,currenttime//3600,currenttime//60,currenttime%60))
```

```
    plt.figure(1)
    if grille.H0!=0:
        magnetization.append(magnetmoy)
    if grille.ANIMATION:
        spin_ani = animation.ArtistAnimation(fig0, process,interval=50)
        spin_ani.save('ising%d_%d.mp4'%(int(grille.T0*100),nIter))
    return
#-------------------------MAIN-----------------------------
grille=Grid([N,N],config)           #initialize grid

global magnetization
magnetization=[]
tempinit=True
for i in range(len(temperatures)):
    grille.T0=temperatures[i]                   #setting T0 for this simulation
    grille.H0=source[i]                         #and source term
    grille.DELTAT=deltatemp[i]                  #and delta T
    generate_evolution(grille,i,len(temperatures))    #evolution of the grid over nIter iteration
    grille=Grid([N,N],config)       #reset to initial config for next simulation


totaltime=time.time()-start
print("Total time : %d h %d m %d s"%(totaltime//3600,(totaltime//60)%60,totaltime%60))
```

5.2 Programme d'exploration numérique : Modèle de Barabási-Albert

```
# -*- coding: utf-8 -*-
"""
PHY3075 - Modele de Barabasi-Albert
Patrice Bechard
mars 2017
"""
import networkx as nx
import matplotlib.pyplot as plt
import random
import time
import copy
from matplotlib import animation
import numpy as np

start=time.time()

N0=2                #size of network at beginning
N=1000              #final size of network
nIter=100
probConnect=0.5
probInfect=0.5
"""
epidemic models (etype):
    1 : SIR
```

```
    2 : SEIR
    3 : SIRS
"""
etype=3

[S,E,I,R]=[[0 for i in range(N)] for j in range(4)]
thresE=5 ; thresI=10 ; thresR=5

#-------------------------FONCTIONS-------------------------
def create_network(G):
    for i in range(N0):
        G.add_node(i,status='S')
        S[i]=1
    G.add_edge(0,1)                          #we create the starting nodes
    for i in range(N0,N):
        connection=random.randrange(G.order())
        G.add_node(i,status='S')
        S[i]=1
        if random.random()<probConnect:
            G.add_edge(i,connection)
        else:
            G.add_edge(i,random.choice(G.neighbors(connection)))

def evolution(G):
    if etype==1:
        SIR(G)
    elif etype==2:
        SEIR(G)
    elif etype==3:
        SIRS(G)
    else:
        raise Exception ("Invalid epidemic model")

def SIR(G):
    temp_I=copy.copy(I)
    infect_neighbors(G)
    update_list(I,temp_I,thresI,R,'R')
    return

def SEIR(G):
    temp_E=copy.copy(E)
    temp_I=copy.copy(I)
    infect_neighbors(G)
    update_list(E,temp_E,thresE,I,'I')
    update_list(I,temp_I,thresI,R,'R')
    return

def SIRS(G):
    temp_I=copy.copy(I)
    temp_R=copy.copy(R)
```

```
    infect_neighbors(G)
    update_list(I,temp_I,thresI,R,'R')
    update_list(R,temp_R,thresR,S,'S')
    return

def infect(G,node,param=None):
    if etype!=2 or param is not None:
        G.node[node]['status']='I'
        I[node]=1
        S[node]=0
    else:
        G.node[node]['status']='E'
        E[node]=1
        S[node]=0

def update_list(status,copie,thres,nextstatus,change):
    for j in range(N):
        if status[j]>0 and status[j]==copie[j]:
            status[j]+=1
            if status[j]==thres:
                status[j]=0
                nextstatus[j]=1
                G.node[j]['status']=change

def infect_neighbors(G):
    Gprime=copy.deepcopy(G)
    for i in range(N):
        if Gprime.node[i]['status']=='I':
            for j in range(len(G.neighbors(i))):
                if Gprime.node[G.neighbors(i)[j]]['status']=='S'\
                        and random.random()<probInfect:
                            infect(G,G.neighbors(i)[j])
"""
dom=[i for i in range(2,11)]
dom2=copy.deepcopy(dom)
yes=[]
for thresE in dom:
    for thresI in dom2:
        value=0
        for nice in range(10):
            G=nx.Graph()
            create_network(G)
            pos=nx.spring_layout(G)
            infect(G,random.randrange(N),1)
            values=np.zeros(N)
            #nx.draw_networkx(G,pos=pos,cmap='cool',node_color=values,node_size=10,with_labels=Fa
            progress=np.array([])
            listS,listE,listI,listR=[],[],[],[]
            for i in range(N):
                if S[i]>0:
```

```
                listS.append(i)
            elif E[i]>0:
                listE.append(i)
            elif I[i]>0:
                listI.append(i)
            elif R[i]>0:
                listR.append(i)
        dataS,dataE,dataI,dataR=[len(listS)],[len(listE)],[len(listI)],[len(listR)]
        for i in range(nIter):
            evolution(G)
            listS,listE,listI,listR=[],[],[],[]
            for i in range(N):
                if S[i]>0:
                    listS.append(i)
                elif E[i]>0:
                    listE.append(i)
                elif I[i]>0:
                    listI.append(i)
                elif R[i]>0:
                    listR.append(i)
            dataS+=[len(listS)]
            dataE+=[len(listE)]
            dataI+=[len(listI)]
            dataR+=[len(listR)]

        value+=dataS[-1]/N

        plt.plot(np.linspace(0,nIter,nIter+1),dataS,'g',label='S')
        plt.plot(np.linspace(0,nIter,nIter+1),dataE,'b--',label='E')
        plt.plot(np.linspace(0,nIter,nIter+1),dataI,'r:',label='I')
        plt.plot(np.linspace(0,nIter,nIter+1),dataR,'k-.',label='R')
        plt.xlabel('ItÃ©ration')
        plt.ylabel('Nombre')
        plt.text(0,1.08*N,r'$p_C = $%.2f , $p_I = $%.2f, $t_E = $%d , $t_I = $%d'%(probConnec
        plt.axis([0,nIter,0,N*1.05])
        plt.legend(fancybox=True,shadow=True)
        value+=dataS[-1]/N

        if etype==1:
            plt.savefig('distSIR%d_%d_%d.png'%(N,nIter,int(probInfect*100)))
        elif etype==2:
            plt.savefig('distSEIR%d_%d_%d.png'%(N,nIter,int(probInfect*100)))
        elif etype==3:
            plt.savefig('distSIRS%d_%d_%d.png'%(N,nIter,int(probInfect*100)))
        plt.show()

    value=value/10
    print(value)
"""
G=nx.Graph()
```

```
create_network(G)
pos=nx.spring_layout(G)
infect(G,random.randrange(N),1)

def show_network(x):
    print('hey')
    evolution(G)
    listS,listE,listI,listR=[],[],[],[]
    for i in range(N):
        if S[i]>0:
            listS.append(i)
        elif E[i]>0:
            listE.append(i)
        elif I[i]>0:
            listI.append(i)
        elif R[i]>0:
            listR.append(i)
    print(x,"  elapsed : ",time.time()-start)
    nx.draw_networkx(G,pos=pos,node_color='g',node_size=10,with_labels=False,nodelist=listS,width
    nx.draw_networkx(G,pos=pos,node_color='y',node_size=10,with_labels=False,nodelist=listE,width
    nx.draw_networkx(G,pos=pos,node_color='r',node_size=10,with_labels=False,nodelist=listI,width
    nx.draw_networkx(G,pos=pos,node_color='b',node_size=10,with_labels=False,nodelist=listR,width

fig = plt.gcf()
anim = animation.FuncAnimation(fig, show_network, frames=nIter, interval=200)
#fig.legend(fancybox=True,shadow=True,handles=['g.','y.','r.','b.'],labels=['S','E','I','R'])

anim.save('barabasi%d_%d.mp4'%(N,nIter))

plt.show()
```

Références

[1] Charbonneau, P., *PHY3075 - Modélisation numérique en physique, Chapitre 3 : Équations différentielles partielles*, Université de Montréal, Montréal, 2017, 34p.

[2] Barabási, A.L., *Network Science*, Cambridge University Press, Cambridge, United Kingdom, 2016, 456p.

[3] Smith ?, R., «Simple Epidemic Models», [En ligne], University of Ottawa, [s.d.], 26p., `http://mysite.science.uottawa.ca/rsmith43/MAT4996/Epidemic.pdf`

[4] Wikipedia, «Epidemic Model», [En ligne], Wikipedia, [s.d.], `https://en.wikipedia.org/wiki/Epidemic_model`