RAPPORT DE PROJET INDUSTRIEL

-

Développement d'un simulateur d'épidémies végétales

Chef de projet

BLEICHNER Matthieu

Equipe projet

COSSET Benoit

DE RODELLEC Alexis

METAIS Vincent

SIMONNEAU Matthieu

Sommaire

I. Présentation du projet 9
A. Description du projet 9
1. Contexte d'étude 9
2. Contexte technique 9
B. Cahier des charges et spécifications 10
1. Développement d'un code efficace d'automate cellulaire 10
2. Optimisation du temps de calcul 13
3. Développement de modules d'affichage 13
4. Spécifications d'ordre général 13
C. Charte du projet 14
D. Organisation et gestion de projet 15
1. Parties prenantes 15
2. Planning et réunions 15
3. Activités et planning 16
4. Bilan à mi-parcours 23
II. Partie technique – solutions techniques apportées 25
A. Etudes des langages – choix et orientation 25
1. Etude de la technologie CUDA 25
2. Etude de la technologie C++ multithread 26
3. Etude de la Toolbox « Parallel Computing Toolbox » de Matlab 28
4. Etude de la Toolbox GPUmat 30
5. Etude de la technologie OpenCL 32
6. Récapitulatif 33
B. Choix et orientations 33

1. Choix de GPUmat 34

2. La programmation C++ Multithread 35

C. Automate cellulaire en Matlab 38

1. Description et explication du fonctionnement général de l'automate cellulaire codé uniquement en Matlab. 38

2. Voisinage implémenté 39

3. Evolution pour un voisinage de Moore étendu 41

4. Taille des matrices rsx, rsy et rsz dans le calcul de probabilité secondaire 43

5. Validation 45

6. Test 46

D. Le modèle SIR 48

E. Automate cellulaire en GPUmat 50

F. Développement en C++ multithreadé 51

1. Les différents composants du programme 52

2. Les échanges de données entre les composants 53

3. Description des différents programmes 54

4. Le multithreading 56

5. Tests et validations 58

G. Application concrète : Modélisation d'épidémies végétales 59

H. Comparaison des temps d'exécution 61

III. Lancement et utilisation des programmes 69

A. Automate cellulaire en Matlab SIDR 69

1. Fichiers 69

2. Fichier Run_me.m 69

3. Fonctions automate 70

4. Fonctions prob_primaire 72

5. Fonctions prob_secondaire 73

6. Fonctions prob_IversR 74
7. Fonctions prob_IversD 75
8. Fonctions prob_DversR 75
B. Automate cellulaire en Matlab SIR 77
1. Fichiers 77
2. Fichier Run_me.m 77
3. Fonction automate 78
4. Fonction prob_primaire 80
5. Fonction prob_secondaire 81
6. Fonction prob_IversR 82
7. Fonction prob_attibut_D 82
C. Automate cellulaire en GPUmat 83
1. Fonctions 84
2. Problèmes et limites 86
D. Développement en C++ multithreadé 87
1. Compilation des sources 87
2. Lancement d'une simulation 88
E. Interface Matlab et R 89
1. Interface Matlab 89
2. Interface R 91
IV. Annexes 97
A. Diagramme de Gant 97
B. Guide d'installation de Visual Studio 104
C. Guide d'installation de CUDA 106
D. Guide d'installation de GPUmat 106
1. Pré-requis 107
2. Comment installer GPUmat 107

3. Lancement de GPUmat 107

E. Guide d'installation de la librairie Boost 107

F. Guide d'installation R (et ses attributs) 108

1. Pour Windows 108

Remerciements

Nous tenons avant tout à remercier les personnes qui suivent pour leur aide et leur contribution, que s'ait été tout au long du projet ou ponctuellement :

- Mathieu FEUILLOY, Professeur ESEO

- Sylvain POGGI, Chercheur à l'INRA

- Melen LECLERC, Doctorant à l'INRA

- Nicolas PARISEY, Informaticien à l'INRA

- l'ensemble du corps professoral de l'ESEO

- le service informatique de l'ESEO

Introduction

Le Projet de Fin d'Etudes est un projet industriel mené par un groupe d'étudiants de dernière année à l'ESEO. Dans l'option Traitement du Signal et des Télécoms, plus de 200 heures par personne y sont consacrées durant le semestre.

Notre groupe était ainsi composé de 5 étudiants. Nous avions pour but de choisir un sujet intéressant afin de mettre en œuvre nos compétences et les méthodes de travail acquises au cours de notre formation, mais également un sujet plaisant et de notre goût. C'est pourquoi nous avons choisi le projet proposé par l'INRA de Rennes sur le développement d'un simulateur d'épidémies végétales.

Nous nous sommes ainsi organisés en fonction et investi dans ce projet selon nos compétences personnelles, mais aussi selon nos envies et nos affinités.

I. Présentation du projet

A. Description du projet

1. Contexte d'étude

Le sujet de ce projet est « Développement d'un simulateur d'épidémies végétales : implémentation & optimisation ». Ce projet permet avant tout aux chercheurs d'avoir une meilleure compréhension des modèles épidémiologiques, il participe aussi, entre autres, aux nombreuses actions menées dans le contexte du Plan Écophyto 2018 (Grenelle de l'Environnement) qui vise à réduire de 50% l'usage des pesticides au niveau national dans un délai de dix ans, si possible. Ainsi, les objectifs principaux mis en jeu sont :

- Une meilleure compréhension des processus gouvernant les épidémies afin d'optimiser les méthodes de gestion des maladies.
- Une proposition de méthodes innovantes afin de protéger les cultures.

Notre étude trouverait alors son utilité pour des développements réalisés par l'INRA.

2. Contexte technique

Afin de développer ce simulateur d'épidémies, il est bien important de tenir compte de deux paramètres :

- Tout d'abord, de la formalisation des mécanismes responsables de la propagation de la maladie, que nous avons assimilé au modèle épidémiologique SIDR (ou SIR dans certains cas). Avec la lettre S pour une cellule *Saine*, I pour *Infectée*, D pour *Détectée* et R pour *Retirée*.

- Ensuite, de la modélisation de la dynamique épidémique dans l'espace et dans le temps. Nous raisonnerons en effet sur des grilles à deux dimensions, qui représenteront les champs, et selon un pas de temps, représentant l'évolution dans le temps de ces cellules.

Ainsi, nous avons décidé de résonner avec des automates cellulaires, tout en devant surmonter une difficulté technique, celle de pouvoir réaliser un très grand nombre de simulations, et donc d'optimiser nos temps de calcul et notre code.

Il fallait donc partir de cette contrainte, ou plutôt de ce paramètre majeur, afin de définir pour quel type de langage et de technologie nous allions opter.

B. Cahier des charges et spécifications

1. Développement d'un code efficace d'automate cellulaire

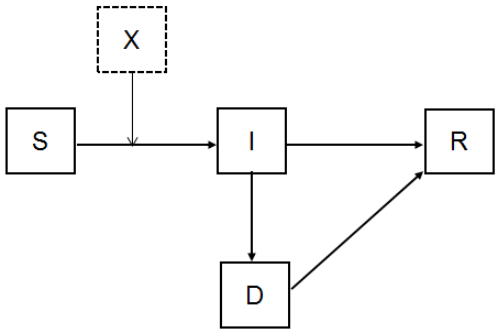
Les deux points importants imposés par le client sont :

- Une implémentation de l'automate cellulaire selon des paramètres pouvant varier (dimension, voisinage, transitions, pas de temps, coefficients de probabilités d'infection...)
- Un travail devant être le plus didactique possible afin qu'un chercheur modélisateur puisse rapidement pouvoir utiliser et modifier les outils créés

Ainsi les spécifications suivantes ont été établies.

a) Modèle épidémiologique

Nous utiliserons donc le modèle SIDR cité ci-dessus, avec toutefois un état implicite, nommé X, provoquant une infection primaire.



- Compartiment des plantes saines : S
- Compartiment des plantes infectées : I
- Compartiment X, implicite. Population de l'agent pathogène présent dans le sol de la parcelle
- Compartiment détecté : D
- Compartiment retiré : R

Une plante saine peut ainsi être infectée de deux façons :

- Par le champignon présent dans le sol. Il est alors associé un taux d'infection primaire (rp), qui correspond à la capacité de l'agent pathogène X à infecter une plante saine S qui passe alors dans l'état I. L'infection primaire est aléatoire et ne dépend pas directement du voisinage.
- Infecté par une plante qui est infectieuse. Il est alors associé un taux d'infection secondaire (rs), qui correspond à la capacité d'une plante I à transmettre la maladie à une plante S. Le voisinage est primordial dans l'infection secondaire.

b) Automate cellulaire

Afin d'étudier l'évolution de la maladie nous utiliserons donc un automate cellulaire. Pour chaque transition d'un état à l'autre, des fonctions de transfert sont définies telles que :

- Pour passer de l'état S à l'état I, la fonction à considérer est de la forme f(d,t) dans un premier temps, mais pourra être exprimée en fonction de l'âge des plantes infectées du voisinage ainsi que de l'âge de la plante saine attaquée.

- Les transitions I -> R et D->R sont définies soit par la date d'infection et l'âge de la plante à la date de l'infection, soit en fonction d'un nombre de pas prédéfini

- Pour passer de l'état I à D, la fonction de transfert est de la même forme que pour le passage de I à R.

Les fonctions utilisées pour le passage de l'état sain à l'état infecté sont les suivantes :

- Voisinage de von Neumann, dispersion isotrope:

$$\text{Prob}\left(S_t \rightarrow I_{t+1} \right) = 1 - \left(1 - r_p \right) \left(1 - P(d,t) \right)^{N(t)}$$

- Voisinage de von Neumann, dispersion anisotrope:

  $$\text{Prob}(S_t \rightarrow I_{t+1}) = 1 - (1-r_p) (1 - P(d_x,t))^{Nx(t)} (1 - P(d_y,t))^{Ny(t)}$$

- Voisinage de Moore, dispersion isotrope (2 distances possibles) :

  $$\text{Prob}(S_t \rightarrow I_{t+1}) = 1 - (1-r_p) (1 - P(d_1,t))^{N1(t)} (1 - P(d2,t))^{N2(t)}$$

- Voisinage de Moore, dispersion anisotrope (3 distances possibles) :

  $$\text{Prob}(S_t \rightarrow I_{t+1}) = 1 - (1-r_p) (1 - P(d_1,t))^{N1(t)} (1 - P(d_2,t))^{N2(t)} (1 - P(d_3,t))^{N3(t)}$$

Nx(t) : nombre de plantes voisines infectieuses selon l'axe x

dx : distance entre deux plantes voisines suivant l'axe x

c) Configuration de l'automate

Il sera possible de configurer l'automate cellulaire à l'aide des paramètres suivants:

- Les coefficients des probabilités d'infection primaire et secondaires, rp et rs,
- Le type de voisinage (von Neumann ou Moore),
- Le nombre de pas de l'exécution du programme,
- Le nombre de pas avant la transition I->R,
- La taille de la grille.

Il sera également possible de configurer les sorties désirées :

- Récupérer et afficher les statuts de la grille à chaque instant, à la fin de la simulation, ou bien à des instants prédéfinis
- Stocker l'évolution temporelle :
  - Du nombre de plantes saines,
  - Du nombre de plantes infectées,
  - Du nombre de paires plantes saines/à côté d'une plante saine,
  - Du nombre de paires plantes saines/à côté d'une plante infectée,

L'interface utilisateur devra être à la fois intuitive et simple d'utilisation.

2. Optimisation du temps de calcul

L'une des principales exigences est d'optimiser le temps d'exécution du programme. Pour cela, il sera nécessaire :

- De minimiser le temps d'exécution du cœur du programme en utilisant un langage de programmation adapté
- De choisir un langage optimal (étude et test de comparaison de différents langages, en termes de vitesse d'exécution)
- De comparer le temps d'exécution en fonction de différents OS
- De rechercher et de proposer des solutions innovantes, aussi bien logicielles que hardwares

3. Développement de modules d'affichage

Afin d'analyser correctement les simulations, il est important pour le client de pouvoir suivre graphiquement l'évolution du système.

- Représentation des états du système
- Représentation de l'état des cellules
- Représentation de l'évolution du nombre de plantes saines (S) et infectées (I), de l'évolution du nombre de paires SS, SI ou II, etc.
- Statistiques descriptives de base du système

Tous ces modules d'affichage seront à présenter en deux langages différents, Matlab et R.

4. Spécifications d'ordre général

Une documentation très précise de l'ensemble de nos démarches, qu'elles soient soit des échecs ou des réussites.

C. Charte du projet

1. Information générale	
Nom du Projet :	Développement d'un simulateur d'épidémies végétales : implémentation & optimisation
Project Sponsor:	Bleichner Matthieu

2. Définition du projet	
Quel est le produit ou le service final du projet?	Le but de ce projet est d'implémenter un programme qui simule la propagation d'une épidémie dans un champ. Pour cela nous devrons développer un automate cellulaire. L'objet de ce projet est d'apporter aux clients la possibilité de lancer un certain nombre de simulations puis d'étudier les résultats renvoyés par ces simulations.
Qui sont les utilisateurs finaux ?	Les chercheurs de l'INRA, ou toutes autres personnes désirant utiliser notre automate
Qui bénéficiera ? Comment?	Toute personne travaillant sur un modèle épidémiologique
Quels sont les facteurs de succès du projet ?	- Un programme efficient sous MATLAB - De faibles temps d'exécution
Quels sont les principaux livrables ?	- Un document MATLAB exécutable - Une dll et un fichier SO, contenant le cœur du programme. - Une fichier c pour coder les transitions. - Un rapport expliquant notre travail

3. Détails du projet			
Date du début :	06/11/2011	**Date de fin :**	28/01/2012

D. Organisation et gestion de projet

Un projet de 210h regroupant une équipe de cinq personnes nécessite une conduite de projet continue et maitrisée. Dans cette partie nous décrirons les différents outils utilisés lors de ce projet.

1. Parties prenantes

-Sylvain POGGI, Chercheur à l'INRA, commanditaire du projet

-Melen LECLERC, Doctorant à l'INRA, commanditaire du projet

-Nicolas PARISEY, Informaticien à l'INRA

-Mathieu FEUILLOY, Professeur ESEO

-Matthieu BLEICHNER, élève ESEO, chef de projet

-Benoit COSSET, élève ESEO, secrétaire

-Matthieu SIMONNEAU, élève ESEO, responsable qualité et développeur

-Alexis De RODELLEC, élève ESEO, développeur

-Vincent METAIS, élève ESEO, développeur

2. Planning et réunions

Afin de pouvoir suivre notre projet de façon continue nous avons décidé d'organiser des réunions hebdomadaires, dans un premier temps ces réunions ont eu lieu tous les Lundi à 13h30 jusqu'en Décembre, puis à partir du mois de Janvier à 7h45 le lundi matin.

Nous avons également organisé des réunions avec Monsieur POGGI et Monsieur LECLERC. Nous avons pu les rencontrer à deux occasions, une fois à l'INRA le 15 Novembre 2011 et une à l'ESEO le 04 Janvier 2012. La présentation finale du projet a été prévue à l'INRA le 27 Janvier 2012.

Ces réunions étant peu régulières nous avons organisé de nombreuses réunions téléphoniques dès que cela s'avérait nécessaire.

Finalement dans le but de toujours pouvoir quantifier notre avancement, après chaque séance nous avons rédigé un compte-rendu qui explique l'avancement du projet

3. Activités et planning

Nous avons divisé notre projet en activités et sous-activité, pour chaque activité nous avons défini les ressources, une date de début et une date de fin. La fin officielle du Projet Industriel est le 27 Janvier 2012, nous aimerions avoir terminé tout le code pour le 13 Janvier. Nous souhaiterions l'avoir envoyer à l'INRA pour cette date. Nous pourrons ainsi apporter les modifications nécessaires pendant la dernière semaine du projet. Nous utiliserons également ce temps pour terminer le rapport et préparer la soutenance.

Les principales activités que nous avons recensées sont :

- Le code MATLAB
- L'optimisation
- Le langage R
- Les interactions C++/Matlab et C++/R
- Le modèle SIR

Chaque activité comprend une phase implicite de rédaction du rapport.

Le diagramme de Gant est disponible en annexe.

- Code MATLAB

Taches	**Activité 1 : Spécifications**	**Activité 2 : Coder le programme en MATLAB**	**Activité 3 : post-traitement**	**Activité 4 : Test et validation**	**Activité 5 : GPUmat**
Travail à faire	-Détailler les spécifications,	-algorithme, pseudo langage -code MATLAB - développement des automates cellulaires. - tests et validation	-afficher la grille à chaque instant -calcul du nombre de paires	- tester le code	- Optimiser le code avec GPUmat
Ressources	Matthieu Bleichner Vincent Métais	Vincent Métais	Vincent Métais	Matthieu Bleichner Vincent Métais	Matthieu Bleichner
Date début	18/10/2011	04/11/2011	25/11/2011	30/11/2011	15/11/2011
Date fin (prévue)	8/11/2011	16/12/2011	29/11/2011	16/12/2011	16/12/2011
Date fin (effective)	8/11/2011	16/12/2011	02/11/2011	16/12/2011	16/12/2011
Nombre d'heures effectives	4 séances : 16h	25 séances :100h	5 séances : 20h	9 séances : 36h	21 séances : 86h
Document fourni	Liste des spécifications.docx	Dossier Code_Matlab	Dossier Code_Matlab	Dossier Code_Matlab	Installation et utilisation de GPUmat.docx

- Optimisation

Taches	**Activité 1 : Prospection de solutions pour optimiser le temps d'exécution**	**Activité 2 : formation sur les différents langages**	**Activité 3 : choix du meilleur langage**
Travail à faire	-particularités avantages/inconvénients de chaque solution -ex : CUDA, Openc, matlab parallèle, C++ multithreadé, autres	-formation sur C++ multithreadé et CUDA - installation du matériel et des bibliothèques nécessaires	- présentation à l'INRA -choix du meilleur langage
Ressources	Benoit Cosset, Alexis de Rodellec, Matthieu Simonneau	Benoit Cosset, Alexis de Rodellec, Matthieu Simonneau	B. Cosset, A.de Rodellec, M. Simonneau, M. Bleichner, V. Metais
Date début	18/10/2011	04/11/2011	15/11/2011
Date fin (prévue)	02/11/2011	14/11/2011	15/11/2011
Date fin (effective)	02/11/2011	14/11/2011	16/11/2011
Nombre d'heures effectives		12h	4h
Document fourni	Présentation des langages étudiés.docx	-Installation et utilisation de CUDA -Installation et utilisation de la librairie boost (C++ multithreadé).docx	- Compte Rendu rencontre INRA-ESEO 15.11.2011.docx

Taches	**Activité 4 : Code C++**	**Activité 5 : Code C++ multithreadé**	**Activité 6: Test et validation**
Travail à faire	- code C++ - test et validation -Vérifier que le projet C++ compile et s'exécute sous Linux. -Convertir les .dll en .so	Diviser le code C++ en taches indépendantes pour créer des threads et optimiser le temps de calcul -Vérifier que le projet C++ mt compile et s'exécute sous Linux. -Convertir les .dll en .so	- tester le code
Ressources	Alexis De rodellec Matthieu simonneau	Alexis De Rodellec Matthieu Simonneau Vincent Métais	Matthieu Simonneau Vincent Metais
Date début	16/11/2011	03/01/2012	03/01/2011
Date fin (prévue)	16/12/2011	13/01/2011	13/01/2011
Date fin (effective)	16/12/2011	13/01/2011	13/01/2011
Nombre d'heures effectives	20 séances : 80h		
Document fourni	Dossier Code_C++	Dossier Code_C++ multithreadé	

- Langage R

Taches	**Activité 1 : formation et découverte du langage**	**Activité 2 : Tests & Post-traitement**
Travail à faire	-Comprendre le langage - Charger une DLL - graphiques	-appeler le .so ou .dll depuis R -traiter les données en sortie - afficher l'état de la grille
Ressources	Benoit Cosset	Benoit Cosset Matthieu Bleichner
Date début	21/11/2011	26/11/2011
Date fin (prévue)	25/11/2011	16/12/2011
Date fin (effective)	02/01/2012	10/01/2012
Nombre d'heures effectives	4 séances : 16h	13 séances : 52h
Document fourni	Installation et utilisation du langage R.docx	lancementSimulation.R Documentation_utilisation_code_cpp.pdf

- Interactions MATLAB/ C++, R/ C++

Taches	**Activité 1 :** **MATLAB et C++**	**Activité 2 :** **R et C++**
Travail à faire	-Appeler la dll en C depuis Matlab -Tests et validations	-appeler le .so ou .dll depuis R -traiter les données en sortie - afficher l'état de la grille
Ressources	-Simonneau Matthieu	Benoit Cosset Matthieu Bleichner
Date début	-12/12/2012	26/11/2011
Date fin (prévue)	-16/12/2011	16/12/2011
Date fin (effective)	-16/12/2011	10/01/2012
Nombre d'heures effectives	3 séances : 12h	13 séances : 52h
Document fourni	lancementSimulation.m lancerSimulation.m	lancementSimulation.R Documentation_utilisation_code_cpp.pdf

- Compléments au code. Modèle SIR

Taches	**Activité 1 : MATLAB et C++**	**Activité 2 : Changement du code**
Travail à faire	- Analyser les codes et comprendre comment les changer pour répondre à la demande	- Apporter les modifications nécessaires
Ressources	- Alexis De Rodellec	- Alexis De Rodellec - Vincent Métais
Date début	05/12/2011	12/12/2011
Date fin (prévue)	09/12/2011	13/01/2012
Date fin (effective)	03/01/2012	14/01/2012 (Matlab)
Nombre d'heures effectives	24h	4h
Document fourni	-	Dossier Code_SIR

4. Bilan à mi-parcours

Nous avons décidé de réaliser une réunion d'avancement à mi-parcours afin de vérifier ce qui a été fait et ce qu'il reste à faire. Cette réunion a eu lieu le lundi 3 Janvier 2012, soit 120 heures après le début du projet et 96 heures avant la fin.

En ce qui concerne la partie du code Matlab, nous sommes bien dans les temps, le code a été envoyé à l'INRA le 16 Décembre. Nous attendons un retour pour pouvoir valider cette partie, ou bien ajouter les modifications nécessaires. Afin de comprendre le fonctionnement du programme ainsi que les différents paramètres d'entrée et de sortie nous avons rédigé un document explicatif : Rapport Matlab.docx. Pour GPUmat les changements apportés au code Matlab ont entrainé beaucoup de modifications, nous souhaitons avoir terminé cette partie pour la fin de la semaine.

En ce qui concerne le C++ simple, nous avons codé le cœur du programme et comme demandé dans le cahier des charges nous arrivons à appeler ce code depuis Matlab. Nous enverrons ce code à l'INRA dans la journée.

Pour l'interfaçage avec R, nous avons toujours des problèmes. En effet nous n'arrivons toujours pas à appeler une dll, sauf si elle a été compilée avec Codeblocks 32 bits. Nous travaillons toujours sur l'affichage des matrices. Une fois ces problèmes résolus nous nous pencherons sur le post traitement. Nous avons du mal à estimer le temps qu'il faudra pour résoudre ces problèmes.

Nous avons commencé le code C++ multithreadé, nous nous posons des questions sur la façon dont le multithreading doit être fait. En effet nous envisageons deux possibilités soit nous multithreadons les réplicas, soit nous divisons la grille en quatre et multithreadons chacune de ses parties. Le choix de l'algorithme dépend entre autres du nombre de réplicas désirés. Un document PowerPoint expliquant les deux cas de figures a été rédigé.

Finalement, pour le modèle SIR que vous nous avez demandé, où D est un attribut. Nous avons commencé à modifier le code C++ qui devrait être terminé d'ici la fin de semaine, nous mettrons ensuite à jour le code Matlab.

Pour conclure sur l'avancement global du projet, nous pensons être plutôt dans les temps. Nous avons pris du retard sur GPUmat et sur l'interfaçage R/C++ cependant nous respectons notre planning sur le Matlab général et sur le C++. La fin officielle du projet est fixée au 24 Janvier, nous espérons avoir terminé le 13 Janvier pour nous laisser une dizaine de jours pour apporter d'ultimes modifications et terminer le rapport. Enfin la soutenance de projet est prévue entre le 25 et 27 Janvier.

II. Partie technique – solutions techniques apportées

Dans cette partie nous nous pencherons sur l'architecture de nos programmes ainsi que sur les différentes fonctions implémentées pour chacun des langages.

A. Etudes des langages – choix et orientation

Dans le cadre de notre projet de fin d'étude il nous a été demandé par l'INRA de créer un simulateur de propagation d'une épidémie végétale. Ce simulateur devra être exécutable avec Matlab et R.

Pour étudier l'évolution de l'épidémie nous devrons utiliser un automate cellulaire codé dans un langage de programmation rapide. La simulation pouvant être lancée plusieurs centaines voire milliers de fois, il nous parait donc assez logique que le temps d'exécution du programme soit le plus faible possible.

Pour cela nous avons fait une étude de différents types de langages afin de choisir celui qui serait le plus approprié à notre projet.

1. Etude de la technologie CUDA

a) Description

CUDA (Compute Unified Device Architecture) est une technologie développée par nVidia pour leurs cartes graphiques GeForce series 8 et 9, c'est à dire qu'on utilise un processeur graphique (GPU) pour exécuter des calculs habituellement exécutés par le processeur central (CPU).

b) Avantages

CUDA est beaucoup utilisé dans la recherche scientifique, notamment car il utilise un CPU simple-core et un GPU. Ainsi, cela permet d'exécuter le code séquentiel sur le CPU et le code parallèle sur le GPU. De plus c'est un niveau de langage intermédiaire.

c) Inconvénients

Le principal inconvénient est que cette technologie nécessite un GPU nVidia et qui est basée sur le langage C (pas de programmation orientée objet).

d) Bilan

Une technologie intéressante pour sa rapidité, mais qui nécessite une formation de notre part et probablement des achats clients pour le GPU.

Analyse SWOT :

Strength	Weaknesses
- **Langage par threads très puissant (permet d'exploiter au maximum l'architecture massivement parallèle des GPU)**	- Contrainte du GPU nVidia
Opportunity	**Threats**
- **Probablement une technologie d'avenir pour la recherche**	- Mode émulation existant, mais extrêmement lent

2. Etude de la technologie C++ multithread

a) Description

Le C++ multithread est une technologie très utile pour tout ce qui est vitesse de calcul avec des vecteurs. Le travail se réalise essentiellement sur le CPU mais n'est utile que si celui-ci possède plusieurs processeurs (cœurs). En effet le multithread permet d'utiliser au maximum les capacités d'un processeur multi-cœurs en effectuant du vrai parallélisme.

b) Avantages

Le principal avantage est l'utilisation en parallèle des différents cœurs du CPU. Notons aussi que le C++ est très connu et populaire ce qui signifie qu'il existe beaucoup de documentations

et bibliothèques sur le multithread. Le C++ est une matière enseignée à l'ESEO et nous avons déjà les bases de la programmation. Le temps de formation sur ce langage sera donc assez faible.

Enfin le fait de pouvoir utiliser la programmation orienté objet (POO) facilite grandement la tâche du programmeur pour l'organisation de son travail, la propreté du code etc.

c) Inconvénients

Un défaut existe cependant dans l'utilisation du C++ multithread. Ce type de programmation n'utilise pas le GPU, or la programmation d'un automate cellulaire est bien adaptée à l'utilisation du GPU car celui-ci est très puissant pour le traitement parallèle.

Finalement le C++ peut aussi être parfois difficile à programmer. En effet il faut tenir compte de la synchronisation, des dreadlocks, des échanges de données, de sémaphore.

d) Bilan

Une technologie moins rapide que d'autres existantes (tel que CUDA) mais plus facile de prise en main. Cependant un algorithme complexe mais qui n'a aucun frais client.

Analyse SWOT :

Strength	**Weaknesses**
- **CPU multi-cœurs** - **Grand nombre de bibliothèques** - **Pas de temps de formation** - **Utilisation de la POO**	- Pas d'utilisation du GPU - Programmation multithread complexe
Opportunity	**Threats**
- **Technologie multiprocesseur en expansion**	

3. Etude de la Toolbox « Parallel Computing Toolbox » de Matlab

a) Description

Il nous est demandé de développer un programme rapide et efficace. MATLAB permet une utilisation simple des matrices ce qui est particulièrement pertinent pour notre projet.

Nous allons maintenant envisager une solution qui nous permettrait de directement optimiser le code MATLAB. La Toolbox PCT permet de tirer parti des ordinateurs multi processeurs et des cartes graphiques afin d'accélérer les simulations numériques qui nécessitent beaucoup de temps de calcul.

b) Avantages

Le principal avantage de cette technologie est l'utilisation conjointe de tous les processeurs de l'ordinateur et de la carte graphique, cette toolbox est basée sur le langage CUDA, l'utilisateur doit donc posséder une carte graphique nVidia. Un avantage non négligeable est que cette toolbox permet d'exécuter des programmes Matlab sur le GPU avec un minimum de modifications.

Cette technologie est très haut niveau et ne nécessite pas de formation particulière de notre part.

Si la ToolBox est présente, on bénéficie :

- De nouveaux mots-clés Matlab permettant de paralléliser nos programmes (boucles for parallèles …)
- De nombreux produits Mathworks déjà parallélisés

De plus, si la ToolBox « MATLAB Distributed Computing Server » est aussi présente, il est possible de déployer les programmes sur des clusters d'ordinateurs (ordinateurs en réseaux) afin de profiter de la puissance de tous les ordinateurs.

c) Inconvénients

Le principal inconvénient est le prix de la licence qui semble être relativement élevé. De plus, cette licence n'est pas présente à l'ESEO et devra donc être achetée en cas de choix de cette technologie.

Nous n'avons pas trouvé le prix exact car pour l'obtenir il est nécessaire de prendre contact avec un commercial Mathworks. Cependant, nous avons trouvé une estimation du prix sur un forum (source non officielle) : environ 1000$ la licence. De plus, il existe très peu de documentation et de littérature sur cette technologie.

Le dernier inconvénient est qu'il n'est pas possible de choisir la répartition du programme entre le CPU et le GPU (la répartition est choisie par Matlab). Et il peut être intéressant de pouvoir choisir nous-mêmes cette répartition afin d'optimiser au maximum le programme.

d) Bilan

Cette technologie semble très intéressante et fournit des programmes très rapides mais le prix de la licence sera probablement rédhibitoire.

Analyse SWOT :

Strength	**Weaknesses**
- **Technologie très rapide (si carte graphique nVidia)** - **Très haut niveau** - **Pas de temps de formation**	- Prix de la licence - Peu de littérature - Pas de choix de répartition CPU/GPU
Opportunity	**Threats**
- **Toolbox qui pourra être utilisée pour tous les autres programmes** - **Existe une toolbox équivalente gratuite (mais nécessite techno nVidia)**	

Pour conclure sur cette partie, nous trouvons que cette solution peut être vraiment intéressante. Le prix nous semble cependant être un véritable frein. Une étude des différentes solutions proposées nous a amené à nous intéresser à la toolbox GPUmat.

4. Etude de la Toolbox GPUmat

a) Description

Comme nous l'avons vu dans la partie précédente MATLAB est une solution intéressante pour notre projet. L'utilisation d'une toolbox peut permettre d'optimiser notre temps de calcul. GPUmat est une toolbox développée par nVidia, elle est développée en CUDA et elle permet d'utiliser la carte graphique de l'ordinateur ainsi que les différents processeurs.

b) Avantages

Tout comme la computing parallel toolbox, le principal avantage de cette technologie est l'utilisation conjointe de tous les processeurs de l'ordinateur et de la carte graphique (si c'est une nVidia compatible). Pour optimiser le temps de calcul et utiliser le GPU il suffit parfois juste de rajouter un mot clé. Cette technologie est très haut niveau et ne nécessite pas de formation particulière de notre part.

Si la ToolBox est présente, on bénéficie :

- De nouveaux mots-clés MATLAB permettant de paralléliser nos programmes (boucles for parallèles …)
- De nombreuses fonctions de bases codées avec GPUmat (Exponentielle, logarithme …)

Un avantage supplémentaire est la gratuité de cette toolbox.

c) Inconvénients

Malgré la documentation fournie par nVidia, il existe encore très peu de documentation et de littérature sur cette technologie.

d) Bilan

Cette technologie peut être un bon compromis. Elle permettra tout du moins de gagner du temps lors de l'exécution du programme MATLAB que nous allons développer. Nous comparerons ensuite ces performances avec le langage choisi.

Analyse SWOT :

Strength	**Weaknesses**
- **Technologie très rapide (si carte graphique nVidia)** - **Très haut niveau** - **Pas de temps de formation**	- Peu de littérature
Opportunity	**Threats**
- **Toolbox qui pourra être utilisée pour tous les autres programmes et par tous les développeurs MATLAB.**	

5. Etude de la technologie OpenCL

a) Description

OpenCL est une bibliothèque libre qui permet de tirer parti des ordinateurs multi processeurs et de toutes les cartes graphiques pour accélérer les calculs numériques. Cette technologie est relativement récente puisqu'elle date de 2009.

b) Avantages

Le principal avantage de cette technologie est sa portabilité. En effet, elle permet d'utiliser tous les CPU et tous les GPU (quel que soit leur marque). C'est donc une technologie très rapide.

Un autre avantage est qu'il est possible de choisir la répartition du programme entre le CPU et le GPU (code séquentiel sur CPU et code parallèle sur GPU par exemple).

c) Inconvénients

Le principal inconvénient est que c'est une technologie récente et qu'il existe très peu de littérature, peu d'outils intégrés permettant de développer et pas de débugger efficace.

Cette technologie est très bas niveau et inconnue de tous les membres de l'équipe. Si cette technologie est retenue, elle nécessitera une importante formation pour l'équipe de développement.

Pour terminer, cette bibliothèque se base sur le langage C et il ne sera donc pas possible d'utiliser la programmation orientée objet dans nos algorithmes.

d) Bilan

Cette technologie est intéressante pour sa portabilité mais qui nécessitera un temps de formation important. Toutefois, elle ne nécessite aucun achat, ni pour l'équipe de développement et ni pour le client.

Analyse SWOT :

Strength	**Weaknesses**
- **Technologie très rapide** - **Choix de répartition CPU/GPU**	- Très bas niveau - Beaucoup de formation - Pas de littérature - Pas d'IDE, de débugger …
Opportunity	**Threats**
	- Technologie récente, va-t-elle perdurer (sachant que CUDA semble plus utilisé) ?

6. Récapitulatif

Ci-dessous un récapitulatif montrant le classement des différents langages étudiés. (Le 1 étant la meilleure solution tandis que le 4 correspond à la moins pertinente)

	Rapidité	**Portabilité**	**Faisabilité**	**Prix**
1.	Matlab //	OpenCL	Matlab //	OpenCL
2.	CUDA	Matlab //	CUDA	C++ mt
3.	OpenCL	C++ mt	C++ mt	Cuda
4.	C++ mt	CUDA	OpenCL	Matlab //

B. Choix et orientations

Après de nombreuses recherches et une réunion au sein de l'INRA à Rennes notre choix s'est porté sur le C++ multithread pour sa portabilité vis à vis du matériel fourni.

En effet si le choix s'était porté sur le CUDA, (un des langages les plus rapides pour réaliser un automate cellulaire) nous aurions eu dans un premier temps un problème de portabilité (CUDA ne fonctionne qu'avec des cartes nVidia) puis un temps de prise en main de cette technologie nouvelle.

De même la technologie Matlab parallèle n'a pas été sélectionnée car son prix d'achat était trop élevé pour nous. Nous utiliserons cependant GPUmat afin de voir si ses performances peuvent être utiles à notre projet.

1. Choix de GPUmat

La PCT de Matlab nous parait une bonne solution, rapide et simple d'utilisation c'est un bon compromis. Cependant le prix de cette toolbox est une limite importante, nous avons donc décidé d'étudier la librairie GPUmat.

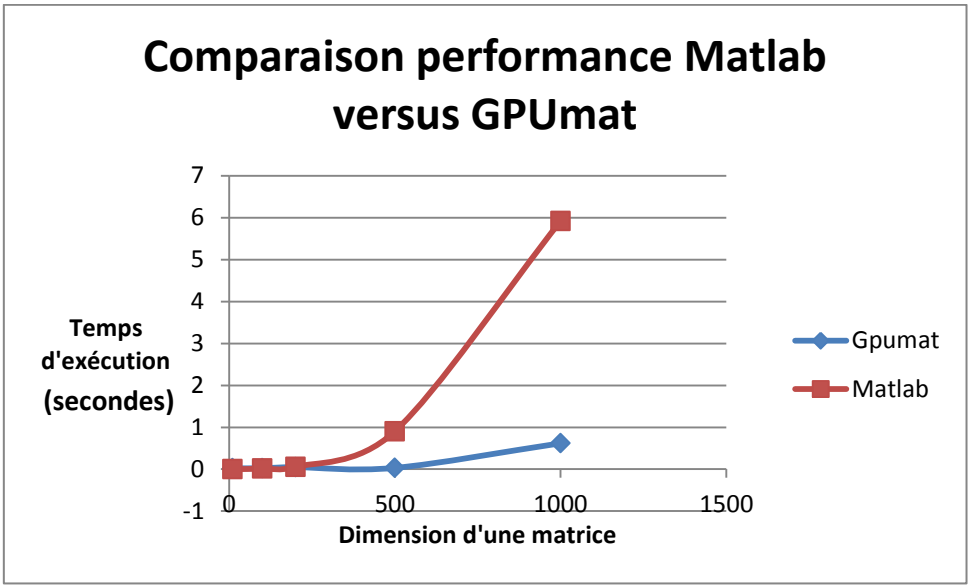
Afin de comprendre quelles sont les conditions optimales pour l'utilisation de GPUmat, nous allons comparer un programme MATLAB et un programme GPUmat, ce programme effectuera 100 multiplications matricielles pour des matrices de tailles différentes.

Les codes exécutés sont les suivants :

```
Tic
      dim = 100
A = ones(dim, dim);
B = ones(dim, dim);
 for i=1:100
    C = A * B;
 end
toc
```

```
Tic
dim = 100
Agpu = ones(dim,dim,GPUsingle);
Bgpu = ones(dim,dim,GPUsingle);
for i=1:100
    C = Agpu * Bgpu;
end
toc
```

Les résultats obtenus sont représentés sur le graphique suivant



A partir du moment où l'on souhaite étudier des matrices dont les dimensions sont supérieures à 500*500, GPU semble être beaucoup plus performant. Dans notre cas nous souhaitons étudier des matrices de taille 1000*1000, le choix de ce langage parait donc tout à fait approprié pour optimiser le temps d'exécution du programme Matlab.

2. La programmation C++ Multithread

Le multithreading est une technologie complexe mais qui a un avantage, augmenter la vitesse de calcul en optimisant le travail des processeurs disponibles sur un ordinateur.

Pour notre projet nous avons décidé de travailler avec la bibliothèque Boost (cf. Guide d'installation librairie Boost) sous Microsoft Visual Studio 2008 (cf. Guide d'installation Visual Studio 2008).

a) Le choix du nombre de thread

Le multithreading comme son nom l'indique consiste en la création de plusieurs threads (tâches) pour effectuer des calculs en parallèles sur les différents processeurs de l'ordinateur utilisé.

Pour pouvoir choisir quel est le nombre optimal de threads que nous devons utiliser, nous avons réalisé un programme test qui compare la vitesse d'exécution d'un programme C++ et d'un programme C++ multithread. La partie multithread est celle que nous avons cherché à optimiser.

Pour cela nous avons créé un algorithme qui travaille sur les produits matriciels. Le programme C++ effectue 16 produits matriciels de façon linéaire tandis que le C++ multithread effectue ces 16 produits en parallèles grâce à l'utilisation de groupes de threads.

b) Résultats des tests expérimentaux

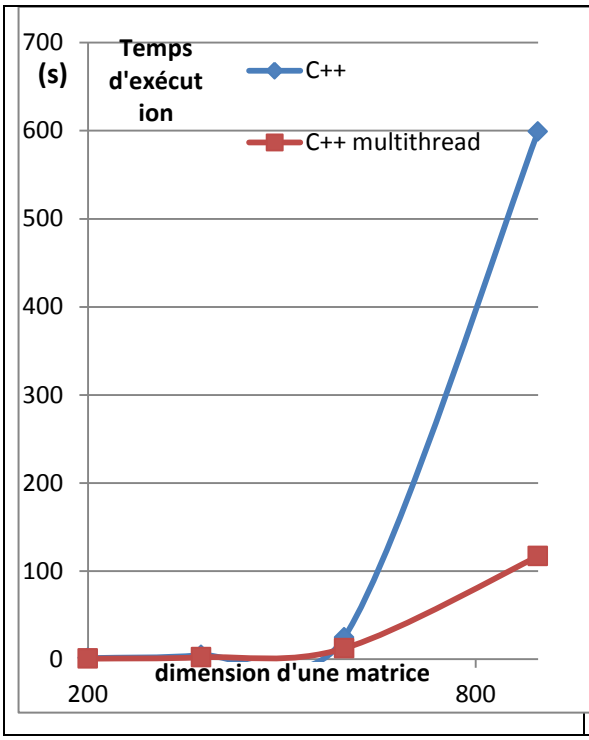
Nous avons effectué les comparaisons suivantes avec des groupes de threads de tailles différentes (16, 8, 4 et 2 threads) avec un programme codé en C++.

Sur une question purement théorique il faudrait utiliser autant de threads que de cœurs (processeurs) disponibles.

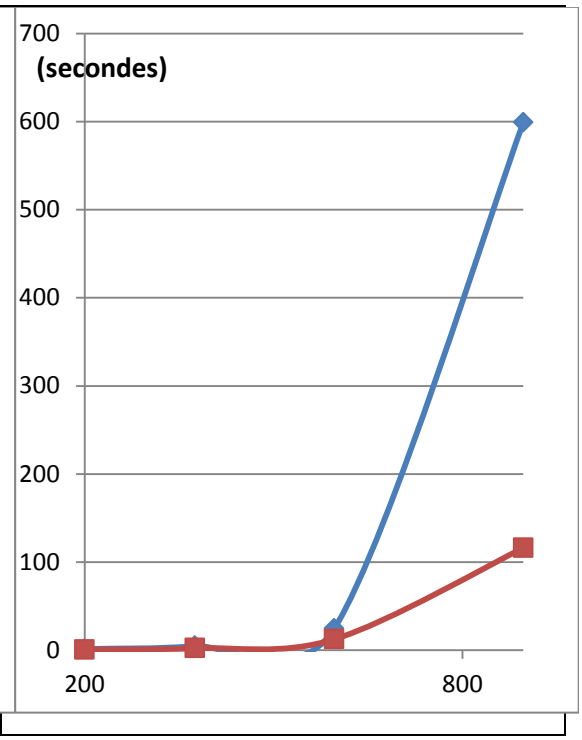
L'ordinateur qui est utilisé pour réaliser ces différents tests possède 4 cœurs. Donc d'un point de vue théorique il faut utiliser 4 threads pour obtenir les meilleurs résultats possibles.

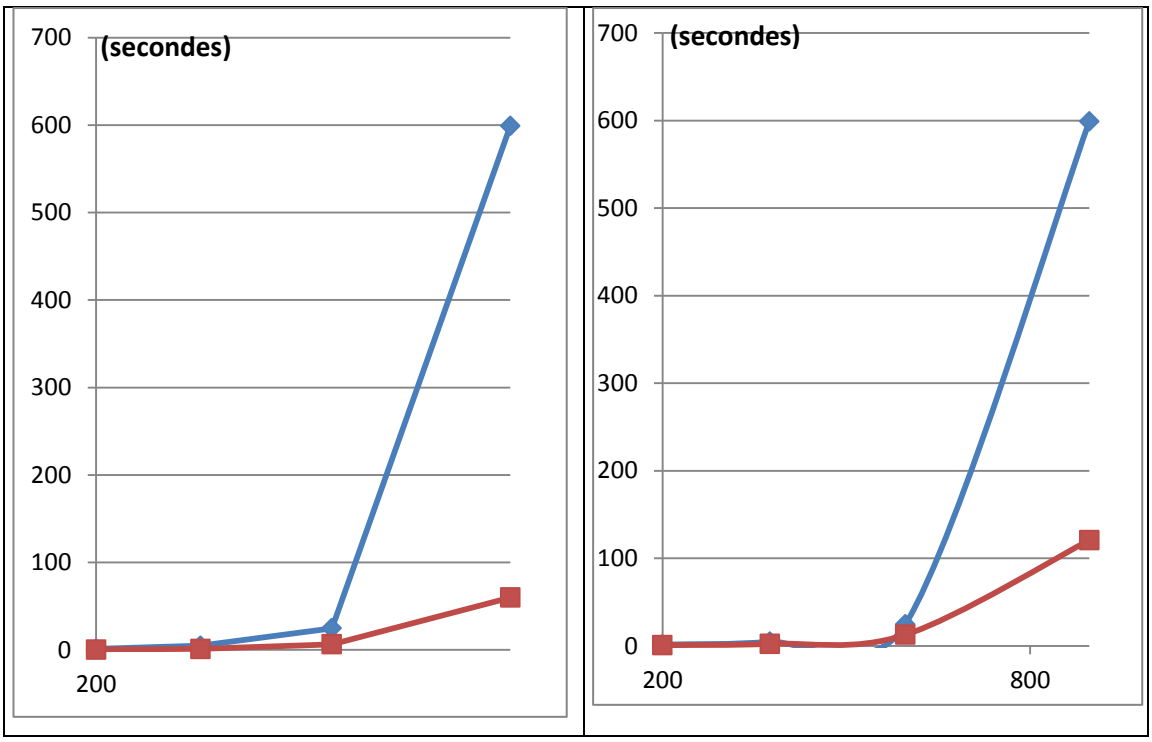
Voici les résultats obtenus :

16 threads



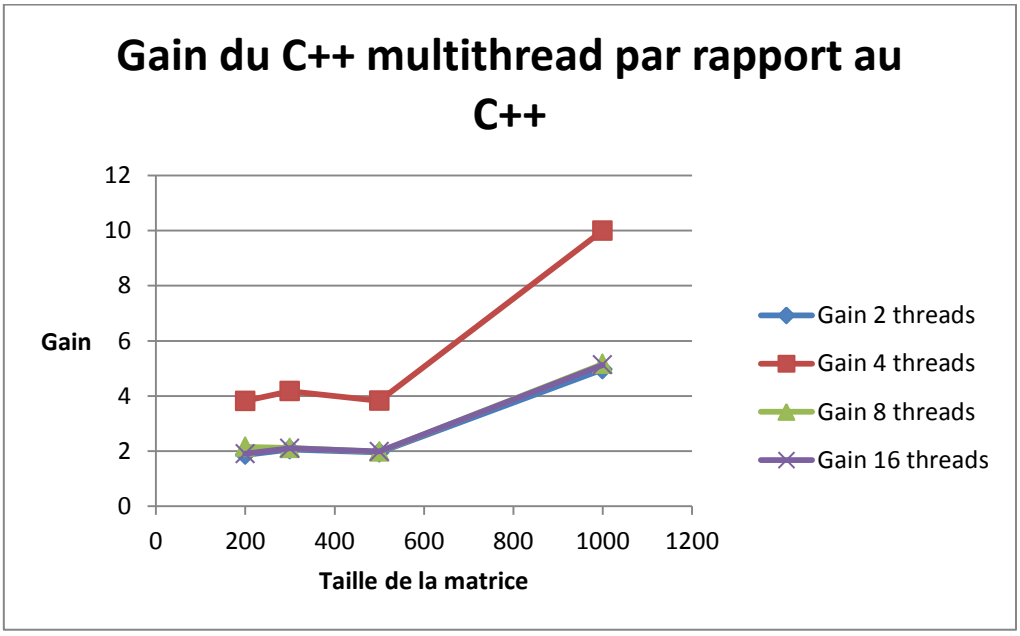
8 threads





4 threads 2 threads

On remarque que le C++ multithread est vraiment beaucoup plus rapide que le C++ normal. Et que plus la taille des matrices est importante plus le gain de temps l'est aussi. Le graphique ci-dessous compare les différents gains.



On remarque aussi que le choix optimal est bien de prendre autant de threads que de cœurs disponibles. C'est-à-dire dans notre cas, quatre threads pour quatre processeurs.

Cependant nous noterons aussi que le multithreading n'est utile que pour certain type de calcul. En effet créer beaucoup de threads pour peu de calculs engendre une grosse perte de temps d'exécution. Par contre utiliser peu de threads (le nombre threads adapté à la machine utilisée) avec beaucoup de calculs à l'intérieur permet d'optimiser au maximum ce temps d'exécution.

C. Automate cellulaire en Matlab

Matlab est un langage haut niveau permettant l'exécution de tâches nécessitant une grande puissance de calcul et dont la mise en œuvre est plus rapide qu'avec des langages de programmation traditionnels tels que le C/C++. Nous décrirons et expliquerons dans un premier temps le fonctionnement général de l'automate codé sous Matlab. Puis nous aborderons les points délicats de l'implémentation à savoir, la manière dont le voisinage Moore et Von Neumann a été implémenté, comment étendre cet algorithme à un voisinage de Moore étendu, la taille des matrices de probabilités secondaires (rsx, rsy, rsz). Enfin, nous expliquerons de quelle manière, nous avons validé l'implémentation et nous montrerons quelques tests du programme.

1. Description et explication du fonctionnement général de l'automate cellulaire codé uniquement en Matlab.

L'automate cellulaire développé en Matlab est de type SIDR. Ce programme a été optimisé pour Matlab c'est-à-dire qu'il utilise au maximum des opérations sur des matrices. Ainsi le programme utilise uniquement deux boucles : une pour les réplicas et l'autre pour le temps de la simulation.

Pour ne pas parcourir l'ensemble des cellules de la grille et gagner en temps de calcul, nous utilisons des masques. Ces masques sont des matrices de 1 et de 0 de la taille de la grille permettant de sélectionner les éléments sains, infectés, détectés ou retirés de la grille. Ces matrices sont nommées : Matrice_element_infecte, Matrice_element_detecte, Matrice_element_retire.

En sortie de l'automate, nous utilisons 3 Matrice 3d : Matrice_temps_infecte, Matrice_temps_detecte, Matrice_temps_retire. Ces matrices représentent le temps depuis lequel une cellule est infectée, détectée ou retirée, suivant sa position dans la grille et le numéro de réplica. L'avantage est qu'à partir de ces trois matrices nous pouvons reconstruire l'ensemble de la simulation sans avoir à stocker tous les pas de temps de la simulation. Cependant, chaque matrice s'incrémente pendant toute la durée de la simulation, dès lors que la cellule est passée dans l'état infecté, détecté ou retiré. Ainsi pour une cellule retirée, sa valeur correspondante dans Matrice_temps_infecte s'incrémente toujours durant la simulation. Ces matrices sont initialisées par l'utilisateur grâce aux paramètres d'entrée.

Le programme se décompose en plusieurs fichiers. Le fichier Run_me.m permet d'initialiser et de lancer l'automate. Cependant, l'utilisateur peut souhaiter modifier les fonctions de transitions. Il a la possibilité de modifier les fichiers : prob_primaire.m, prob_secondaire.m, prob_IversR.m, prob_IversD.m, prob_DversR.m. Ces fichiers sont décrits dans la partie III.A, Lancement et utilisation des programmes.

2. Voisinage implémenté

La question du voisinage (Moore ou Von Neumann) se pose uniquement lors de l'infection secondaire. Ces deux voisinages ont été implémentés mais peuvent être délicats à comprendre sous Matlab. Nous détaillerons donc l'algorithme utilisé et proposerons une méthode pour étendre celui-ci à un cas de Moore étendu.

Afin d'optimiser Matlab, nous avons choisi d'effectuer un maximum de calculs grâce à des matrices. Ainsi pour calculer la probabilité d'infection secondaire, sous Matlab, il n'est pas intéressant de parcourir la grille et calculer les probabilités que chaque élément infecte son voisin. En effet, les probabilités sont initialement calculées dans les matrices 3d rsx, rsy et rsz.

Dans un premier temps nous récupérons la probabilité que chaque élément infecte son voisin, pour toute la grille en une seule commande :

```
matrice_rsx = rsx(t,Matrice_temps_infecte_sans_retire+1);
```

Le '+1' est nécessaire car le premier élément d'un vecteur sous Matlab est à la position 1 et non 0. Sans cet ajout, Matlab cherche à récupérer une valeur de rsx à la position (0,0) qui revoit une erreur.

Cette commande renvoie un vecteur qu'il faut transformer en matrice de dimensions identiques à celles de la grille. On doit ensuite prendre la probabilité complémentaire :

```
matrice_rsx = reshape(matrice_rsx',width_grid,length_grid); % on remet le vecteur sous forme d'une matrice
matrice_rsx = 1-matrice_rsx; % On calcul la probabilité complémentaire
```

Nous obtenons ainsi la probabilité qu'un élément de la grille n'infecte pas son voisin suivant les colonnes (en x). Voici un exemple de résultat à ce stade

Position :

1	2	3
4	5	6
7	8	9

Probabilité :

0.1	0.2	0.3
0.4	0.5	0.6
0.7	0.8	0.9

L'objectif est d'obtenir la probabilité qu'un élément de la grille ne devienne pas infecté (probabilité complémentaire) suivant les colonnes. Avec l'exemple précédent, si nous souhaitons connaitre la probabilité que l'élément à la position 5 ne devienne pas infecté suivant les colonnes, nous devons multiplier la probabilité située à la position 4 par celle de la position 6.

La solution retenue, sous Matlab, est de créer une matrice_rsx2 dont ses colonnes sont toutes décalées vers la gauche de deux positions. Nous devons donc rajouter 2 colonnes afin d'avoir le même nombre que la grille. L'avant-dernière colonne est mise à 1 et la dernière à 0.

```
matrice_rsx2 = [matrice_rsx(:,3:end) ones(width_grid,1) zeros(width_grid,1)];
```

Enfin nous multiplions les deux matrices et positionnons les résultats à la bonne position de la grille :

```
matrice_rsx2 = matrice_rsx2.*matrice_rsx;
matrice_rsx = [matrice_rsx(:,2) matrice_rsx2(:,1:end-1)];
```

Cet algorithme est ensuite réalisé de la même manière pour avoir les probabilités d'infections secondaires suivant les lignes (en y) et un voisinage de Moore(en z). Concernant les lignes, le décalage s'effectue de deux positions vers le haut de la matrice. Si nous sommes dans un voisinage de Moore, nous effectuons exactement le même l'algorithme suivant les colonnes (en x) puis, avec le résultat obtenu, nous l'effectuons suivant les lignes (en y). Les 3 matrices (matrice_rsx, matrice_rsy, matrice_rsz) sont ensuite multipliées entre elles et avec la probabilité d'infection primaire pour avoir la probabilité qu'un élément sain ne devienne pas infecté.

3. Evolution pour un voisinage de Moore étendu

Afin de calculer la probabilité d'infection secondaire avec un voisinage de Moore étendu, de nouvelles fonctions doivent être créées (matrice_rs2x, matrice_rs2y, matrice_rs2z_1, matrice_rs2z_2, matrice_rs2z_3).

Voici un exemple de résultat à ce stade

Position

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

Probabilité

0.01	0.02	0.03	0.04	0.05
0.06	0.07	0.08	0.09	0.1
0.11	0.12	0.13	0.14	0.15
0.16	0.17	0.18	0.19	0.20
0.21	0.22	0.23	0.24	.025

L'objectif est de calculer la probabilité que l'élément 13 ne devienne pas infecté. La première partie reste identique au voisinage de Moore simple, sans besoin d'utiliser des fonctions supplémentaires.

Pour passer à l'étape étendue, les nouvelles fonctions doivent calculer la probabilité qu'un élément puisse infecter son 2ème voisin :

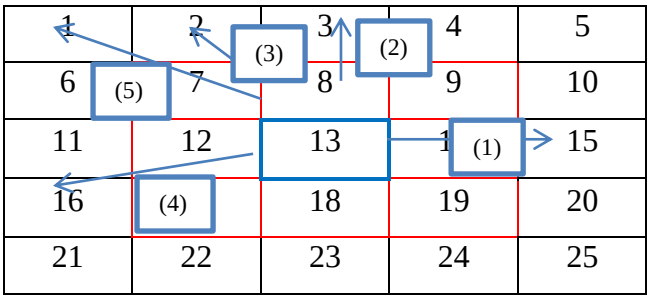
rs2x : probabilité d'infecter un voisin à une distance de 2x -> (1) sur le schéma ci-dessous

rs2y: probabilité d'infecter un voisin à une distance de 2y -> (2)

rs2z_1 : probabilité d'infecter un voisin à une distance de 1x +2y -> (3)

rs2z_2 : probabilité d'infecter un voisin à une distance de 2x + 1y -> (4)

rs2z_3 : probabilité d'infecter un voisin à une distance de 2x + 2y -> (5)



Puis des opérations de décalages et de multiplications de matrices doivent être effectuées :

- On calcule les probabilités de ne pas infecter un voisin à une distance de 2x + 2y dans la matrice matrice_ rs2z_3

  On décale la matrice de quatre colonnes vers la gauche (les positions 5,10,15,20,25 deviennent 1,6,11,16,21). Les deux dernières colonnes sont mises à zéros et les 3ème et 4ème dernières, à 1 pour avoir une matrice de dimensions égales à la grille. Puis on multiplie la nouvelle matrice par la matrice initiale.

  On récupère les 3ème et 4ème colonnes de la matrice initiale que l'on positionne en 1ère et 2ème position de la matrice intermédiaire. On complète les colonnes de cette matrice intermédiaire avec les valeurs issues de la multiplication. Les deux dernières colonnes de

la matrice issue de la multiplication ne sont pas intégrées dans la matrice intermédiaire pour avoir les mêmes dimensions que la grille.

On décale la matrice intermédiaire obtenue de quatre lignes vers le haut (les positions 21,22,23,24,25 deviennent 1,2,3,4,5). Les deux dernières colonnes sont mises à zéros et les 3ème et 4ème dernières, à 1 pour avoir une matrice de dimensions égales à la grille. Puis on multiplie la nouvelle matrice par la matrice intermédiaire.

On récupère les 3ème et 4ème colonnes de la matrice intermédiaire que l'on positionne en 1ère et 2ème position de la matrice finale. On complète les colonnes de cette matrice finale avec les valeurs issues de la dernière multiplication. Les deux dernières colonnes de la matrice issue de la dernière multiplication ne sont pas intégrées dans la matrice intermédiaire pour avoir les mêmes dimensions que la grille.

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

- On effectue exactement la même opération pour calculer les autres probabilités d'infecter un voisin. Cependant il faut faire attention à décaler correctement les matrices afin d'effectuer les bonnes multiplications.

4. Taille des matrices rsx, rsy et rsz dans le calcul de probabilité secondaire

Un point qui peut paraitre délicat pour l'utilisateur est la taille des matrices rsx, rsy et rsz représentant la probabilité qu'une cellule infecte son voisin suivant les colonnes (en x), les lignes (en y) et les diagonales (en z). Les lignes de ces matrices représentent la durée de la

simulation et les colonnes, la durée depuis laquelle une cellule est infectée. Or les lignes ont une dimension égale au temps de la simulation et les colonnes, à la durée de la simulation ajoutée au temps maximal d'infection initiale de la grille. De plus la première colonne est la probabilité qu'une plante non infectée infecte son voisin (à priori égale à zéro). Ainsi nous ne prenons pas en compte la durée maximale qu'une plante est infectée puisque pour calculer la probabilité qu'une plante infecte un autre, nous avons besoins de sa durée d'infection précédente.

Exemple :

Prenons un exemple de durée de simulation de 10 jours avec une initialisation de la grille telle que la cellule la plus infectée initialement soit de 10 jours. Les matrices de transitions rsx, rsy, rsz définies actuellement sont de la forme :

		Durée depuis laquelle une cellule est infectée							
		0	1	2	…	10	11	…	19
Temps de la simulation	1	0	0.5	0.5	…	0.5	0.5	…	0.5
	2	0	0.5	0.5	…	0.5	0.5	…	0.5
	…	…	…	…	…	0.5	0.5	…	0.5
	10	0	0.5	0.5	…	0.5	0.5	…	0.5

Deux cas sont possibles :

- Une cellule n'est pas infectée initialement
- Une cellule est infectée depuis un certain temps initialement

Déroulons la simulation suivant ces deux cas :

- 1er cas : la cellule n'est pas infectée initialement

Au 1er jour de simulation, t=1 : la cellule n'est pas infectée. On récupère donc la probabilité que cette cellule infecte son voisin, dans la matrice à la position (1,1).

Supposons que la cellule soit infectée au 2ème jour de simulation, t=2. Pour calculer la probabilité que cette cellule infecte son voisin, on récupère la probabilité dans la matrice à la position (2,2). Cette position correspond à une durée d'infection égale à 1 jour.

Au dernier jour de simulation t=10 : supposons que la cellule soit toujours infectée. Pour calculer la probabilité que cette cellule infecte son voisin, on récupère la probabilité dans la matrice à la position (10,10). Cette position correspond à une durée d'infection égale à 9 jours.

- 2er cas : On initialise une cellule, à un état infecté depuis 10jours

Au 1er jour de simulation, t=1 : la cellule est infectée depuis 10 jours. On récupère donc la probabilité que cette cellule infecte son voisin, dans la matrice à la position (1,11). Cette position correspond à une durée d'infection égale à 10 jours.

Au 2ème jour de simulation, t=2 : la cellule est infectée depuis 11 jours. On récupère donc la probabilité que cette cellule infecte son voisin, dans la matrice à la position (2,12). Cette position correspond à une durée d'infection égale à 11 jours.

Au dernier jour de simulation t=10 : la cellule est infectée depuis 19 jours. Pour calculer la probabilité que cette cellule infecte son voisin, on récupère la probabilité dans la matrice à la position (10,20). Cette position correspond à une durée d'infection égale à 19 jours.

Au final, les matrices de transitions rsx, rsy, rsz, ont bien des dimensions de 10 lignes et 20 colonnes.

5. Validation

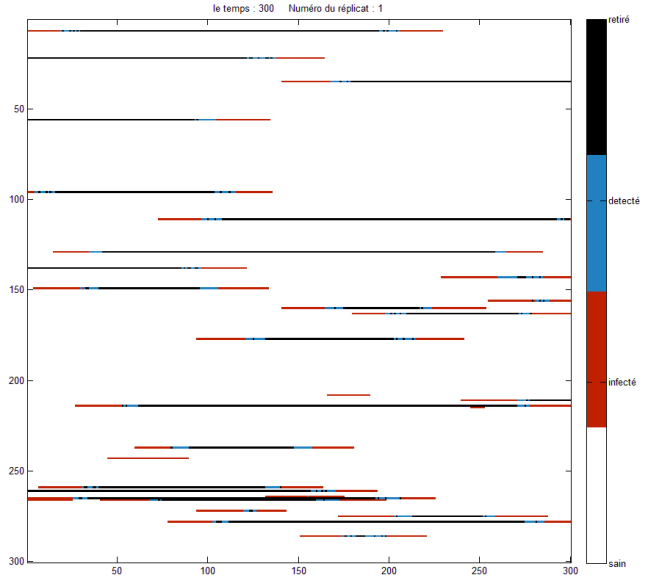
Afin de valider le code Matlab, nous avons fixé toutes les probabilités à des constantes égales ou non à zéros ainsi que les pas de temps avant détection et la transition vers l'état retiré. Nous avons ensuite vérifié que chaque ligne de code fournissait les valeurs exactes de probabilités, d'incrémentations ou de masquage. Enfin, nous avons vérifié que chaque résultat de changement d'état était en accord avec les tirages aléatoires.

Puis nous avons fait varier tous les paramètres d'entrée ainsi que les fonctions de transitions et vérifié que les sorties donnaient le résultat des calculs de l'automate.

6. Test

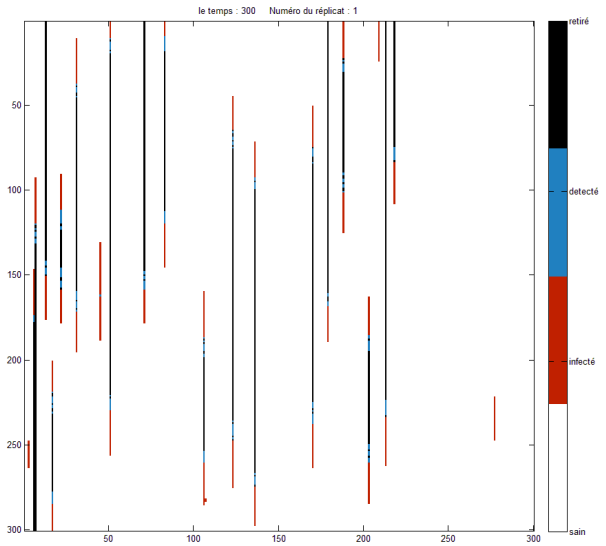
Test 1 :

La grille est de taille 300x300.
Le temps de la simulation = 300j
Infection initiale = aucune
Nombre de voisins = 8 (voisinage de Moore)
Nombre de réplicas = 1 ;
Probabilité primaire = 1 e-6
Probabilité secondaire :
rsx = 0.5 pour toutes les plantes infectées
rsy = 0
rsz = 0



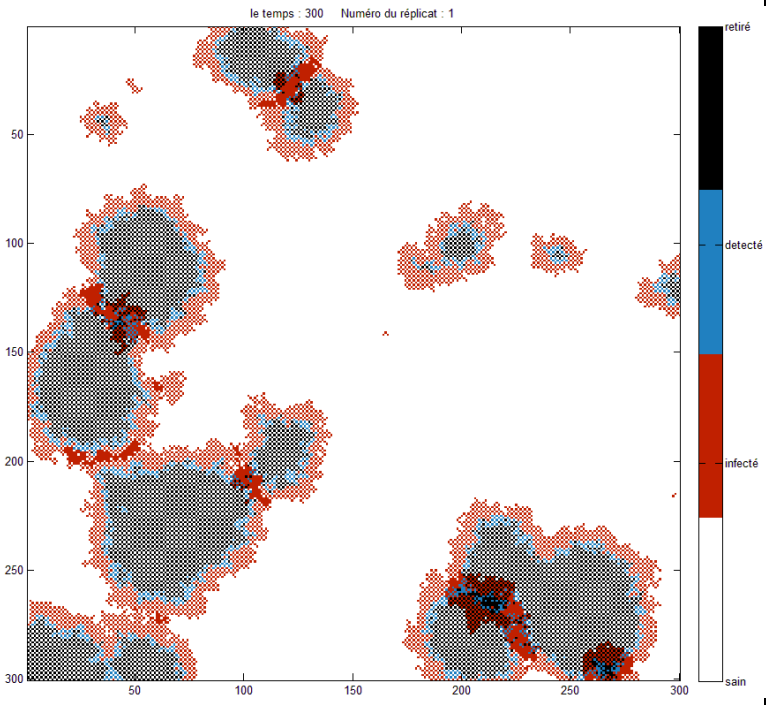
Test 2 :

La grille est de taille 300x300.
Le temps de la simulation = 300j
Infection initiale = aucune
Nombre de voisins = 8 (voisinage de Moore)
Nombre de réplicas = 1 ;
Probabilité primaire = 1 e-6
Probabilité secondaire :
rsx = 0
rsy = 0.5 pour toutes les plantes infectées
rsz = 0



Test 3 :

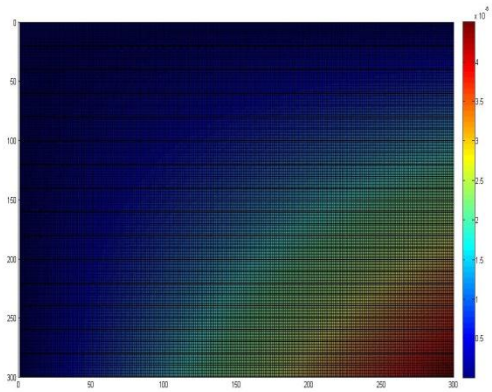
La grille est de taille 300x300.
Le temps de la simulation = 300j
Infection initiale = aucune
Nombre de voisins = 8 (voisinage de Moore)
Nombre de réplicas = 1 ;
Probabilité primaire = 1 e-6
Probabilité secondaire :
rsx = 0
rsy = 0
rsz = 0.5 pour toutes les plantes infectées

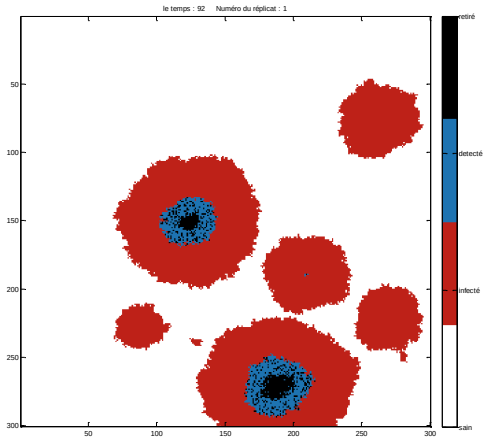


Test 4 :

La grille est de taille 300x300.
Le temps de la simulation = 300j
Infection initiale : aucune
Nombre de voisins = 8 (voisinage de Moore)
Nombre de réplicas = 1 ;
Probabilité primaire
$f(x,y,t) = 5e-5*x*y*1e-6*\exp(-1e-5*t)$
Probabilité secondaire :
rsx = 0.3 pour toutes les plantes infectées
rsy = 0.2 pour toutes les plantes infectées
rsz = 0.1 pour toutes les plantes infectées

1- visualisation de f(x,y,t) au temps 1





D. Le modèle SIR

En épidémiologie, on veut comprendre comment les agents pathogènes se transmettent d'un individu à l'autre afin d'être capable de prédire les épidémies, leur ampleur dans le temps.

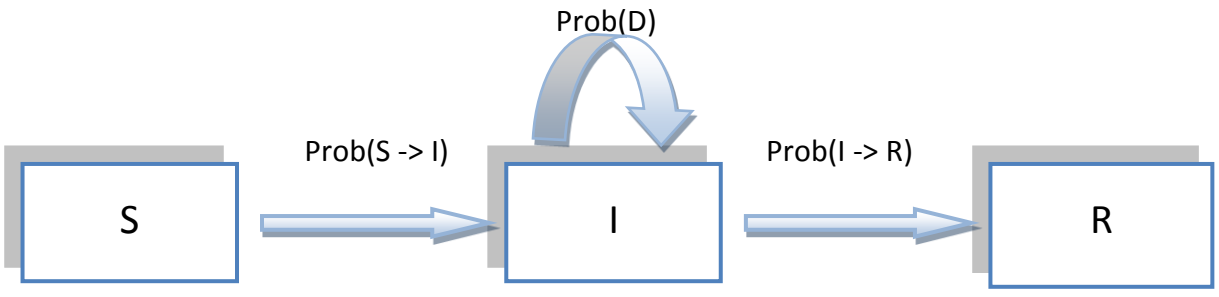
La question est donc comment modéliser un système hôte-parasite ? Le cycle de vie d'un agent pathogène comprend trois étapes : L'agent infecte un individu hôte puis se propage à d'autres individus hôtes jusqu'à tuer les individus.

Les modèles classiques en épidémiologie utilisent des équations différentielles qui mettent en jeu trois types d'individus, classés selon leurs états vis-vis de la maladie considérée (ici le *pathosystème Rhizoctonia solani* sur la Betterave sucrière). Il s'agit du modèle *SIR*, où *S* désigne les individus « sains », *I* les individus « infectés » et *R* les individus « retirés ».

Dans ce modèle nous nous intéressons sur l'état de nos hôtes. Ainsi cette approche nous permet de compartimenter la population (cellules) des individus et d'observer la propagation de l'épidémie au cours du temps et de l'espace.

Pour notre projet, nous avons débuté avec un modèle *SIDR* mais il nous a été demandé de créer un autre modèle, le modèle *SIR*. Dans celui-ci l'état *D* est implicite et devient un attribut de l'état *I*. Nous avons donc reprit les base du modèle *SIDR* implémenté précédemment et supprimé toute ce qui tourne autour de l'état « DETECTE ».

On parle « de Modèle Compartimentaux » représenté comme suit :



Nous avons placé la probabilité *D* comme un attribut de notre programme qui peut être appelé et ainsi définir si la cellule courante est infectée ou infectée_détectée.

Ainsi l'état « DETECTE » n'existe plus, la cellule courante peut rester infectée_detectée ou passer à l'état « RETIRE ».

Les résultats recherchés peuvent être obtenu avec Matlab. Quand une cellule devient infectée, la détection n'est pas obtenue (visuellement) sur le graphique final. Cependant l'utilisateur peut choisir de la visualiser en rajoutant la matrice *Matrice_temps_attribut_detecte*, à la matrice *grid-time*.

Notons aussi que le nombre de cellules infectée_détectée peut être obtenu en regardant dans la matrice de la grille à un temps t pour un réplica donné.

Modèle SIR sous Matlab

Par rapport au modèle SIDR, l'automate du modèle SIR possède les mêmes sorties. Cependant les vecteurs représentant le nombre de paires avec l'état détecté ont été enlevés. En effet, l'état détecté n'est plus un état mais un attribut de l'état infecté.

Le calcul de probabilité pour avoir un attribut détecté sur un individu infecté dépend du temps depuis lequel l'élément est infecté et du temps de la simulation. Cependant il n'y a pas de choix à faire entre les probabilités de passer de l'état I à R et de I à D. De plus la probabilité de passer de D à R est supprimée. Enfin les masquages à l'aide de la matrice Matrice_element_detecte ont été tous supprimés sauf pour le calcul de la probabilité d'avoir l'attribut détecté.

Le nom de la matrice Matrice_element_detecte a été remplacé par Matrice_attribut_detecte et Matrice_temps_detecte par Matrice_temps_attribut_detecte (qui est une sortie de l'automate). Enfin le nom de la fonction proba_IversD.m a été renommée prob_attribut_D.m.

E. Automate cellulaire en GPUmat

L'algorithme de GPUmat est le même que celui utilisé pour, Matlab, nous avons simplement mis les matrices qui interviennent dans la boucle du temps et des réplicas sur la carte Graphique.

La fonction d'appel run_meGPU.m est le même que le fichier run_me.m, à l'exception de l'appel à la fonction automateGPU.

Choix de développement :

- Vous trouverez ci-dessous un tableau récapitulant, l'espace mémoire utilisé par notre programme sur le GPU. Dans le cas de matrice 1000*1000 nous arrivons rapidement à une centaine de méga-octets utilisés, afin que notre programme soit utilisable par un maximum de carte graphique nous avons décidé de ne pas parallélisé la boucle principale car cela décuplerait l'occupation mémoire.
- Pour définir une matrice sur le GPU, il suffit d'ajouter GPUsingle à la fin de la définition d'une « matrice classique ».

Exemple en Matlab :

```
Matrice_temps_infecte = zeros(width_grid,length_grid,nbre_replicat);
```

Exemple en GPUmat :

```
Matrice_temps_infecte =
zeros(width_grid,length_grid,nbre_replicat,GPUsingle);
```

- Limites du code

Le tableau ci-dessous résume la place mémoire nécessaire sur le GPU pour l'exécution du programme. Sur 200 pas de temps.

nbre réplicas /Taille de la grille	**1**	**10**	**20**	**50**	**100**
100 * 100				-	-
250 * 250			-	-	-
100 * 1000		-	<20Mo	22 Mo	23 Mo
1000 * 1000	146 Mo	150 Mo	155 Mo	160 Mo	170 Mo

La carte graphique de notre client est une NVIDIA Quadro NVS 290, sa mémoire est de 256 Mo. Théoriquement vous pouvez facilement utiliser des matrices 1000*1000 avec 100 réplicas.

Les résultats obtenus sont très décevants par rapport à nos attentes. Vous trouverez une explication à ces temps d'exécution plus longs dans la partie III.C.2. Problèmes et limites de GPUMat.

Pour une réelle optimisation avec GPUmat il faudrait complètement réécrire le code et le penser pour GPUmat. Conformément avec le cahier des charges nous avons décidé de consacrer nos efforts sur le C++ multithreadé. Notons finalement que le gros avantage par rapport au C++ est que l'évolution du temps en fonction du nombre de réplicas est linéaire.

F. Développement en C++ multithreadé

Dans cette partie, nous allons décrire le code C++ que nous avons produit dans le cadre de ce projet. Nous allons commencer par détailler les différents composants du programme et l'échange de données entre ces composants. Nous décrirons ensuite de manière plus précise chacun de ces composants et nous terminerons par décrire les façons dont nous avons multithreadé nos algorithmes.

1. Les différents composants du programme

Nous avons choisi de décomposer notre programme en trois principaux composants. Ce choix permet de bien découper l'ensemble du programme en blocs fonctionnels, mais il permet aussi de séparer le code C/C++ en différentes phases de compilation. Ainsi, si on modifie les fonctions de transitions, il ne sera pas utile de recompiler l'ensemble de l'automate cellulaire.

Le premier composant est le **programme Matlab ou R** qui permet de configurer les paramètres de la simulation (taille de la grille, durée de la simulation …) et d'appeler l'automate cellulaire de manière transparente.

Le second est un **programme C** qui ne contient que les fonctions de transitions (modifiables par le client) et qui permet de faire l'interface entre le programme Matlab ou R et l'automate cellulaire. Ce programme permet de générer une librairie dynamique (.dll sous Windows ou .so sous Linux).

Enfin, le dernier composant est un **programme C++** qui contient le code complet de l'automate cellulaire. Ce programme permet de générer une autre librairie dynamique (.dll ou .so).

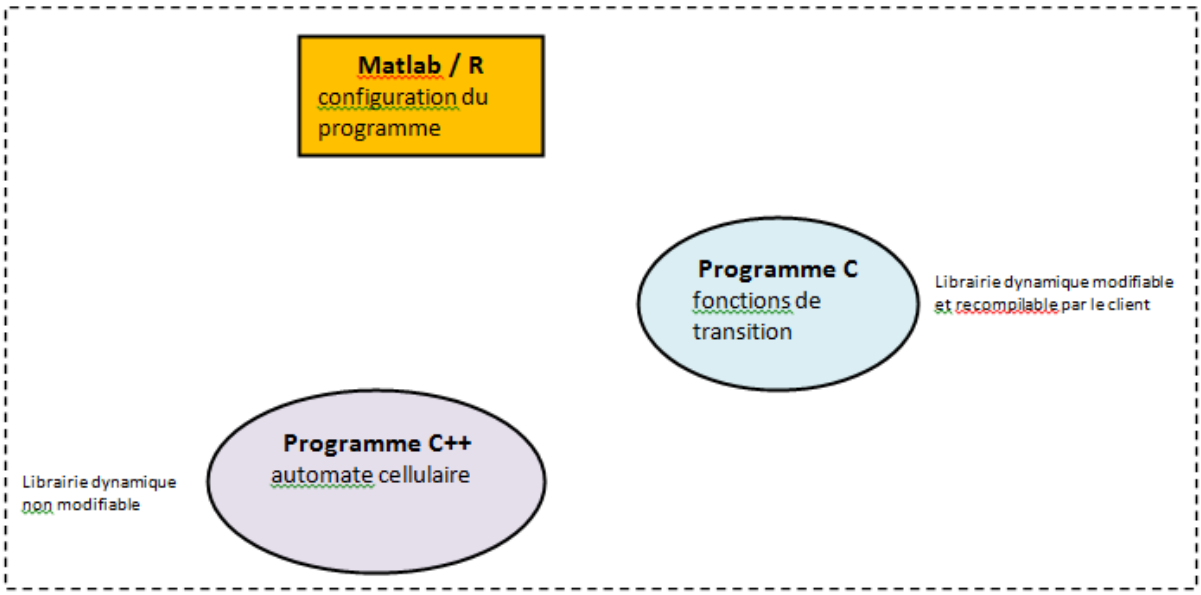


Figure 1 : Décomposition structurelle

2. Les échanges de données entre les composants

Nous allons maintenant décrire les échanges de données et les appels entre les différents composants décris ci-dessus.

Lancement du programme (en vert sur le schéma suivant) :

Premièrement le programme Matlab ou R définit tous les paramètres de configuration du programme. Ensuite, il appelle une première fois le programme C en lui transmettant tous ces paramètres.

Le programme C, quant à lui, définit les fonctions de transitions (S vers I, I vers D …) et appelle ensuite le programme C++ en lui transmettant les fonctions de transition et les paramètres de configuration de Matlab/R.

Le programme C++ déroule toute la simulation. Il ne sort aucun résultat (pas de fichiers) et n'appelle aucun autre composant.

Récupération des résultats (en gris sur le schéma suivant) :

Une fois que la simulation est terminée, il est possible de récupérer les résultats. Pour cela le programme Matlab ou R réappelle le programme C en lui transmettant des pointeurs vers les résultats à remplir.

Le programme C ne fait qu'appeler le programme C++ en lui transmettant ces mêmes pointeurs.

Le programme C++ se charge de remplir ces résultats et comme les passages se sont effectués par pointeurs, les résultats sont automatiquement mis à jour dans Matlab ou R.

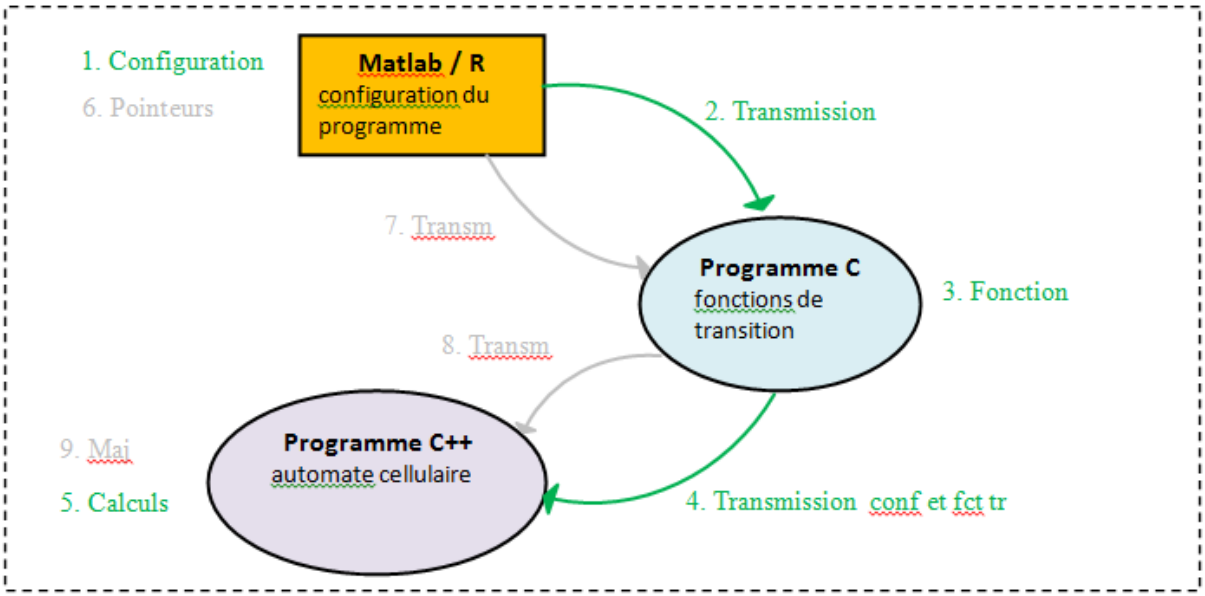


Figure 2 : Les échanges et appels

Afin de pouvoir se retrouver dans les différents fichiers de notre programme, nous allons décrire de façon plus précise chacun des composants.

3. Description des différents programmes

Cette partie sera consacrée à une description plus précise de chacun des programmes.

Le programme Matlab ou R :

Le programme Matlab ou R sera décrit plus en détail dans la partie suivante.

Le programme C, les fonctions de transition :

Le projet C *interface_fonctionTransition* permet, comme son nom l'indique, de définir les fonctions de transitions et de faire l'interface entre le programme Matlab/R et le programme C++.

Pour modifier les fonctions de transitions, il suffit de les modifier dans les fichiers *fonctions_transition.h* et *fonctions_transition.c* et de recompiler le projet afin de générer un fichier .dll (Windows) ou un fichier .so (Linux). Cependant, il n'est pas utile de recompiler l'automate cellulaire (programme C++), c'est tout l'avantage de cette décomposition structurelle. La compilation sera décrite plus en détail dans le paragraphe III.C.

Les fichiers *interface_librairie_dynamique.h* et *interface_librairie_dynamique.c* fournissent l'ensemble des fonctions qui permettent d'interfacer les différents composants. Ces fichiers ne doivent pas être modifiés par l'utilisateur final.

Enfin, les fichiers *point_entree.h* et *point_entree.c* contiennent les deux fonctions appelées par Matlan/R. Ces fichiers ne doivent pas être modifiés par l'utilisateur final.

Le programme C++, l'automate cellulaire :

Le projet C++ *automate_cellulaire* contient la plus grande partie du programme et implémente tout l'automate cellulaire. C'est d'ailleurs ce programme qui sera multithreadé (la description du multithreading sera effectuée dans la partie suivante).

Nous n'allons pas décrire tout le code de ce projet car l'utilisateur final ne devra pas le modifier pour lancer des simulations. Nous allons plutôt décrire la philosophie générale du projet, qui permettra (avec les nombreux commentaires du code) de facilement prendre en main les algorithmes.

Nous avons choisi de baser notre automate sur de la programmation orientée objet en créant trois classes :

- Cellule : cette classe permet de définir complètement une cellule de la grille. On définit notamment son état, sa position… Une cellule connait aussi la grille à laquelle elle appartient.

- Grille : cette classe permet de définir complètement une grille (un réplica). On définit notamment ses dimensions et un tableau contenant toutes les cellules de la grille. Une grille connait aussi la simulation à laquelle elle appartient.

- Simulation : cette classe permet de définir complètement une simulation. On définit notamment les dimensions dx, dy, dz, la durée de simulation, le nombre de réplicas et un tableau contenant toutes les grilles (tous les réplicas).

La figure suivante récapitule les liens entre ces classes

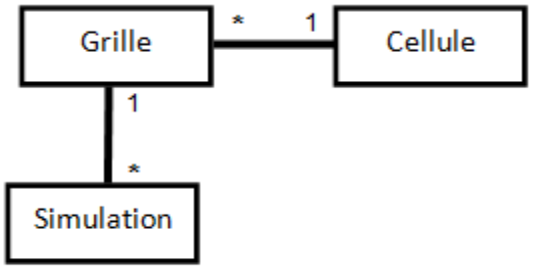


Figure 3 : Les classes de l'automate

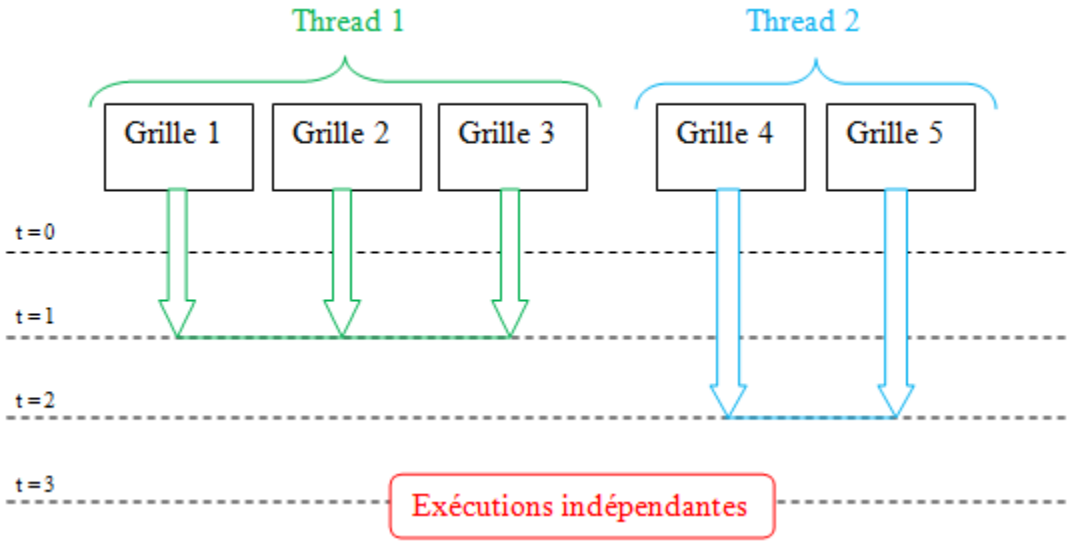
4. Le multithreading

Nous allons maintenant décrire les façons dont nous avons multithreadé nos programmes. Tout d'abord, nous avions retenu deux approches différentes :

- Chaque thread déroule plusieurs réplicas (**PRM** : Plusieurs Réplicas Multithreadés)
- Chaque thread déroule un pas de temps d'une portion de grille (**URM** : Un Réplica Multithreadé)

L'approche PRM :

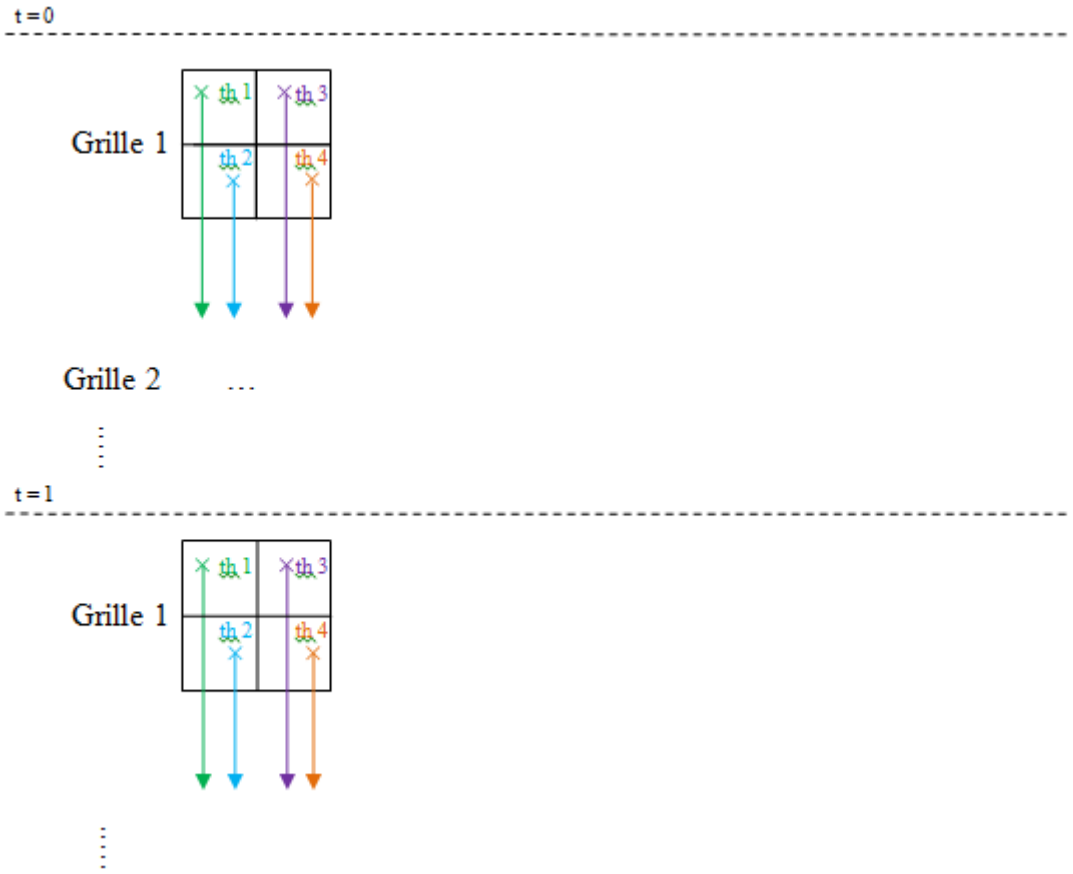
L'approche PRM consiste à répartir les réplicas dans des threads différents. Chaque thread va dérouler l'ensemble des pas de simulation pour les réplicas qui lui sont associés. Tous les threads s'exécutent de façon indépendante : cela signifie qu'à un instant donné, les threads ne sont pas tous rendus au même pas de simulation (cf. figure suivante).



L'approche URM :

L'approche URM est un peu plus complexe. Voici les étapes qui sont effectuées :

- Pour le premier pas de temps donné
- Pour la première grille
- On divise cette grille en N portions, et on affecte chaque portion à un thread
- On lance l'exécution de ces N threads en parallèle
- On attend que ces N threads soient terminés
- On passe à la deuxième grille
- …
- On passe au second pas de temps
- …



Travail effectué :

Nous avions initialement retenu la solution URM (après concertation avec le client) mais finalement nous avons réussi à implémenter les deux solutions dans le temps imparti. Les comparaisons des temps d'exécutions seront décrites dans la suite de ce document.

5. Tests et validations

Pour valider nos programmes C et C++, nous avons effectué les tests suivants :

- Tests unitaires : nous avons testé informatiquement les sorties de toutes les fonctions que nous avons développées en vérifiant qu'elles fournissaient les sorties attendues
- Tests fonctionnels : une fois toutes les fonctions testées et enchainées les unes aux autres, nous avons effectué des tests fonctionnels afin de vérifier la validité de nos programmes. Ces tests sont très semblables à ceux effectués avec Matlab et décrits dans les parties

précédentes : nous avons, par exemple, fixé une propagation uniquement suivant l'axe x, testé différentes valeurs de probabilités primaires et secondaires …

Dans ce rapport, nous ne décrirons pas plus précisément tous les tests que nous avons effectués pendant ce projet. Les tests couvrent toutes les fonctions et toutes les fonctionnalités.

G. Application concrète : Modélisation d'épidémies végétales

Agent pathogène : champignon tellurique *Rhizoctonia solani*, polyphage et croissance saprophytique, responsable de nombreux dégâts sur les cultures.

1er cas Propagation du rhizoctone sur un gazon

Certaines souches de rhizoctone s'attaquent au gazon provoquant des tâches brunes.

Hypothèses de modélisation :

- l'espace est considéré comme isotrope
- l'inoculum primaire (donc indirectement les infections primaires) décroissent de manière exponentielle dans le temps (rp= a * exp(-b*t))
- la probabilité d'infection secondaire est constante au cours du temps

Simulations :

Pas de temps d'un jour, temps maximal de 300j. Grille 300*300

On compare deux cas : faible et forte concentration d'inoculum primaire

→ rp = 0.01 * exp(-0.005*t) et 0.0001*exp(-0.005*t)

Pour chaque cas on compare un cas où les infections secondaires sont probables ainsi qu'un autre cas où elles le sont moins rsx=rsy=0.001 et 0.0001

Simuler 100 réplicas pour chaque cas, quel cas ressemble le plus à la photo ?

Les simulations correspondent aux figures suivantes :

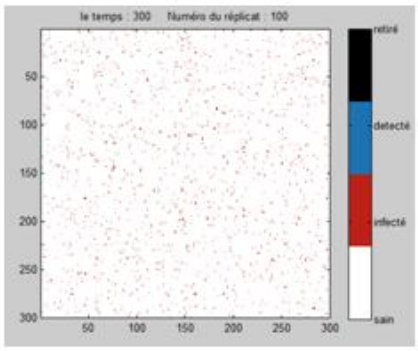


Figure 1 : Faible primaire - Faible secondaire

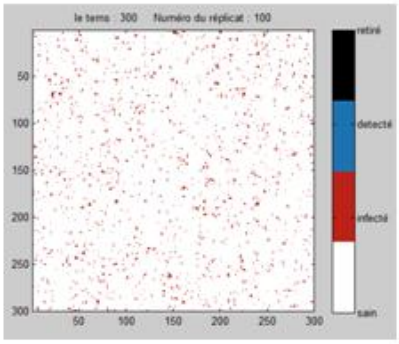


Figure 2 : Faible primaire - Forte secondaire

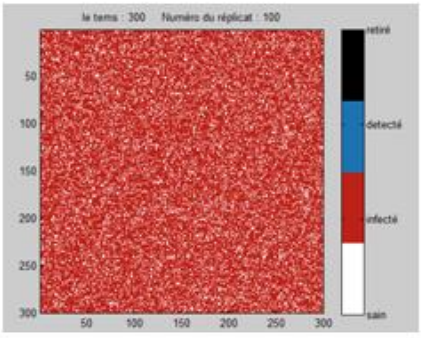


Figure 3 : Forte primaire - Faible secondaire

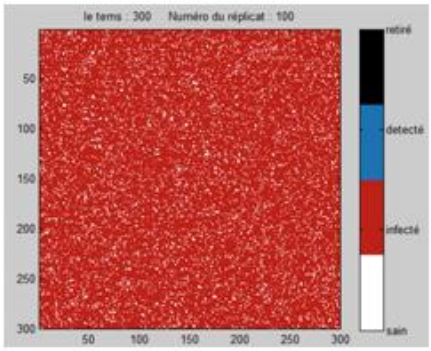
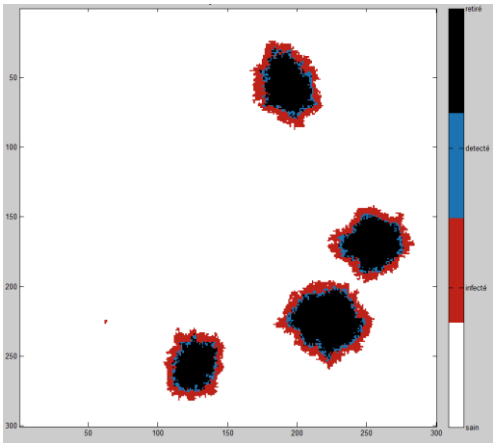


Figure 4 : Forte primaire - Forte secondaire



La grille est de taille 300x300.

Le temps de la simulation = 300j

Infection initiale = aucune

Nombre de voisins = 4 (voisinage de Von Neumann)

Probabilité primaire = 2e-7 * exp(-1 e-5 * t)

Probabilité secondaire :

rsx = 0.05 pour toutes les plantes infectées

rsy = 0.05

H. Comparaison des temps d'exécution

Dans cette partie nous allons comparer les temps d'exécution des différents programmes : Matlab, C++ et C++ multithreadé avec les deux versions. Le but est de connaître la rapidité de chaque langage et de pouvoir identifier l'influence de chaque paramètre sur les temps d'exécution.

Nous nous sommes définis une configuration de référence qui est la suivante :

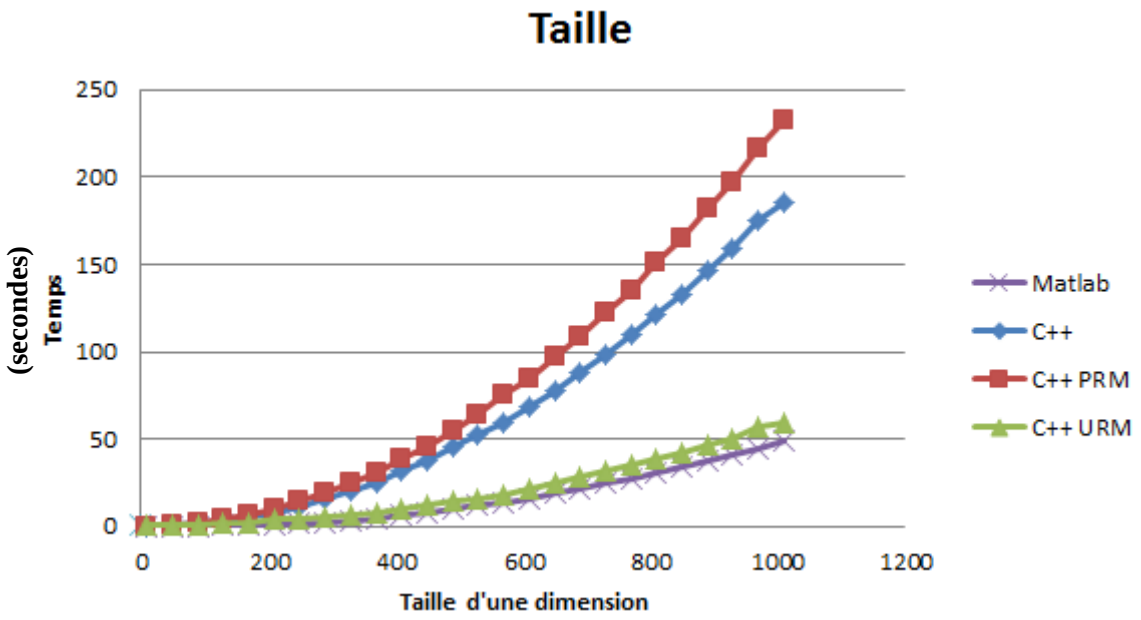
- Taille : 300 par 300
- Durée de la simulation : 100 pas de temps

- Nombre de réplicas : 1
- Voisinage : 8

Ensuite, nous avons fait varier chacun des paramètres afin de connaître leurs influences et de voir les technologies les plus performantes. Nous avons lancé cinq fois chaque configuration afin d'obtenir les durées moyennes.

Variation de la taille de la grille :

Nous avons commencé par faire varier la taille de la grille (*10 x 10, puis 20 x 20 … 1000 x 1000*) tout en laissant les autres paramètres constants. Nous obtenons les performances suivantes :



Pour ce cas précis (et c'est souvent ce qu'on retrouve), la technologie la plus rapide est **Matlab**. Le **C++ URM** (Un Réplica Multithreadé, cf. partie sur le multithreading) est un tout petit peu moins rapide mais l'écart n'est pas flagrant.

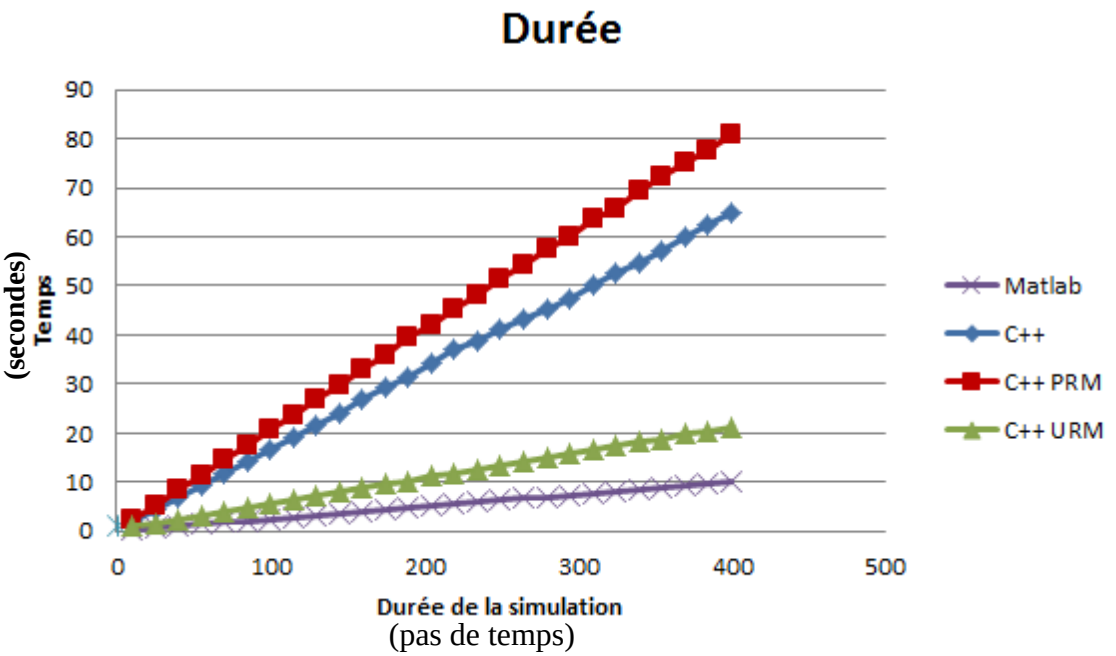
Par contre le C++ et le C++ PRM (Plusieurs Réplica Multithreadé, cf. partie sur le multithreading) sont vraiment plus lents et ne sont pas conseillés pour des grilles de grandes dimensions. On remarque même, dans ce cas, que le C++ multithreadé PRM est moins rapide que le C++ normal. C'est normal car on ne lance qu'un seul réplica et cette version de multithreading est optimale pour plusieurs réplicas.

L'allure des courbes est un polynôme d'ordre deux. Ceci est tout à fait cohérent et était tout à fait prévisible.

Pour de grandes grilles on peut utiliser indifféremment Matlab ou le C++ URM.

Variation de la durée de la simulation :

Puis, nous avons fait varier la durée de la simulation tout en laissant les autres paramètres constants. Nous obtenons les résultats suivants :



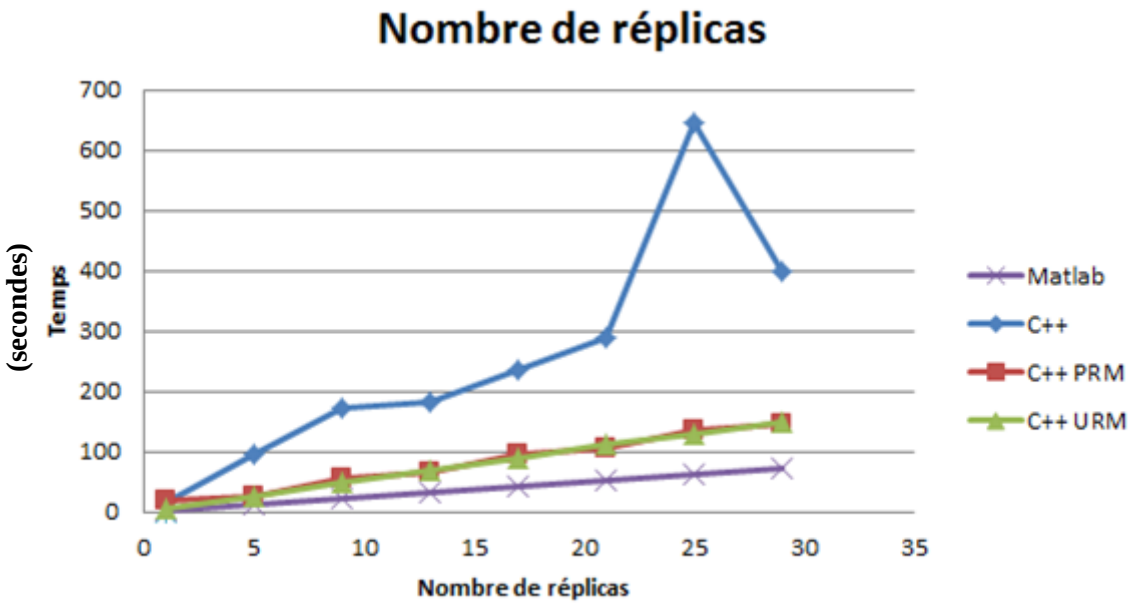
Encore une fois **Matlab** est toujours le plus rapide, suivi par le **C++ URM**. On remarque toutefois que l'écart entre les deux est un peu plus important que pour les simulations précédentes. Les deux autres technologies sont, encore une fois, beaucoup moins rapides.

L'allure des courbes est linéaire, ce qui est aussi cohérent et prévisible.

Pour des durées importantes, on conseille donc d'utiliser Matlab.

Variation du nombre de réplicas :

Nous avons ensuite fait varier le nombre de réplicas et nous obtenons les résultats suivants :



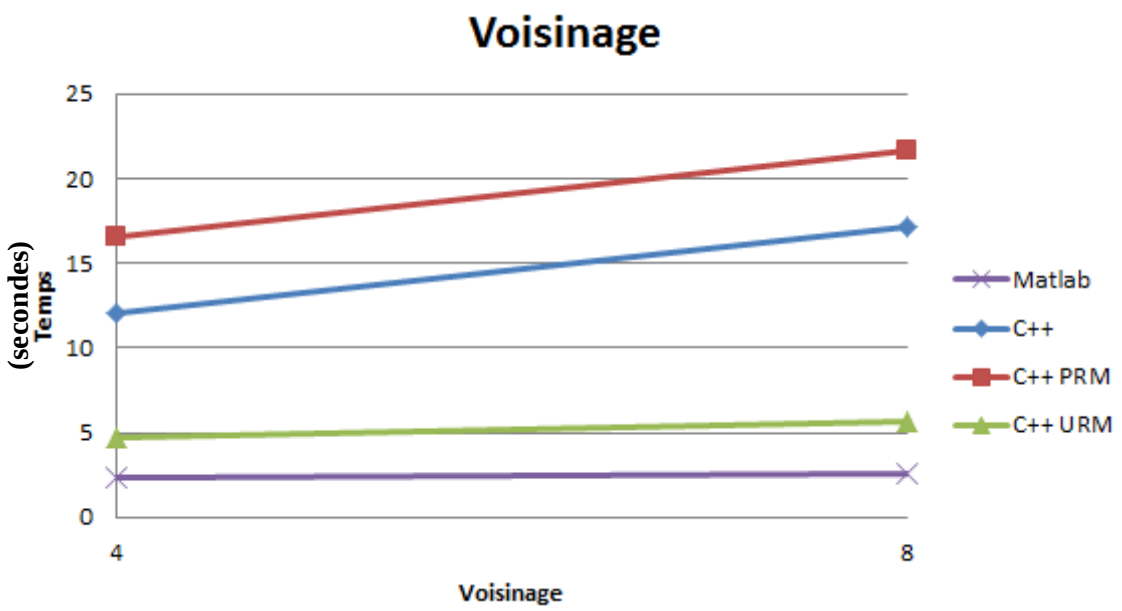
Cette fois-ci le C++ PRM montre son intérêt puisqu'il se rapproche du Matlab et du C++ URM. Cependant Matlab est toujours le plus performant.

L'allure des courbes est à peu près linéaire. Nous avons toutefois un point dont la valeur est étrange (pour 25 réplicas). Nous avons relancé ce calcul plusieurs fois mais nous obtenons toujours à peu près les mêmes temps d'exécution.

Pour beaucoup de réplicas, on conseille d'utiliser Matlab ou les deux C++.

Variation du voisinage :

Puis, nous avons fait varier le voisinage (4 ou 8). Les résultats sont les suivants :

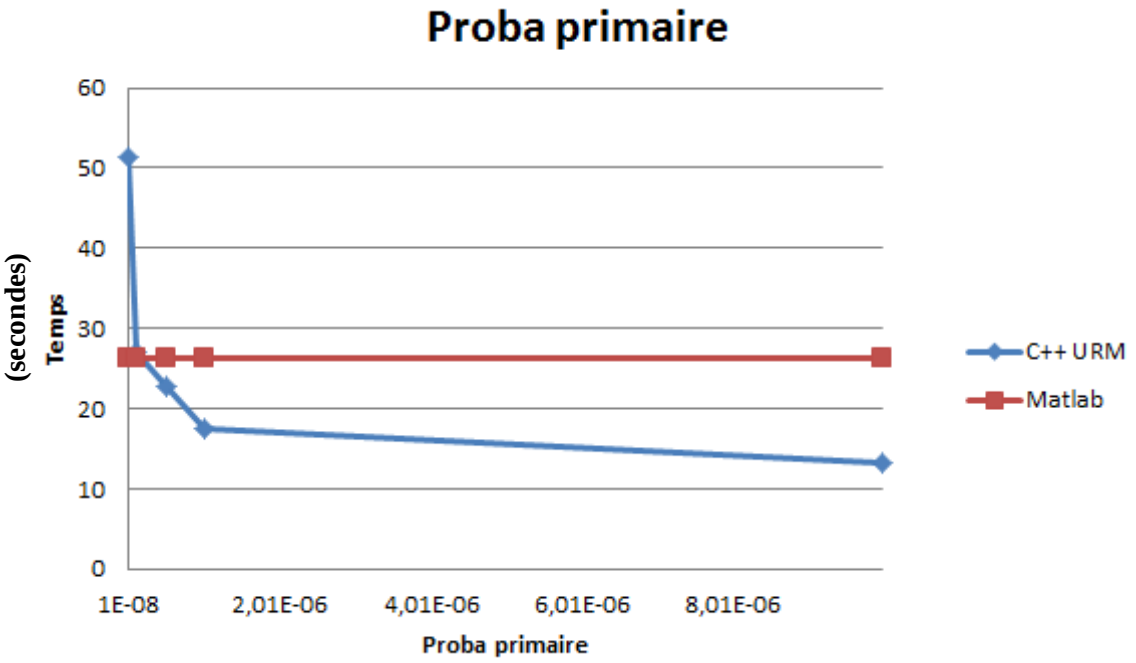


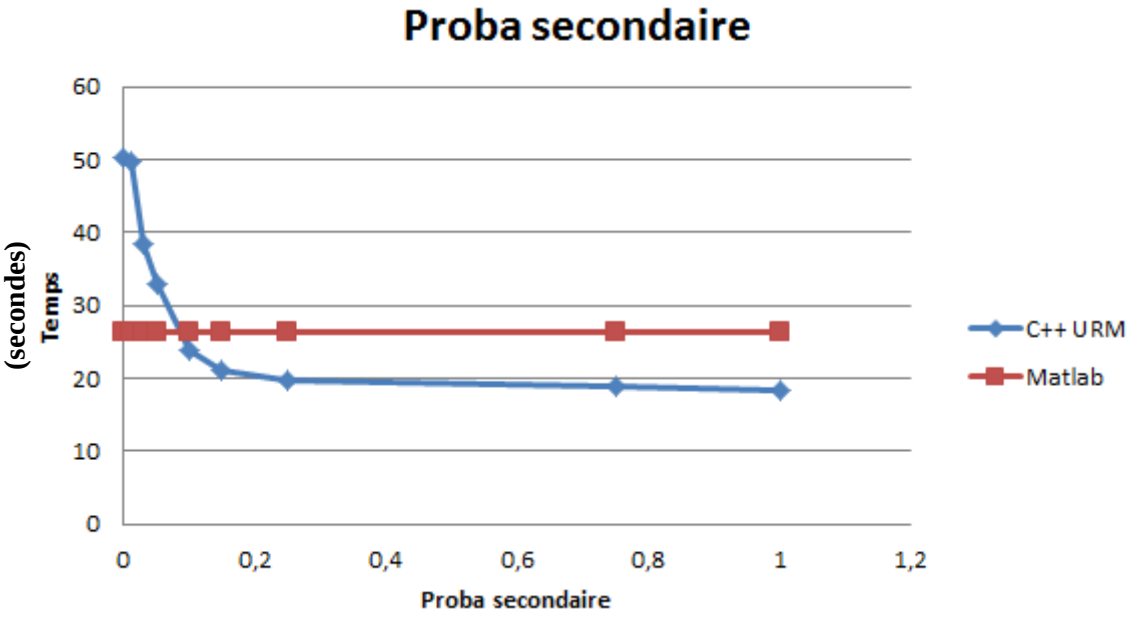
La technologie la plus rapide est encore **Matlab** suivi par le **C++ URM** puis le C++ et enfin le C++ PRM.

Pour Matlab le voisinage n'influe pas du tout sur la durée d'exécution ce qui est tout à fait normal lorsqu'on connait l'implémentation. En C++ URM, elle a une petite influence. Par contre, en C++ et C++ PRM ce paramètre a une plus grande influence.

Variation des probabilités :

Pour terminer, nous avons fait varier les probabilités primaires et secondaires. Ces probabilités n'ont pas d'influence sur le temps d'exécution de Matlab par contre elles influent beaucoup sur celui du C++ (qu'il soit multithreadé ou pas). En effet, moins il y a de cellules saines et plus le programme C++ va vite (car il n'a pas besoin d'aller examiner les voisins de chaque cellule). Nous obtenons les résultats suivants :





Pour des probabilités faibles, Matlab est plus rapide par contre dès qu'on monte dans des probabilités plus importantes le C++ devient nettement plus rapide.

Bilan :

Pour conclure sur les comparaisons, les deux technologies à retenir sont :

- **Matlab** reste la technologie la plus rapide actuellement. Il faut bien comprendre que le code Matlab a été développé de façon très optimale (sans boucles for …) mais du coup est relativement difficile à prendre en main et difficilement modifiable pour d'autres cas.

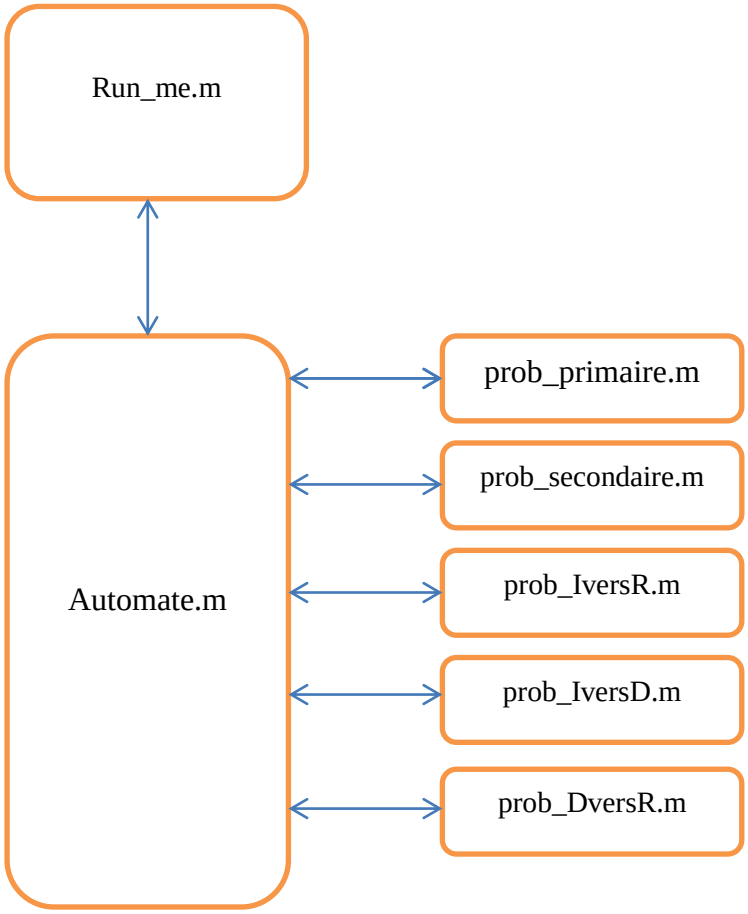
- **C++ URM** est un peu moins rapide que Matlab mais s'en rapproche pas mal. Par contre le code a plutôt été développé pour être facile à prendre en main et très facilement modifiable et extensible à d'autres cas (ce choix a été fait pour que vous puissiez comprendre et maîtriser nos algorithmes sans avoir de notions avancées en C++).

Tout en sachant que Matlab est plus rapide si les probabilités primaires et secondaires sont faibles, et plus lent si elles sont élevées.

III. Lancement et utilisation des programmes

A. Automate cellulaire en Matlab SIDR

1. Fichiers



2. Fichier Run_me.m

Le fichier Run_me.m est le fichier d'initialisation qui permet de configurer les paramètres d'entrée de l'automate. Ces paramètres seront détaillés dans la partie suivante. Une fois les paramètres définis, l'automate est lancé. En sortie de celui-ci, nous obtenons un certain nombre de variables (explicitées dans la partie suivante), dont 3 matrices 3d :

`Matrice_temps_infecte`, `Matrice_temps_detecte`, `Matrice_temps_retire`.
A partir de ces trois matrices correspondant au temps depuis lequel un élément d'une grille est infecté, détecté et retiré, nous pouvons reconstruire l'état de la grille à chaque instant, quel que soit le réplica.

La deuxième partie de ce fichier permet d'afficher l'état de la grille à chaque instant. La variable `pas_de_temps_affichage` permet de définir le pas de temps de l'affichage. L'affichage se fait donc une fois que l'automate a effectué l'ensemble de ses calculs.

Ce fichier permet de plus, de définir les couleurs de l'affichage de la grille. Les couleurs sont définies au format RGB pour chaque état des cellules, sain, infecté, détecté, retiré.

Enfin ce fichier permet d'afficher les évolutions des paires ainsi que le nombre d'éléments sains, infectés, détectés ou retirés au cours de la simulation.

3. Fonctions automate

```
function [Matrice_temps_infecte, Matrice_temps_detecte, Matrice_temps_retire, nb_S, nb_I, nb_D, nb_R, II,ID, IR, DD, DR, RR, SS, SI, SD, SR, DI, RI, RD, IS, DS, RS] = automate (grid_initial, Matrice_temps_infecte_initial, Matrice_temps_detecte_initial, Matrice_temps_retire_initial, time_max, nbre_voisins, nbre_replicat, dx, dy, dz)
```

a) Description

La fonction *automate* est le cœur du programme. C'est elle qui gère l'algorithme épidémiologique de type SIDR (cf. spécifications).

b) Paramètres d'entrée

Grid_initial : il s'agit d'une matrice définie par l'utilisateur comportant des éléments sains, infectés, détectés et retirés. Cette grille permet d'initialiser la grille de départ à un instant quelconque d'une simulation.

Matrice_temps_infecte_initial : Il s'agit d'une matrice représentant la durée depuis laquelle les éléments, de la grille initiale, sont infectés. Attention, un élément détecté reste un élément

infecté, avec une certaine durée depuis laquelle il est infecté. Cette matrice a la même taille que la grille initiale.

Matrice_temps_détecté_initial : Il s'agit d'une matrice représentant la durée depuis laquelle les éléments, de la grille initiale, sont détectés. Cette matrice a la même taille que la grille initiale.

Time_max : temps que va durer la simulation. Il s'agit d'un entier positif.

Nbre_voisins : On définit ici, le type de voisinage, Von Neumann ou Moore. La valeur du paramètre est nécessairement 4 ou 8 pour Von Neumann ou Moore. Si une autre valeur est passée, une erreur est produite.

nbre_replicat : Il s'agit d'un entier positif, qui définit le nombre de réplicas à simuler. Chaque réplica aura les mêmes conditions d'initialisations.

dx : distance entre deux éléments suivants x

dy : distance entre deux éléments suivants y

dz : distance entre deux éléments suivants z

c) Paramètres de sortie

Matrice_temps_infecte : matrice 3d représentant la durée depuis laquelle les éléments, de la grille, sont infectés. Attention un élément détecté ou retiré voit son temps d'infection augmenter. Les lignes et colonnes représentent les éléments de la grille. La 3ème dimension, représente l'état des réplicas.

Matrice_temps_detecte : matrice représentant la durée depuis laquelle les éléments, de la grille, sont détectés. Attention un élément détecté ou retiré voit son temps de détection augmenter. Les lignes et colonnes représentent les éléments de la grille. La 3ème dimension, représente l'état des réplicas.

Matrice_temps_retire : matrice représentant la durée depuis laquelle les éléments, de la grille, sont retirés. Attention un élément retiré voit son temps de détection augmenter. Les lignes et

colonnes représentent les éléments de la grille. La 3ème dimension, représente l'état des réplicas.

nb_S : vecteur représentant le nombre d'éléments sains au cours du temps. Attention, le premier élément de ce vecteur représente l'état initial.

nb_I : vecteur représentant le nombre d'éléments infectés au cours du temps. Attention, le premier élément de ce vecteur représente l'état initial.

nb_D : vecteur représentant le nombre d'éléments détectés au cours du temps. Attention, le premier élément de ce vecteur représente l'état initial.

nb_R : vecteur représentant le nombre d'éléments retirés au cours du temps. Attention, le premier élément de ce vecteur représente l'état initial.

II : Vecteur représentant le nombre de paires infecté-infecté au cours du temps. Attention, le premier élément de ce vecteur représente l'état initial.

ID, IR, DD, DR, RR, SS, SI, SD, SR, DI, RI, RD, IS, DS, RS : idem que II pour les paire : infecté-détecté, infecté-retiré, détecté-détecté, détecté-retiré, retiré-retiré, sain-sain, sain-infecté, sain-détecté, sain-retiré, détecté –infecté, retiré- infecté, retiré- détecté, infecté-sain, détecté-sain, retiré-sain

4. Fonctions prob_primaire

```
function [rp] = prob_primaire(width_grid,length_grid,
time)
```

a) Description

Fonction qui calcule la probabilité qu'une plante devienne infectée aléatoirement. Il s'agit d'une matrice 3d représentant la probabilité en fonction du temps et de l'espace. La première dimension correspond aux nombre de lignes de la grille, la deuxième, au nombre de colonnes et la dernière dimension au temps de la simulation

b) Paramètres d'entrée

width_grid : nombre de lignes de la grille

length_grid : nombre de colonnes de la grille

`time_max`: durée maximale de la simulation

c) Paramètres de sortie

Rp : matrice 3d des probabilités primaires dont l'ordre des dimensions est : 1ère dimension = lignes, 2ème dimension = colonnes, 3ème dimension = temps de la simulation

5. Fonctions prob_secondaire

```
function [rsx rsy rsz] = prob_secondaire(dx, dy, dz, time_max, time_max_infecte)
```

a) Description

Calcul des probabilités secondaires, rsx, rsy ,rsz, qu'une cellule infectée, infecte son voisin suivant x, y ou z, en fonction de son âge et de sa durée d'infection. Ce sont des matrices où les lignes représentent l'âge des cellules et les colonnes, leur durée d'infection.

Exemple : rsx(20,10) représente la probabilité d'infection secondaire d'une cellule infectée depuis 10 jours et âgée de 20 jours

b) Paramètres d'entrée

Time_max : temps maximal de la durée de la simulation.

`time_max_infecte`: temps maximal de la durée d'infection d'une cellule. Attention, ce nombre doit tenir compte de l'initialisation de la grille et de la durée de la simulation. En effet, un élément infecté, initialement depuis un temps t, a la possibilité de rester infecté toute la durée de la simulation donc jusqu'à `time_max_infecte` +Time_max.

dx : distance entre deux éléments suivants x

dy : distance entre deux éléments suivants y

dz : distance entre deux éléments suivants z

c) Paramètres de sortie

Rsx : Matrice de probabilité d'infection secondaire suivant x. Ce sont des matrices où les lignes représentent l'âge des cellules et les colonnes, leur durée d'infection.

Rsy : Matrice de probabilité d'infection secondaire suivant y. Ce sont des matrices où les lignes représentent l'âge des cellules et les colonnes, leur durée d'infection.

Rsz : Matrice de probabilité d'infection secondaire suivant z. Ce sont des matrices où les lignes représentent l'âge des cellules et les colonnes, leur durée d'infection.

6. Fonctions prob_IversR

```
function transition_IversR = prob_IversR(time_max, time_max_infecte)
```

a) Description

Cette fonction calcule la probabilité de passer d'un état infecté à un état retiré. Cette probabilité dépend de la durée de la simulation et de la durée depuis laquelle une plante est infectée.

b) Paramètres d'entrée

Time_max : temps maximal de la durée de la simulation.

`time_max_infecte`: temps maximal de la durée d'infection d'une cellule. Attention, ce nombre doit tenir compte de l'initialisation de la grille et de la durée de la simulation. En effet, un élément infecté, initialement depuis un temps t, a la possibilité de rester infecté toute la durée de la simulation donc jusqu'à `time_max_infecte` +Time_max.

c) Paramètres de sortie

transition_IversR : Matrice de probabilité de passer de l'état infecté à l'état retiré. Les lignes représentent le temps de la simulation tandis que les colonnes, la durée depuis laquelle une plante est infectée.

7. Fonctions prob_IversD

```
function transition_IversD = prob_IversD(time_max, time_max_detecte)
```

a) Description

Cette fonction calcule la probabilité de passer d'un état infecté à un état détecté. Cette probabilité dépend de la durée de la simulation et de la durée depuis laquelle une plante est infectée.

b) Paramètres d'entrée

Time_max : temps maximal de la durée de la simulation.

`time_max_infecte`: temps maximal de la durée d'infection d'une cellule. Attention, ce nombre doit tenir compte de l'initialisation de la grille et de la durée de la simulation. En effet, un élément infecté, initialement depuis un temps t, a la possibilité de rester infecté toute la durée de la simulation donc jusqu'à `time_max_infecte` +Time_max.

c) Paramètres de sortie

transition_IversD : Matrice de probabilité de passer de l'état infecté à l'état détecté. Les lignes représentent le temps de la simulation tandis que les colonnes, la durée depuis laquelle une plante est infectée.

8. Fonctions prob_DversR

```
function transition_DversR = prob_DversR(time_max, time_max_detecte)
```

a) Description

Cette fonction calcule la probabilité de passer d'un état détecté à un état retiré. Cette probabilité dépend de la durée de la simulation et de la durée depuis laquelle une plante est détectée.

b) Paramètres d'entrée

Time_max : temps maximal de la durée de la simulation.

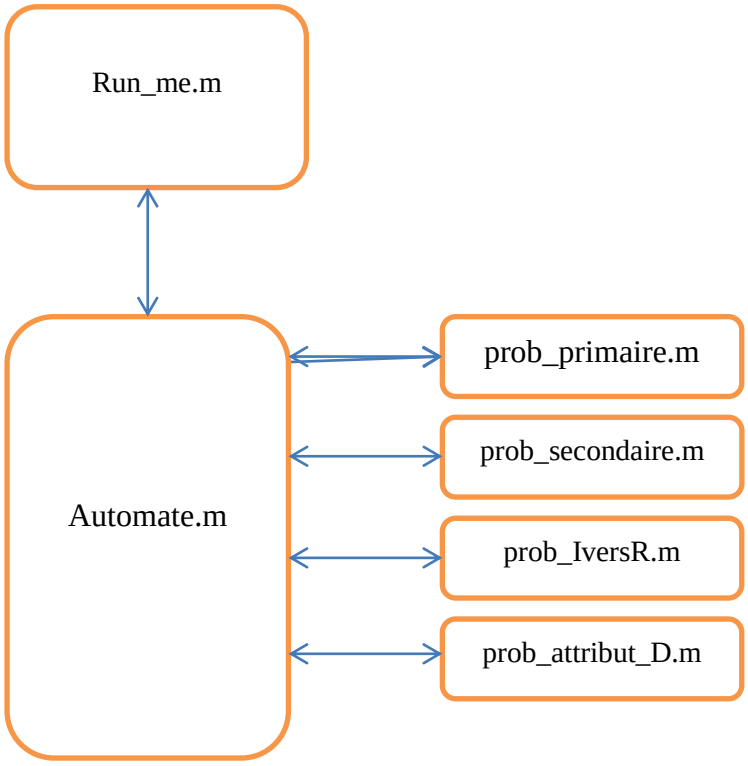
`time_max_detecte` temps maximal de la durée de détection d'une cellule. Attention, une plante initialement détectée aura un temps de détection plus long que les autres. En effet, un élément détecté, initialement depuis un temps t, a la possibilité de rester détecté toute la durée de la simulation donc jusqu'à `time_max_detecte` +Time_max.

c) Paramètres de sortie

transition_DversR : Matrice de probabilité de passer de l'état détecté à l'état retiré. Les lignes représentent le temps de la simulation tandis que les colonnes, la durée depuis laquelle une plante est détectée.

B. Automate cellulaire en Matlab SIR

1. Fichiers



2. Fichier Run_me.m

Le fichier Run_me.m est le fichier d'initialisation qui permet de configurer les paramètres d'entrée de l'automate. Ces paramètres seront détaillés dans la partie suivante. Une fois les paramètres définis, l'automate est lancé. En sortie de celui-ci, nous obtenons un certain nombre de variables (explicitées dans la partie suivante), dont 3 matrices 3d : `Matrice_temps_infecte`, `Matrice_temps_attribut_detecte`, `Matrice_temps_retire`. A partir de ces trois matrices correspondant au temps depuis lequel un élément d'une grille est infecté, détecté et retiré, nous pouvons reconstruire l'état de la grille à chaque instant, quel que soit le réplica.

La deuxième partie de ce fichier permet d'afficher l'état de la grille à chaque instant. La variable `pas_de_temps_affichage` permet de définir le pas de temps de l'affichage. L'affichage se fait donc une fois que l'automate à effectuer l'ensemble de ses calculs.

Ce fichier permet de plus, de définir les couleurs de l'affichage de la grille. Les couleurs sont définies au format RGB pour chaque état des cellules, sain, infecté, détecté, retiré.

Enfin ce fichier permet d'afficher les évolutions des paires ainsi que le nombre d'éléments sains, infectés, détectés ou retirés au cours de la simulation.

3. Fonction automate

```
function [Matrice_temps_infecte, Matrice_temps_attribut_detecte,
Matrice_temps_retire, nb_S, nb_I, nb_attributs_D, nb_R, II, IR, RR, SS, SI,
SR, RI, IS, RS] = automate (grid_initial, Matrice_temps_infecte_initial,
Matrice_temps_attribut_detecte_initial, Matrice_temps_retire_initial,
time_max, nbre_voisins, nbre_replicat, dx, dy, dz)
```

a) Description

La fonction *automate* est le cœur du programme. C'est elle qui gère l'algorithme épidémiologique de type SIDR (cf spécifications).

b) Paramètres d'entrée

Grid_initial : il s'agit d'une matrice définie par l'utilisateur comportant des éléments sains, infectés, détectés et retirés. Cette grille permet d'initialiser la grille de départ à un instant quelconque d'une simulation.

Matrice_temps_infecte_initial : Il s'agit d'une matrice représentant la durée depuis laquelle les éléments, de la grille initiale, sont infectés. Attention, un élément détecté reste un élément infecté, avec une certaine durée depuis laquelle il est infecté. Cette matrice a la même taille que la grille initiale.

Matrice_temps_attribut_détecté_initial : Il s'agit d'une matrice représentant la durée depuis laquelle les éléments infectés, de la grille initiale, ont l'attribut détecté. Cette matrice a la même taille que la grille initiale.

Time_max : temps que va durer la simulation. Il s'agit d'un entier positif.

Nbre_voisins : On définit ici, le type de voisinage, Von Neumann ou Moore. La valeur du paramètre est nécessairement 4 ou 8 pour Von Neumann ou Moore. Si une autre valeur est passée, une erreur est produite.

nbre_replicat : Il s'agit d'un entier positif, qui définit le nombre de réplicas à simuler. Chaque réplica aura les mêmes conditions d'initialisations.

dx : distance entre deux éléments suivants x

dy : distance entre deux éléments suivants y

dz : distance entre deux éléments suivants z

c) Paramètres de sortie

Matrice_temps_infecte : matrice 3d représentant la durée depuis laquelle les éléments, de la grille, sont infectés. Attention un élément détecté ou retiré voit son temps d'infection augmenter. Les lignes et colonnes représentent les éléments de la grille. La 3ème dimension, représente l'état des réplicas.

Matrice_temps_attribut_detecte : matrice représentant la durée depuis laquelle les éléments infectés, de la grille, ont l'attribut détecté. Attention un élément détecté ou retiré voit son temps de détection augmenter. Les lignes et colonnes représentent les éléments de la grille. La 3ème dimension, représente l'état des réplicas.

Matrice_temps_retire : matrice représentant la durée depuis laquelle les éléments, de la grille, sont retirés. Attention un élément retiré voit son temps de détection augmenter. Les lignes et colonnes représentent les éléments de la grille. La 3ème dimension, représente l'état des réplicas.

nb_S : vecteur représentant le nombre d'éléments sains au cours du temps. Attention, le premier élément de ce vecteur représente l'état initial.

nb_I : vecteur représentant le nombre d'éléments infectés au cours du temps. Attention, le premier élément de ce vecteur représente l'état initial.

nb_attributs_D : vecteur représentant le nombre d'éléments infectés qui ont eu l'attribut détecté au cours du temps. Attention, le premier élément de ce vecteur représente l'état initial.

nb_R : vecteur représentant le nombre d'éléments retirés au cours du temps. Attention, le premier élément de ce vecteur représente l'état initial.

II : Vecteur représentant le nombre de paires infecté-infecté au cours du temps. Attention, le premier élément de ce vecteur représente l'état initial.

IR, RR, SS, SI, SR, RI, IS, RS : idem que II pour les paire : infecté-retiré, retiré-retiré, sain-sain, sain-infecté, sain-retiré, retiré- infecté, infecté-sain, retiré-sain.

4. Fonction prob_primaire

```
function [rp] = prob_primaire(width_grid,length_grid,
time)
```

a) Description

Fonction qui calcule la probabilité qu'une plante devienne infectée aléatoirement. Il s'agit d'une matrice 3d représentant la probabilité en fonction du temps et de l'espace. La première dimension correspond aux nombre de lignes de la grille, la deuxième, au nombre de colonnes et la dernière dimension au temps de la simulation

b) Paramètres d'entrée

width_grid : nombre de lignes de la grille

length_grid : nombre de colonnes de la grille

`time_max`: durée maximale de la simulation

c) Paramètres de sortie

Rp : matrice 3d des probabilités primaires dont l'ordre des dimensions est : 1ère dimension = lignes, 2ème dimension = colonnes, 3ème dimension = temps de la simulation

5. Fonction prob_secondaire

```
function [rsx rsy rsz] = prob_secondaire(dx, dy, dz, time_max, time_max_infecte)
```

a) Description

Calcul des probabilités secondaires, rsx, rsy ,rsz, qu'une cellule infectée, infecte son voisin suivant x, y ou z, en fonction de son âge et de sa durée d'infection. Ce sont des matrices où les lignes représentent l'âge des cellules et les colonnes, leur durée d'infection.

Exemple : rsx(20,10) représente la probabilité d'infection secondaire d'une cellule infectée depuis 10 jours et âgée de 20 jours

b) Paramètres d'entrée

Time_max : temps maximal de la durée de la simulation.

`time_max_infecte`: temps maximal de la durée d'infection d'une cellule. Attention, ce nombre doit tenir compte de l'initialisation de la grille et de la durée de la simulation. En effet, un élément infecté, initialement depuis un temps t, a la possibilité de rester infecté toute la durée de la simulation donc jusqu'à `time_max_infecte` +Time_max.

dx : distance entre deux éléments suivants x

dy : distance entre deux éléments suivants y

dz : distance entre deux éléments suivants z

c) Paramètres de sortie

Rsx : Matrice de probabilité d'infection secondaire suivant x. Ce sont des matrices où les lignes représentent l'âge des cellules et les colonnes, leur durée d'infection.

Rsy : Matrice de probabilité d'infection secondaire suivant y. Ce sont des matrices où les lignes représentent l'âge des cellules et les colonnes, leur durée d'infection.

Rsz : Matrice de probabilité d'infection secondaire suivant z. Ce sont des matrices où les lignes représentent l'âge des cellules et les colonnes, leur durée d'infection.

6. Fonction prob_IversR

```
function transition_IversR = prob_IversR(time_max, time_max_infecte)
```

a) Description

Cette fonction calcule la probabilité de passer d'un état infecté à un état retiré. Cette probabilité dépend de la durée de la simulation et de la durée depuis laquelle une plante est infectée.

b) Paramètres d'entrée

Time_max : temps maximal de la durée de la simulation.

`time_max_infecte`: temps maximal de la durée d'infection d'une cellule. Attention, ce nombre doit tenir compte de l'initialisation de la grille et de la durée de la simulation. En effet, un élément infecté, initialement depuis un temps t, à la possibilité de rester infecté toute la durée de la simulation donc jusqu'à `time_max_infecte` +Time_max.

c) Paramètres de sortie

transition_IversR : Matrice de probabilité de passer de l'état infecté à l'état retiré. Les lignes représentent le temps de la simulation tandis que les colonnes, la durée depuis laquelle une plante est infectée.

7. Fonction prob_attibut_D

```
function matrice_proba_D = prob_attribut_D(time_max, time_max_infecte)
```

a) Description

Cette fonction calcule la probabilité de passer qu'un état infecté est l'attribut détecté. Cette probabilité dépend de la durée de la simulation et de la durée depuis laquelle une plante est infectée.

b) Paramètres d'entrée

Time_max : temps maximal de la durée de la simulation.

`time_max_infecte`: temps maximal de la durée d'infection d'une cellule. Attention, ce nombre doit tenir compte de l'initialisation de la grille et de la durée de la simulation. En effet, un élément infecté, initialement depuis un temps t, a la possibilité de rester infecté toute la durée de la simulation donc jusqu'à `time_max_infecte` +Time_max.

c) Paramètres de sortie

`matrice_proba_D`: Matrice de probabilité qu'une cellule infectée est l'attribut détecté. Les lignes représentent le temps de la simulation tandis que les colonnes, la durée depuis laquelle une plante est infectée.

C. Automate cellulaire en GPUmat

Avant de pouvoir lancer le programme GPUmat il est indispensable d'avoir suivi le document Guide d'installation GPUmat en Annexe.

Comme évoqué dans la partie technique de GPUmat le changement est transparent pour l'utilisateur, c'est-à-dire qu'il ne verra aucune différence par rapport au lancement du code Matlab.

Pour lancer le programme GPU, l'utilisateur doit ouvrir le script *Run_meGPU* et l'exécuter. Ce script appelle la fonction *automateGPU* qui est la même fonction que *automate.m* mais sur le GPU.

1. Fonctions

```
function [Matrice_temps_infecte, Matrice_temps_detecte,
Matrice_temps_retire, nb_S, nb_I, nb_D, nb_R, II,ID, IR, DD, DR, RR, SS,
SI, SD, SR, DI, RI, RD, IS, DS, RS] = automateGPU (grid_initial,
Matrice_temps_infecte_initial, Matrice_temps_detecte_initial,
Matrice_temps_retire_initial, time_max, nbre_voisins, nbre_replicat, dx,
dy, dz)
```

a) Description

La fonction *automateGPU* est le cœur du programme. C'est elle qui gère l'algorithme épidémiologique de type SIDR (cf spécifications).

b) Paramètres d'entrée

Grid_initial : il s'agit d'une matrice définie par l'utilisateur comportant des éléments sains, infectés, détectés et retirés. Cette grille permet d'initialiser la grille de départ à un instant quelconque d'une simulation.

Matrice_temps_infecte_initial : Il s'agit d'une matrice représentant la durée depuis laquelle les éléments, de la grille initiale, sont infectés. Attention, un élément détecté reste un élément infecté, avec une certaine durée depuis laquelle il est infecté. Cette matrice a la même taille que la grille initiale.

Matrice_temps_détecté_initial : Il s'agit d'une matrice représentant la durée depuis laquelle les éléments, de la grille initiale, sont détectés. Cette matrice a la même taille que la grille initiale.

Time_max : temps que va durer la simulation. Il s'agit d'un entier positif.

Nbre_voisins : On définit ici, le type de voisinage, Von Neumann ou Moore. La valeur du paramètre est nécessairement 4 ou 8 pour Von Neumann ou Moore. Si une autre valeur est passée, une erreur est produite.

nbre_replicat : Il s'agit d'un entier positif, qui définit le nombre de réplicas à simuler. Chaque réplica aura les mêmes conditions d'initialisations.

dx : distance entre deux éléments suivants x

dy : distance entre deux éléments suivants y

dz : distance entre deux éléments suivants z

c) Paramètres de sortie

Matrice_temps_infecte : matrice 3d représentant la durée depuis laquelle les éléments, de la grille, sont infectés. Attention un élément détecté ou retiré voit son temps d'infection augmenter. Les lignes et colonnes représentent les éléments de la grille. La 3ème dimension, représente l'état des réplicas.

Matrice_temps_detecte : matrice représentant la durée depuis laquelle les éléments, de la grille, sont détectés. Attention un élément détecté ou retiré voit son temps de détection augmenter. Les lignes et colonnes représentent les éléments de la grille. La 3ème dimension, représente l'état des réplicas.

Matrice_temps_retire : matrice représentant la durée depuis laquelle les éléments, de la grille, sont retirés. Attention un élément retiré voit son temps de détection augmenter. Les lignes et colonnes représentent les éléments de la grille. La 3ème dimension, représente l'état des réplicas.

nb_S : vecteur représentant le nombre d'éléments sains au cours du temps. Attention, le premier élément de ce vecteur représente l'état initial.

nb_I : vecteur représentant le nombre d'éléments infectés au cours du temps. Attention, le premier élément de ce vecteur représente l'état initial.

nb_D : vecteur représentant le nombre d'éléments détectés au cours du temps. Attention, le premier élément de ce vecteur représente l'état initial.

nb_R : vecteur représentant le nombre d'éléments retirés au cours du temps. Attention, le premier élément de ce vecteur représente l'état initial.

II : Vecteur représentant le nombre de paires infecté-infecté au cours du temps. Attention, le premier élément de ce vecteur représente l'état initial.

ID, IR, DD, DR, RR, SS, SI, SD, SR, DI, RI, RD, IS, DS, RS : idem que II pour les paire : infecté-détecté, infecté-retiré, détecté-détecté, détecté-retiré, retiré-retiré, sain-sain, sain-infecté, sain-détecté, sain-retiré, détecté –infecté, retiré- infecté, retiré- détecté, infecté-sain, détecté-sain, retiré-sain

Les fonctions de transitons sont exactement les mêmes que celles utilisées en Matlab.

2. Problèmes et limites

En théorie l'utilisation de la carte graphique devrait décupler les performances de notre programme. Cependant nous avons remarqué que le temps d'exécution est environ dix fois plus long.

Après avoir étudié notre code nous avons constaté que la ligne :

```
matrice_rsx = rsx(t, Matrice_temps_infecte_sans_retire+1);
```

Ce calcul renvoi la probabilité que chaque élément infecte son voisin. (Pour de plus amples informations, merci de se référer à la partie II.B automate cellulaire en Matlab)

Cette ligne s'exécute en 0.0035s quand les matrices sont sur le GPU contre 0.0015s sur le CPU. Or nous appelons cette fonction 4 fois par itérations, en fonction du temps de la simulation nous pouvons effectuer plus d'une centaine d'itérations. Les différences de temps d'exécutions entre les deux méthodes viennent donc de cette ligne.

Une solution simple serait de copier la matrice sur le CPU, faire le calcul puis la remettre sur le GPU, cependant les temps de lecture et d'écriture sont trop importants pour pouvoir en tirer un réel bénéfice. (0.06s pour copier, faire le calcul, et retourner sur le GPU).

Cette ligne est cruciale car elle permet de ne pas parcourir toutes les cases de la grille, ce qui est très long. Changer le code pour une telle optimisation nous prendrait beaucoup de temps et n'apporterait pas de résultats probants.

A l'heure actuelle nous ne pouvons pas expliquer d'où vient cette perte de temps, elle est récurrente et a été confirmée sur des exemples beaucoup plus simples. Il se pourrait que cette fonction n'ait pas été codée en GPUmat et entraine une copie implicite sur le CPU avant le calcul.

Cette constatation peut paraitre décevant quant aux performances attendues sur le GPU. Cependant il est important de noter que le programme initial n'a pas été pensé pour une optimisation sur GPU mais pour être optimal avec Matlab. Pour une réelle optimisation avec GPUmat il faudrait complètement réécrire le code et le penser pour GPUmat. Conformément avec le cahier des charges nous avons décidé de consacrer nos efforts sur le C++ multithreadé.

D. Développement en C++ multithreadé

Nous allons maintenant expliquer comment compiler et lancer les sources C et C++. Pour compiler et lancer les sources, il est indispensable d'avoir installé les bonnes versions de Visual Studio et de Boost (cf. annexes).

1. Compilation des sources

Compilation sous Windows :

Les projets *interface_fonctionTransition* et *automate_cellulaire* sont compilés en utilisant Microsoft Visual Studio afin de générés deux DLL compatibles avec Matlab et R. Attention toutefois à **compiler en mode Release** et surtout dans la bonne version (Win32 pour du 32 bits et x64 pour du 64 bits).

Les deux DLL générées doivent apparaître dans le dossier *interface_fonctionTransition\Win32\Release* ou *interface_fonctionTransition\x64\Release* en fonction de la version.

Compilation sous Linux :

Les projets *interface_fonctionTransition* et *automate_cellulaire* sont compilés en ligne de commande en lançant le script *Compilation_linux_32.sh* ou *Compilation_linux_64.sh* dans un terminal.

Les deux SO générées doivent apparaître dans le dossier *interface_fonctionTransition/Lin32* ou *interface_fonctionTransition/Lin64.*

2. Lancement d'une simulation

Lancement via Matlab :

Pour lancer une simulation avec Matlab, il suffit de :

- Dans le fichier *Lancement_simulation.m*, mettre à jour le chemin vers les DLL dans la ligne suivante (attention à la version) :

```
dossier_execution = [dossier_courant,
'\interface_fonctionTransition\x64\Release\'];
```

- Vérifier que dans ce répertoire, on trouve bien les deux DLL et surtout le fichier *lancerSimulation.m* qui permet d'appeler les DLL

- Enfin, configurer les entrées de la simulation et lancer le script *Lancement_simulation.m*.

Lancement via R :

Pour lancer une simulation avec R, il suffit d'ouvrir le script Lancement_simulation.R. Ce fichier contient à la fois les initialisations de l'automate cellulaire et l'appel de la dll ou du .so. L'utilisateur doit mettre à jour le chemin vers les .so :

Lors de la première utilisation, il faut changer la ligne de code R :

```
Setwd(" …/Codecpp/interface_fonctionTransition/Lin32/Release")
```

Le dossier Codecpp/interface_fonctionTransition contient quatre sous-dossiers :

- Win32 (Windows 32 bits)
- x64 (Windows 64 bits)
- Lin32 (Linux 32 bits)
- Lin64 (Linux 64 bits)

Dans la ligne de commande ci-dessus, l'utilisateur doit bien renseigner le chemin du dossier correspondant à sa configuration. Ce dossier correspond à l'emplacement du .so (ou .dll) en fonction de l'OS désiré. (L'exemple ci-dessus fonctionne pour Linux 32 bits).

La ligne de commande : *Dyn.load("interface_fonctionTransition.so")* permet d'appeler le .so , il faut remplacer cette ligne par *Dyn.load("interface_fonctionTransition.dll")*, si l'on souhaite appeler une dll.

Finalement l'utilisateur n'a plus qu'à lancer l'exécution du script pour lancer le programme C et C++.

E. Interface Matlab et R

Cette partie sera consacrée à la description des interfaces Matlab et R. Pour chacune d'elle, nous décrirons comment nous fournissons les données au C++, comment nous récupérons les résultats et la façon dont nous les présentons.

1. Interface Matlab

L'interface Matlab est très simple et simplement composée de deux fichiers : *Lancement_simulation.m* et *lancerSimulation.m*. Ces deux fichiers vont être décris l'un après l'autre.

Le fichier principal *Lancement_simulation.m* se situe à la racine du projet. Au début de ce fichier, on définit l'ensemble des configurations de la simulation (excepté les fonctions de transition qui sont définies dans le programme C) :

- les distances dx, dy et dz
- la durée de la simulation
- les dimensions de la grille
- le nombre de réplicas
- le voisinage

- la configuration initiale de la grille. Celle-ci est définie par un booléen *config_init* qui indique (0 ou 1) si nous avons une distribution initiale de l'agent pathogène, et trois matrices 2D (*duree_infection_t0*, *duree_detection_t0*, *duree_retrait_t0*) qui indiquent pour chaque cellule les durées depuis lesquelles les cellules sont infectées, détectées ou retirées à l'état initial.

Après avoir défini les configurations, ce fichier appelle le script *lancerSimulation.m* qui sera décrit plus loin dans cette partie. Le lancement renvoie trois matrices 3D (taille : hauteur *x* largeur *x* nbRep) qui indiquent les durées d'infection, de détection et de retrait pour chaque réplica à l'état final. Il renvoie aussi des matrices qui indiquent les paires et le nombre de cellules dans un état donné (taille : dureeSimu+1 *x* nbRep).

Enfin, un affichage de tous les pas de temps est proposé. Il est aussi possible d'afficher le nombre de paires ou le nombre de cellules dans un état donné, pour chaque réplica et chaque pas de temps. Le code produisant l'affichage est exactement le même que celui de l'automate en Matlab.

Nous allons maintenant décrire le fichier *lancerSimulation.m*. Ce deuxième script est placé dans le dossier qui contient les deux librairies dynamiques (*interface_fonctionTransition\Win32\Release* ou *interface_fonctionTransition\x64\Release*).

Comme évoqué précédemment, ce fichier est appelé par le script principal, c'est lui qui charge la librairie dynamique C, effectue les deux appels (lancement et obtention des résultats) et libère la librairie. Les fonctions associées sont *loadlibrary*, *calllib* et *unloadlibrary*.

Outre les appels aux librairies, ce script s'occupe aussi de la mise en forme des matrices. En effet, sous Matlab nous travaillons uniquement avec des matrices alors qu'en C++ nous ne travaillons qu'avec des vecteurs (unidimensionnels). Il est donc nécessaire d'aplatir les matrices pour les envoyer à Matlab, puis de les remettre sous forme matricielle en sortie. Ceci est réalisé avec la fonction *reshape*.

2. Interface R

Afin de garder un minimum de cohérence nous avons essayé de « calquer » l'interface R sur celui utilisé en Matlab. Ainsi nous gardons les mêmes noms de fonctions, et la même structure. L'interface R est très simple et simplement composée d'un fichier : *Lancement_simulation.R*.

Le fichier principal *Lancement_simulation.R* se situe à la racine du projet. Au début de ce fichier, on définit l'ensemble des configurations de la simulation (excepté les fonctions de transition qui sont définies dans le programme C) :

- les distances dx, dy et dz
- la durée de la simulation
- les dimensions de la grille
- le nombre de réplicas
- le voisinage
- la configuration initiale de la grille. Celle-ci est définie par un booléen *config_init* qui indique (0 ou 1) si nous avons une distribution initiale de l'agent pathogène, et trois matrices 2D (*duree_infection_t0*, *duree_detection_t0*, *duree_retrait_t0*) qui indiquent pour chaque cellule les durées depuis lesquelles les cellules sont infectées, détectées ou retirées à l'état initial.

Après avoir défini les configurations, ce fichier appelle le .SO . Le lancement renvoie trois matrices 3D (taille : hauteur *x* largeur *x* nbRep) qui indiquent les durées d'infection, de détection et de retrait pour chaque réplica à l'état final. Un parcourt de ces boucles en fonction du temps permet d'afficher l'état du champ à chaque pas. Il renvoie aussi des matrices qui indiquent les paires et le nombre de cellules dans un état donné (taille : dureeSimu+1 *x* nbRep).

Enfin, un affichage de tous les pas de temps est proposé. Il est aussi possible d'afficher le nombre de paires ou le nombre de cellules dans un état donné, pour chaque réplica et chaque pas de temps. Le code produisant l'affichage est semblable à celui de l'automate en Matlab.

Comme évoqué précédemment, c'est ce fichier qui charge la librairie dynamique C, effectue les deux appels (lancement et obtention des résultats) et libère la librairie. Les fonctions associées sont *dyn.load(), .C()* et *dynun.load().*

Outre les appels aux librairies, ce script s'occupe aussi de la mise en forme des matrices. En effet, sous R nous travaillons uniquement avec des matrices alors qu'en C ou C++ nous ne travaillons qu'avec des vecteurs (unidimensionnels). Il est donc nécessaire d'aplatir les matrices pour les envoyer au C, puis de les remettre sous forme matricielle en sortie.

- Initialisation du programme

  Il est très important de noter que lors de l'appel du .so ou de la dll, les variables doivent impérativement être des entiers. Il faut penser à caster ces variables car par défaut R les initialise en double. Par ailleurs le .so ne peut transmettre que des vecteurs mais pas de matrices, il faut donc convertir toutes ces valeurs. Pour cela on va utiliser les fonctions as.integer() et as.vector().

- Limites et problèmes récurrents avec R

Lors du développement du programme R nous avons du faire face à trois problèmes.

a) Installation de R sous Linux

Nous avons écrit et testé le programme R sous Windows, cependant nous n'avons pas pu installer R sous Linux par conséquent nous n'avons pas pu le tester dans cette configuration. Nous avons pu confirmer que le programme s'exécutait correctement sous Windows il n'y a pas de raisons qu'il ne fonctionne pas sous Linux.

b) Ouverture et fermeture de la dll

Un problème surprenant se produit lors de l'appel de la dll (dyn.load()). En fonction du système d'exploitation, lorsque que l'on appelle deux fois la dll, il se peut que R affiche un message d'erreur et force la fermeture. Cette erreur a été répertoriée par R et ne présente pas de solutions. Afin d'éviter ce problème nous vous conseillons de lancer une première fois les lignes de commande permettant de charger la dll, puis de les mettre en commentaire.

c) Affichage des graphiques

Dans le cahier des charges il nous est demandé d'étudier des tailles de champs de l'ordre de 1000*1000, cependant avec R nous n'arrivons pas à afficher des images aussi grosses. En accord avec l'INRA nous avons décidé de ne pas nous préoccuper de l'affichage de l'évolution de l'épidémie avec R.

Conclusion

Nous sommes très satisfaits de ce projet dans lequel nous nous sommes beaucoup investis tout au long de ce semestre, et qui pour nous représentait la concrétisation de toutes nos années d'études.

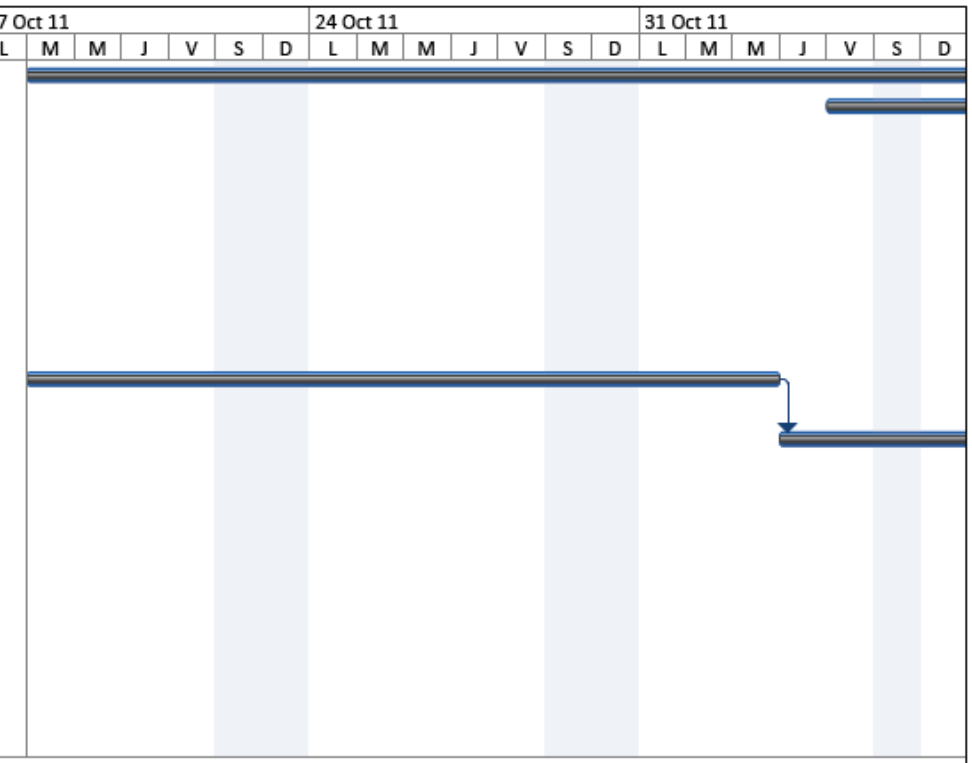
Le projet s'est très bien déroulé dans son ensemble, nous avons toujours été dans les temps et nous sommes bien entendu, que ce soit sur la technique ou pour s'organiser.

Nous sortons enrichis de cette collaboration et remercions encore une fois l'INRA et M. FEUILLOY pour leur temps accordé.

IV. Annexes

A. Diagramme de Gant

N°	Mode Tâche	Nom de la tâche	Durée	Début
1		Activité 1:Spécification	16 jours	Mar 18/10/11
2		Activité 2 : coder le programme en MATLAB	31 jours	Ven 04/11/11
3		Activité 3 : post-traitement	3 jours?	Ven 25/11/11
4		Activité 4 : Test et validation	13 jours?	Mer 30/11/11
5		Activité 5 : GPUmat	24 jours	Mar 15/11/11
6				
7				
8		Activité 1 : Prospection pour l'optimisation	12 jours?	Mar 18/10/11
9		Activité 2 : formation sur les différents langages	7 jours	Jeu 03/11/11
10		Activité 3 : choix du meilleur langage	1 jour	Mar 15/11/11
11		Activité 4 : Code C++	23 jours?	Mer 16/11/11
12		Activité 5 : Code C++ multithreadé	9 jours	Mar 03/01/12
13		activité 6 : Test et validation	9 jours	Mar 03/01/12
14				
15				



Projet : PFE_INRA
Date : Jeu 19/01/12

Tâche	Jalons externes	Report récapitulatif manuel
Fractionnement	Tâche inactive	Récapitulatif manuel
Jalon	Jalon inactif	Début uniquement
Récapitulative	Récapitulatif inactif	Fin uniquement
Récapitulatif du projet	Tâche manuelle	Échéance
Tâches externes	Durée uniquement	Avancement

Page 1

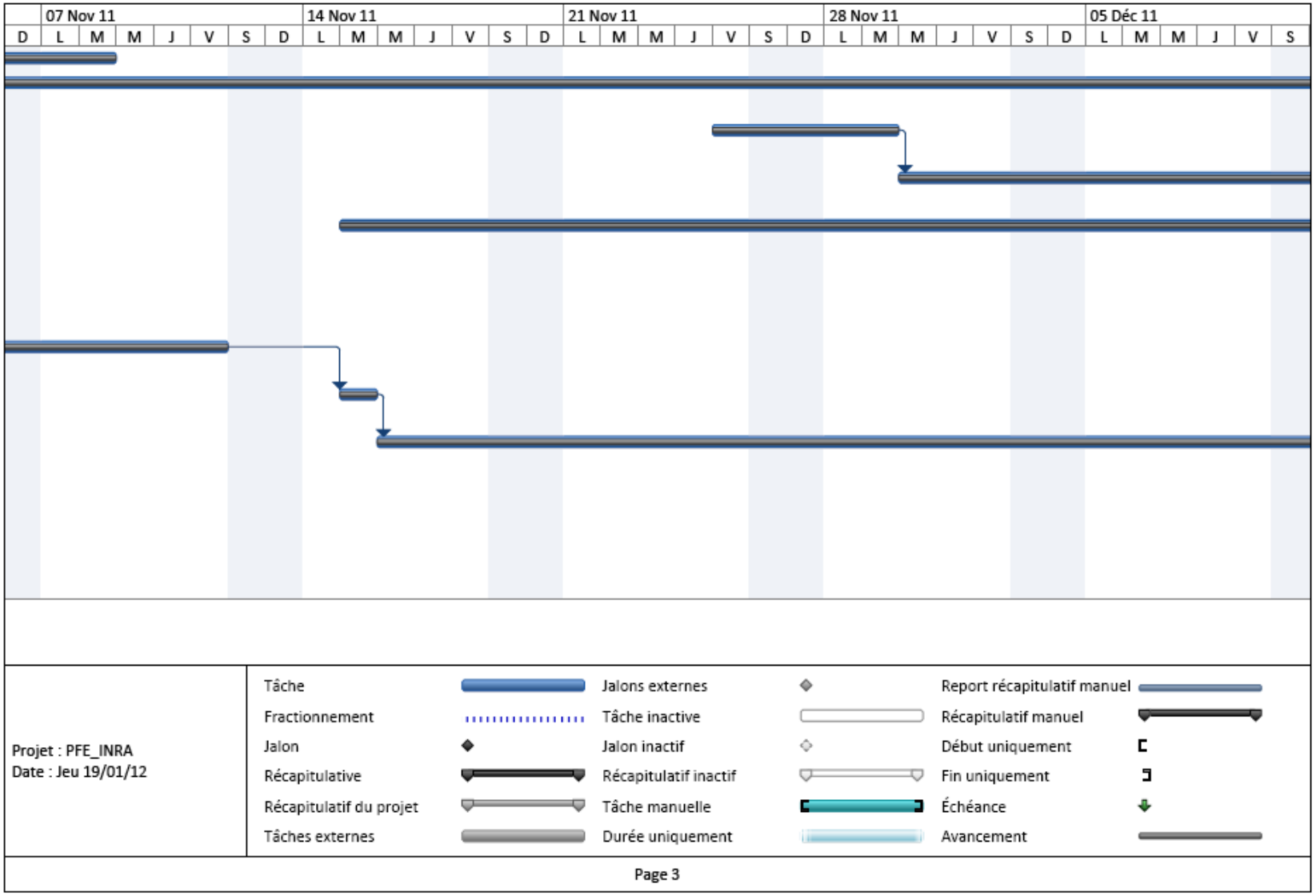
N°	Mode Tâche	Nom de la tâche	Durée	Début
16		Activité 1 : découverte et formation au langage R	5 jours?	Lun 21/11/11
17		Activité 2 : Tests & Post-traitement	16 jours	Sam 26/11/11
18				
19				
20		modèle SIR : activité 1	5 jours?	Lun 05/12/11
21		modèle SIR : activité 2	25 jours	Lun 12/12/11

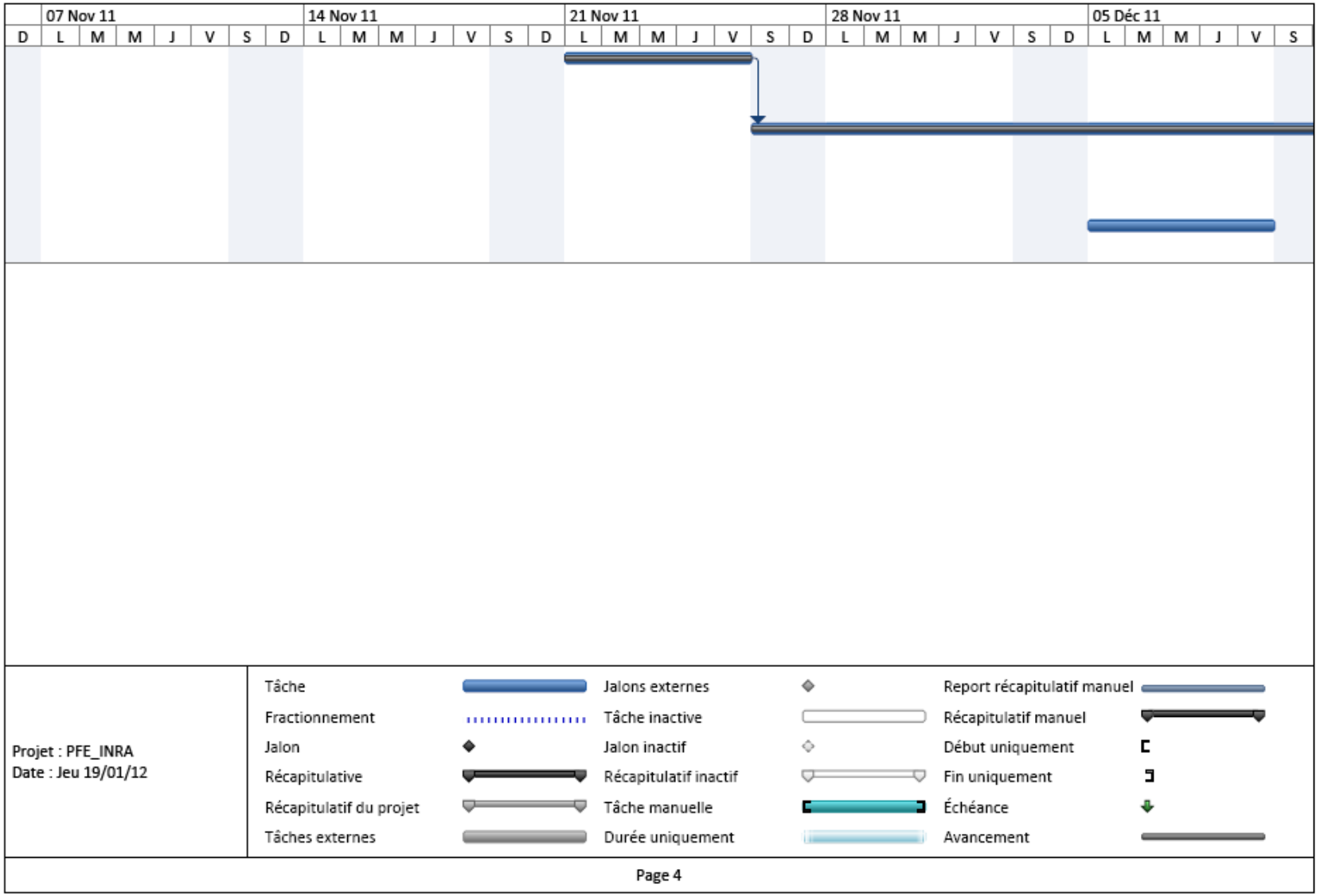
17 Oct 11							24 Oct 11							31 Oct 11						
L	M	M	J	V	S	D	L	M	M	J	V	S	D	L	M	M	J	V	S	D

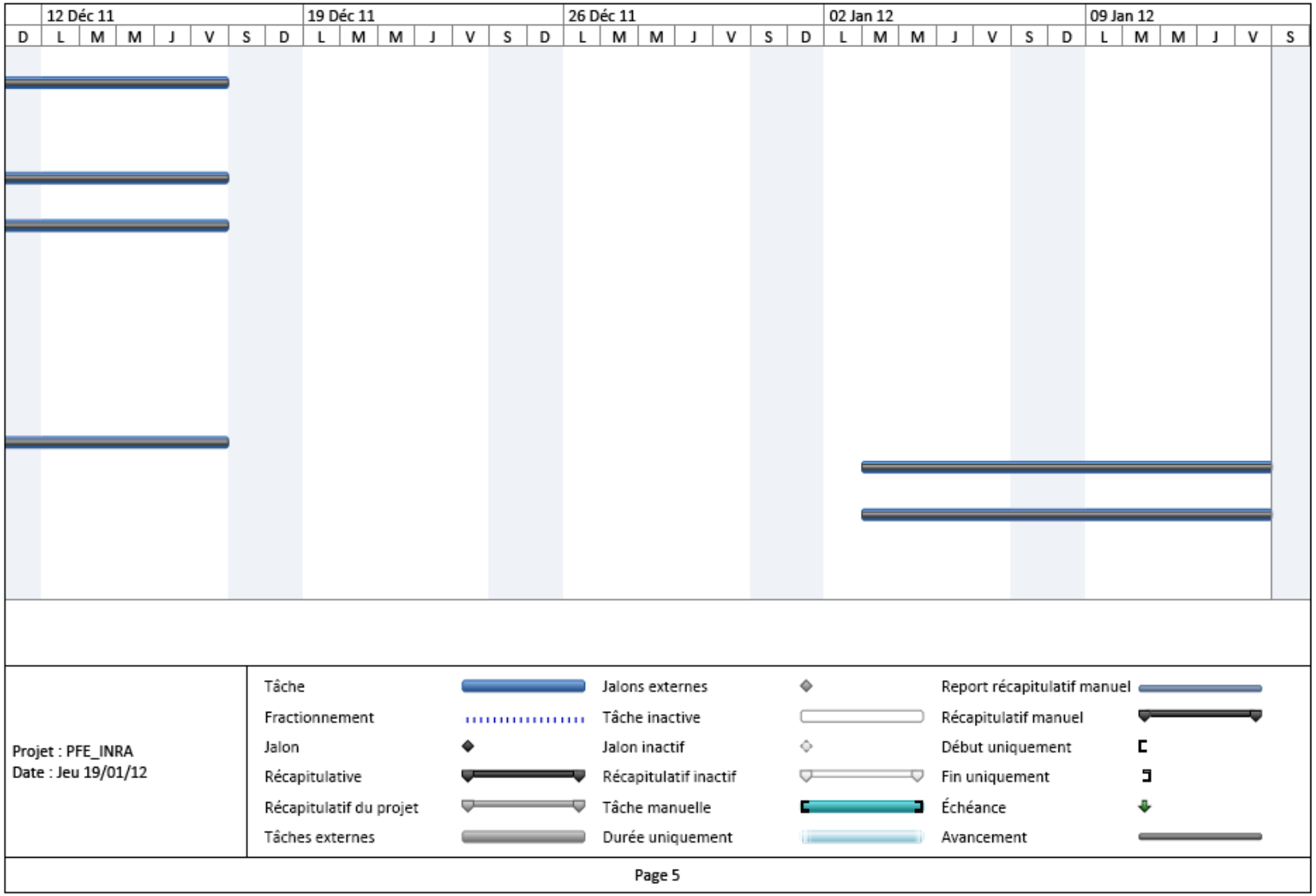
Projet : PFE_INRA
Date : Jeu 19/01/12

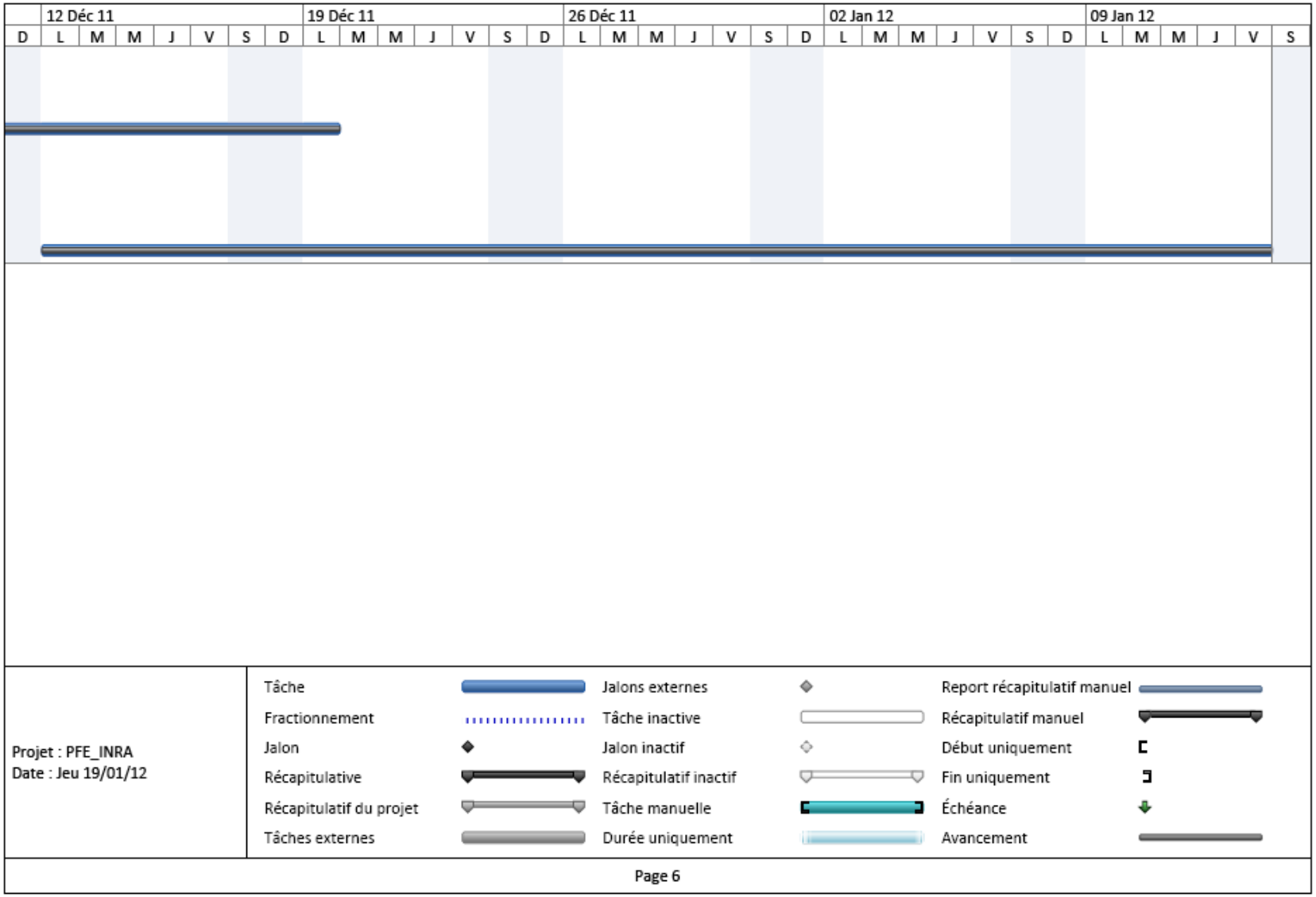
Tâche		Jalons externes		Report récapitulatif manuel	
Fractionnement		Tâche inactive		Récapitulatif manuel	
Jalon		Jalon inactif		Début uniquement	
Récapitulative		Récapitulatif inactif		Fin uniquement	
Récapitulatif du projet		Tâche manuelle		Échéance	
Tâches externes		Durée uniquement		Avancement	

Page 2


07 Nov 11
14 Nov 11
21 Nov 11
28 Nov 11
05 Déc 11
D L M M J V S D L M M J V S D L M M J V S D L M M J V S D L M M J V S
Projet : PFE_INRA
Date : Jeu 19/01/12
Tâche
Fractionnement
Jalon
Récapitulative
Récapitulatif du projet
Tâches externes
Jalons externes
Tâche inactive
Jalon inactif
Récapitulatif inactif
Tâche manuelle
Durée uniquement
Report récapitulatif manuel
Récapitulatif manuel
Début uniquement
Fin uniquement
Échéance
Avancement
Page 3


07 Nov 11
14 Nov 11
21 Nov 11
28 Nov 11
05 Déc 11
D L M M J V S D L M M J V S D L M M J V S D L M M J V S D L M M J V S
Projet : PFE_INRA
Date : Jeu 19/01/12
Tâche
Fractionnement
Jalon
Récapitulative
Récapitulatif du projet
Tâches externes
Jalons externes
Tâche inactive
Jalon inactif
Récapitulatif inactif
Tâche manuelle
Durée uniquement
Report récapitulatif manuel
Récapitulatif manuel
Début uniquement
Fin uniquement
Échéance
Avancement
Page 4


12 Déc 11
19 Déc 11
26 Déc 11
02 Jan 12
09 Jan 12
D L M M J V S D L M M J V S D L M M J V S D L M M J V S D L M M J V S
Projet : PFE_INRA
Date : Jeu 19/01/12
Tâche
Fractionnement
Jalon
Récapitulative
Récapitulatif du projet
Tâches externes
Jalons externes
Tâche inactive
Jalon inactif
Récapitulatif inactif
Tâche manuelle
Durée uniquement
Report récapitulatif manuel
Récapitulatif manuel
Début uniquement
Fin uniquement
Échéance
Avancement
Page 5


12 Déc 11
19 Déc 11
26 Déc 11
02 Jan 12
09 Jan 12
D L M M J V S D L M M J V S D L M M J V S D L M M J V S D L M M J V S
Projet : PFE_INRA
Date : Jeu 19/01/12
Tâche
Fractionnement
Jalon
Récapitulative
Récapitulatif du projet
Tâches externes
Jalons externes
Tâche inactive
Jalon inactif
Récapitulatif inactif
Tâche manuelle
Durée uniquement
Report récapitulatif manuel
Récapitulatif manuel
Début uniquement
Fin uniquement
Échéance
Avancement
Page 6

B. Guide d'installation de Visual Studio

Pour pouvoir utiliser et modifier nos programmes sous Windows, il est nécessaire d'installer le logiciel Visual Studio. Ce logiciel permet de recompiler les projets C et C++ mais il permet surtout de pouvoir charger des DLL avec Matlab.

La version professionnelle de ce logiciel nécessite une licence payante, mais il existe aussi une version Express qui est gratuite. Cette suite logicielle est aussi disponible dans différentes versions en fonction des années de sorties (2005, 2008, 2010).

La version à installer sur un ordinateur sera imposée par la version de Matlab. En effet, il est indispensable que Matlab puisse charger les DLL par le biais de Visual Studio. La liste des compatibilités se trouve sur le site de MathWorks (il faut prendre une version supportée pour les fonctionnalités *"For C and C++ shared libraries"* et *"For loadlibrary"*) :

- http://www.mathworks.fr/support/compilers/R2010b/index.html
- http://www.mathworks.fr/support/compilers/R2011b/win32.html

Pour presque chaque version de Matlab, il existe une version Express de Visual Studio supportée.

A titre d'exemple, nous possédons un Matlab 64 bits en version 2010b et nous avons dû utiliser Visual Studio 2008 Professional (c'est une des seules configurations pour laquelle la version express n'est pas supportée).

MATLAB Product Family – Release 2010b

Compiler	Version	MATLAB For MEX-file compilation and external usage of MATLAB Engine and MAT-file APIs	MATLAB For `loadlibrary`	MATLAB Compiler For C and C++ shared libraries
Microsoft Visual C++ 2010 Express and Windows SDK 7.1 [1] *Available at no charge*	10.0	√		√
Microsoft Visual C++ 2010 Professional	10.0	√		√
Microsoft Visual C++ 2008 Express Edition and Windows SDK 6.1 [2] *Available at no charge*	9.0 [6]	√		√
Microsoft Visual C++ 2008 Professional SP1 and Windows SDK 6.1 [2] [3]	9.0	√	√	√

Téléchargement et installation :

Les versions express se téléchargent directement sur internet :

- Version 2008 : http://go.microsoft.com/?linkid=7729279
- Version 2010 : http://msdn.microsoft.com/fr-fr/gg699327

L'installation est très simple, il suffit de suivre les étapes. Toutefois, si vous utilisez Windows 64 bits et donc Matlab 64 bits, il faut s'assurer qu'il installe bien le compilateur 64 bits.

C. Guide d'installation de CUDA

La bibliothèque CUDA doit être installée pour pouvoir utiliser GPUMat afin de lancer des programmes directement sur des cartes graphiques nVidia (basé sur du langage CUDA).

Téléchargements :

Il est important de prendre les versions 32 bits même si vos configurations sont en 64 bits. En effet, CUDA est à l'heure actuelle encore mal supporté par les environnements 64 bits. Des problèmes peuvent apparaître en cas d'installation de mauvaise version.

Les éléments à installer peuvent être téléchargés avec les liens suivants :

- http://www.nvidia.com/object/thankyou.html?url=/compute/cuda/4_0/toolkit/cudatoolkit_4.0.17_win_32.msi
- http://developer.download.nvidia.com/compute/cuda/4_0/sdk/gpucomputingsdk_4.0.19_win_32.exe

Installations :

Pour commencer, installer *"cudatoolkit_4.0.17_win_32"*. L'installation ne nécessite pas d'explications supplémentaires car il suffit de suivre les étapes.

Enfin pour terminer, installer *"gpucomputingsdk_4.0.19_win_32"*. L'installation ne nécessite pas non plus d'explications supplémentaires car il suffit de suivre les étapes.

D. Guide d'installation de GPUmat

GPUmat permet à l'utilisateur de lancer son programme Matlab sur la carte graphique. Ainsi il faut donc installer GPUmat et CUDA. (Pour installer CUDA merci de se référer à la partie « Guide d'installation de CUDA » en annexe). Nous rappelons que pour pouvoir utilisé le langage CUDA et donc par conséquent GPUmat, l'utilisateur doit avoir une carte graphique Nvidia.

1. Pré-requis

Le « GPUmat User Guide » disponible sur le site de gp-you (voir adresse ci-dessous) explique la marche à suivre pour installer GPUmat.

Sur windows, il est indispensable d'installer « Microsoft Visual C++ 2008 Redistributable Package » . Attention il est absolument nécessaire d'installer Visual studio 2008 (express), dans la même version que Matlab, c'est-à-dire 32 ou 64 bits.

2. Comment installer GPUmat

Pour télécharger GPUmat, il suffit d'aller sur le site :

http://www.gp-you.org/ , de créer un compte et de télécharger la version désirée. L'utilisateur doit télécharger la version correspondant à son Matlab, 32 ou 64 bits.

3. Lancement de GPUmat

Il faut ouvrir Matlab et se placer dans le dossier GPUmat téléchargé précédemment. La commande GPUstart lance GPUmat.

E. Guide d'installation de la librairie Boost

La libraire Boost est utilisée pour multithreader nos programmes C++. Elle doit donc être installée sur tous les PC qu'ils soient Windows ou Linux. Nous allons expliquer comment l'installer pour chaque environnement.

Windows 32 bits :

Si vous utilisez un Windows 32 bits, il faut installer Boost en version 32 bits. Dans ce cas, c'est simplement un exécutable à lancer: http://boostpro.com/download/boost_1_47_setup.exe

Windows 64 bits :

Si vous utilisez un Windows 64 bits, il faut installer Boost en version 64 bits. Pour cela, télécharger les sources sur le site suivant :

http://sourceforge.net/projects/boost/files/boost/1.47.0/boost_1_47_0.zip/download

Placer le dossier *boost_1_47_0* dans *Program Files*. Ensuite, ouvrir un *Invite de commande visual studio* (dans le menu démarrer) et taper les commandes suivantes :

```
cd C:\Program Files (x86)\boost_1_47_0\tools\build\v2\
bootstrap.bat
b2 install --prefix=DIR
cd ..\..\..
tools\build\v2\DIR\bin\bjam.exe --link=static --toolset=msvc address-model=64 --build-type=complete
```

Linux :

Si vous utilisez Linux, il faut télécharger les sources sur :

http://sourceforge.net/projects/boost/files/boost/1.47.0/boost_1_47_0.tar.gz/download.

Puis, compiler les dans la même version que Linux (32 ou 64 bits) afin d'obtenir les librairies compatibles. Cette compilation n'a pas été testée à cause de problèmes de droits à l'ESEO.

F. Guide d'installation R (et ses attributs)

Voici les préalables informatiques à l'installation de R. Tout d'abord, si ce n'est déjà fait, télécharger R pour Windows ou Linux avec le lien suivant : http://cran.cict.fr/

1. Pour Windows

Voici comment préparer son ordinateur Windows à la création de packages :

a) Installation de programmes

On doit premièrement installer les programmes utilitaires suivants si on ne les a pas déjà (R Team, 2005a, annexe E ; http://www.murdoch-sutherland.com/Rtools/) :

- **Perl** : http://www.activestate.com/Products/ActivePerl/Download.html

- Quelques outils Unix appelés **Rtools** (qui proviennent en partie de Cygwin, mais même si Cygwin est déjà installé sur son ordinateur, il est préférable d'installer ces outils) : http://www.murdoch-sutherland.com/Rtools/tools.zip

- **MinGW** : la version la plus récente (février 2006) est actuellement téléchargeable à l'adresse http://prdownloads.sourceforge.net/mingw/MinGW-5.0.0.exe?download (Permet de compiler du code **C**, **C++** et FORTRAN)

Dans notre situation, nous avons installé les programmes suivants :

- ➔ *Installation de Rtools (Rtooles214)*
- ➔ *Installation de ActivePerl 5.12.4.1205*
- ➔ *Installation minGW*

b) Modification du PATH

Il faut ensuite ajouter le répertoire « bin » de R, Rtools, Perl et MinGW. Le PATH doit donc comprendre (par exemple, à adapter selon l'emplacement des répertoires sur son ordinateur):

- C:\Program Files\R\R-2.14.0\bin
- C:\Program Files\R\tools\bin
- C:\Perl\bin
- C:\MinGW\bin

La position de ces répertoires dans le PATH importe peu, à l'exception du répertoire de Rtools qu'il est mieux de placer au tout début.

La marche à suivre pour ajouter des répertoires au PATH varie en fonction de la version de Windows utilisée.

- Sur Windows XP, il faut aller dans :
  - Menu démarrer → Panneau de configuration → Système → Avancé
  - Puis cliquer sur « Variables d'environnement »
  - Dans la section « Variables système », sélectionner PATH

- Cliquer sur « Modifier »
- Ajouter ici les répertoires manquants séparés par des points-virgules (sans espace avant et après les points-virgules)

➢ Sur Windows 7, il faut aller dans :

- Menu démarrer → Panneau de configuration → Système et sécurité → Système → Paramètres de système avancés
- Puis cliquer sur « Variables d'environnement »
- Dans la section « Variables système », sélectionner PATH
- Cliquer sur « Modifier »
- Ajouter ici les répertoires manquants séparés par des points-virgules (sans espace avant et après les points-virgules)

Pour d'autres versions de Windows, on peut se guider avec les explications de la page web http://www.ats.ucla.edu/stat/hlm/faq/path.htm

c) Définition de la variable d'environnement TMPDIR

Pour certaines versions de Windows, il vous sera nécessaire de définir la variable d'environnement TMPDIR en fonction de l'emplacement du répertoire de fichiers temporaires de son ordinateur.

Par exemple, sur Windows XP, il faut aller dans :

- Menu démarrer → Panneau de configuration → Système → Avancé
- cliquer sur « Variables d'environnement »
- Dans la section « Variables système »
- cliquer sur « Nouveau » et entrer les informations suivantes :

Nom de la variable = « TMPDIR », Valeur de la variable = « C:\WINDOWS\TEMP »).

Remarque : Cette dernière étape n'est pas vraiment nécessaire, elle ne sert qu'à régler ce qu'on pourrait juger être des bugs.

d) La DLL MSVCR100d

Enfin, il pourra s'avérer nécessaire de télécharger la DLL suivante afin de faire tourner les DLL en C sous R :

- Cliquez sur le lien : http://www.dll-files.com/dllindex/dll-files.shtml?msvcr100d
- Placer msvcr100d.dll dans le fichier bin de R (ici, C:\Program Files\R\R-2.14.0\bin)

Pour plus d'informations, vous pouvez consulter les liens suivants :

http://www.mat.ulaval.ca/fileadmin/informatique/LogicielR/ProgR_AppelC_Package_Pres.pdf)