PHY-1234

INTRODUCTION À LA PHYSIQUE NUMÉRIQUE [LABORATOIRES]

Notes de Laboratoire
par

Paul Charbonneau
Département de Physique
Université de Montréal

Août 2009

MINI-PRÉFACE

Ces notes contiennent toute la matière couverte dans le cadre du cours PHY-1234, **Hydrodynamique**, offert par le Département de Physique de l'Université de Montréal. Vous devez cependant vous

Table des Matières

Laboratoire 1

Bonjour Monsieur le Professeur

Ceci est le premier de deux laboratoires traitant principalement de divers aspects de base de la programmation en C. Il vous serait probablement profitable d'avoir bien lu le chapitre 1 des notes de cours avant de commencer.

1.1 Objectifs

1. Apprendre à vous brancher sur le réseau informatique de l'université;
2. Apprendre les bases de la gestion de fichiers et de répertoires sous le système d'exploitation LINUX;
3. Apprivoiser un programme d'édition de fichier, vous permettant d'écrire vos propres codes sources;
4. Apprendre à compiler et exécuter un code C simple, et détecter des erreurs de programmation;
5. Apprendre les bases structurelles d'un programme en C: canevas global d'un code, instructions de déclaration, instructions exécutables, types de variables, commentaires, etc.;
6. Apprendre les bases de l'interaction usager-programme (instructions C `scanf`, `printf`, etc.).

1.2 Rapport de laboratoire

Il n'y a pas de rapport à remettre pour ce labo! Mais vous ne sortirez pas d'ici avant qu'on ait vu vos trois codes C exécuter correctement...

1.3 Se brancher sur ESIBAC

ESIBAC est un des systèmes informatiques "public" à l'UdeM, sur lequel vous devrez vous brancher pour faire tous les labs dans ce cours. On vous expliquera en détail comment faire au début de cette session de laboratoire. En bref, un compte est déjà crée pour chacun(e) d'entre vous, et restera actif jusqu'à la fin décembre. Pour plus d'information sur ESIBAC, voir:

`http://www.dgtic.umontreal.ca/esi/esibac.html`

`http://www.bac-esi.umontreal.ca`

Ce qui suit présuppose que vous êtes déjà en mesure de vous brancher sur votre profil via le portail UdeM, et que vous avez défini un mot de passe général. Si ce n'est pas fait, demandez à un des démonstrateurs on vous montrera comment.

À partir des ordis du M-635, malheureusement montés en Windows, la communication sur ESIBAC se fait via une fenêtre dite "terminal". Une fois votre session Windows lancée, cliquez sur le dossier *logiciels avancés*, puis sur *hummingbird*, puis sur *ssh_esilbac1.xs*. Ceci devrait faire apparaitre une fenêtre terminal à l'écran. De cette fenêtre vous avez accès aux compilateur C.

Les fichiers que vous créerez sur votre compte ESIBAC peuvent être accédés de n'importe quel autre ordi ou terminal branché au réseau universitaire public, y compris à la salle d'ordi réservé aux étudiant(e)s du département de physique, soit le G-609 au Pavillon Roger-Gaudry (code d'entrée 3-4-5).

1.4 Fichiers et répertoires

À moins de n'avoir jamais touché à un ordi de votre vie, vous êtes probablement familier(ère)s avec l'idée d'un **fichier** (ou "document"), et de leur organisation en **répertoires** (ou "dossiers", dans l'univers Microsoft). Une organisation intelligente de vos programmes source, fichiers de données, fichiers de sortie, etc., est un saine habitude de travail, et vous aidera grandement à vous retrouver dans vos affaires rendu à la onzième semaine de labo...

La Figure 1.1 illustre ce qui pourrait bien être le contenu du compte de l'usager `milou` à mi-chemin durant le second laboratoire. Sous LINUX le répertoire-mère de l'usager `milou` s'appelera toujours `/home/milou`. Ce répertoire content ici trois répertoires, dont un (`labo2`) qui en contient lui-même un autre, ainsi que divers fichiers, et un autre demeure présentement vide (`labo3`). Le répertoire-mère contient également trois fichiers quelconques en plus des trois répertoires. La gestion de vos fichier exige que vous puissiez créer des répertoire, y déplacer des fichiers, en éliminer les fichiers désuets, etc. Vous avez peut-être déjà effectué ce genre d'opérations via la souris, directement ou à l'aide de menus. Dans l'environnement de type "terminal" fourni par ESIBAC, vous aurez à utiliser les commandes de bases du système d'exploitation LINUX. Le Tableau 1.1 en liste les plus importantes; commencez par mémoriser celles-là, et vous apprendrez graduellement les autres "sur le tas".

Table 1.1: Commandes de base en Linux

Commande/Usage	Fonction	Notez Bien
`ls` *rep*	liste fichiers dans répertoire *rep*	si *rep* omis, liste répertoire courant
`cd` *rep*	change au répertoire *rep*	si *rep* omis, ramène à `/home/usager`
`cp` *nom*1 *nom*2	crée copie *nom*2 du fichier *nom*1	fichier *nom*1 demeure
`mv` *nom*1 *nom*2	assigne le nom *nom*2 au fichier *nom*1	*nom*1 disparait
`cp` *nom*1 *rep*	copie fichier *nom*1 dans *rep*	*rep* doit être dans rép. courant
`mv` *nom*1 *rep*	déplace *nom*1 dans répertoire *rep*	*rep* doit être dans rép. courant
`mv` *rep*1 *rep*2	déplace répertoire *rep*1 dans *rep*2	*rep*2 doit être dans rép. courant
`rm` *nom*1	efface le fichier *nom*1	
`more` *nom*1	liste fichier *nom*1 à l'écran	
`mkdir` *rep*	crée répertoire *rep* dans rép. courant	
`rmdir` *rep*	efface le répertoire *rep*	*rep* doit être vide
`pwd`	liste le nom du répertoire courant	

Notez bien que plusieurs de ces commandes sont définies de manière relative au répertoire courant. Ainsi, une fois branché l'usager `milou` a deux options pour examiner le contenu de son répertoire `labo1`:

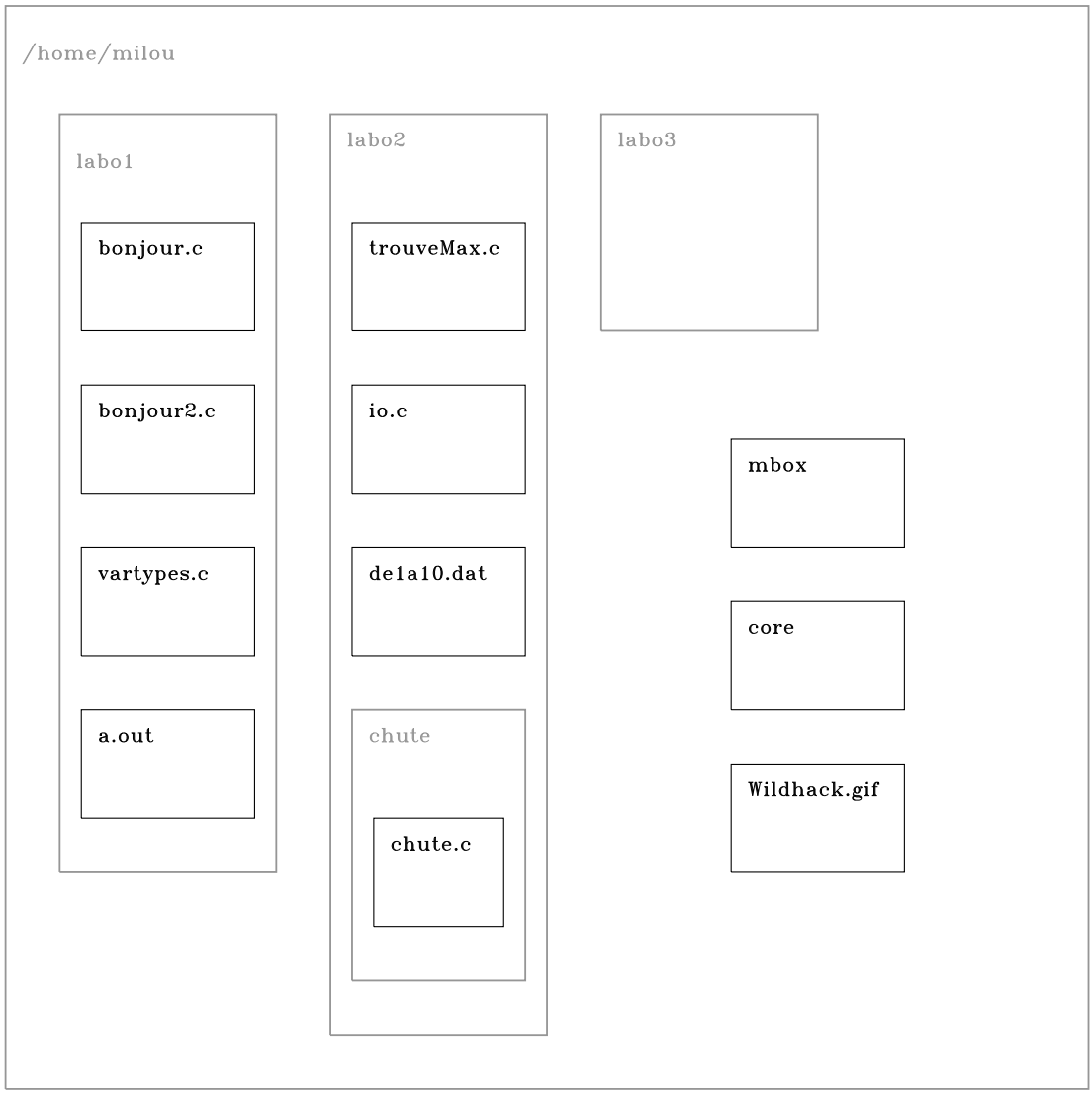

Figure 1.1: Structure typique des fichiers et répertoires de l'usager `milou`, au moment du second laboratoire. Les encadrés gris représentent des répertoires, et ceux en noir des fichiers. Au moment où vous vous branchez sur votre compte, vous atterissez immédiatement et automatiquement dans un répertoire appellé `/home/usager`, où "`usager`" est votre propre identificateur personnel (`milou` dans le cas présent; pour vous, ce sera un code alphanumérique commençant par `p0...`). Ici ce "home" contient trois répertoires, eux-même contenant des fichiers et/ou d'autres répertoires (comme `labo2`, qui ici contient un répertoire appelé `chute`, qui lui-même contient un fichier appelé `chute.c`), ainsi que trois fichiers individuels. Une série de commande Linux, décrites au Tableau 1.1, vous permet de gérer fichiers et répertoires (création/destructions de répertoires, copie et déplacement de fichiers, etc).

1. `cd labo1` *return* `ls` *return*

2. `ls labo1` *return*

La commande `cd` permet de "descendre" dans la structure des répertoires; pour "remonter"; par exemple, si le répertoire courant est `chute` et qu'on désire se déplacer dans le répertoire `labo1`, on a encore plusieurs options équivalentes:

1. `cd /home/milou/labo1` *return*

2. `cd` *return* `cd labo1` *return*

3. `cd ../../labo1` *return*

Ici le "`../../labo1`" indique qu'on doit remonter de deux niveaux à partir du répertoire courant, soit `chute`→`labo2`→`/home/milou`, puis descendre à `labo1`. Notez ici que le répertoire-cible `labo1` peut être identifié en terme absolu à partir du répertoire `/home/milou` (option 1), ou de manière relative par rapport au répertoire courant (option 3). Le mêmes règles s'appliquent aux autres commandes comme `cp` ou `mv`; ainsi, si on est dans le répertoire `/home/milou`, toutes les instructions suivantes auront comme effet de faire une copie dans le répertoire `labo2` du fichier `bonjour.c` contenu dans le répertoire `labo1`:

1. `cp labo1/bonjour.c labo2/bonjour.c` *return*

2. `cd labo1` *return* `cp ../labo2/bonjour.c` *return*

3. `cd labo2` *return* `cp ../labo1/bonjour.c .` *return*

4. `cp /home/milou/labo1/bonjour.c /home/milou/labo2/.` *return*

Notez dans la troisième option que le second argument de la commande `cp` est un simple point, qui représente un raccourci voulant dire "ici et sous le même nom". La quatrième option illustre comment, en utilisant une identification absolue des fichiers, on peut copier ou déplacer n'importe quoi n'importe où, et à partir de n'importe quel répertoire courant.

Votre première tâche sera donc de vous créer un répertoire sous votre répertoire-mère `/home/usager`, dans lequel vous regrouperez et conserverez tous les fichiers contenant les petits codes C que vous aurez à écrire dans la suite de ce labo.

1.5 Un premier code C

Il s'agit maintenant de faire tourner notre premier code en C. C'est une procédure en trois étapes: (1) écriture du code source en C, (2) compilation en langage-machine, et (3) exécution du programme.

Démarrez le programme d'édition de fichier, en tapant dans la fenêtre ESIBAC la commande:

`kate` *return*

Dans la fenêtre d'édition qui apparaitra alors, tapez, ligne par ligne, le petit code C suivant:

```
#include <stdio.h>
int main(void)
{
  /* Ce programme dit "bonjour" a l'ecran */
  printf ("Bonjour Monsieur le Professeur\n") ;
}
```

La ligne `#include <stdio.h>` indique que le programme qui suit utilisera une ou plusieurs fonctions ou commandes prédéfinies contenue dans la librairie `stdio.h`, une des nombreuses librairies standards du langage C. Certains compilateurs tolérent l'omission de cette ligne, mais ce sera une bonne habitude de toujours l'inclure, afin d'éviter des inconsistances si vous travaillez sur différents ordis avec les même codes sources. L'instruction `int main(void)` identifie ce qui suit entre les parenthèses curvilignes {...} comme étant le **programme principal**, par opposition à une **fonction**, concept avec lequel nous ferons connaissance la semaine prochaine (et qui fournira l'occasion de clarifier ce que veulent dire les `int` et `(void)`...). Ici le programme ne contient que deux lignes d'instruction, la première étant une ligne de commentaires, délimitée par les `/* ... */`, qui ne fera absolument rien au moment de l'exécution, et une ligne d'instruction `printf`, dont l'exécution écrira à l'écran tout ce qui est contenu entre les `" ... "`, sauf le `\n` qui produit un saut de ligne final.

Maintenant sauvegardez le fichier en cliquant sur l'icône approprié au haut de la fenêtre d'édition. Pour un nouveau fichier, le programme d'édition vous demandera de spécifier un nom à assigner au fichier. Le code source doit porter un nom qui l'identifiera de manière unique parmi les autres fichiers pouvant être contenus dans le répertoire de travail. Traditionnellement un code source en langage C se terminera par le suffixe "`.c`". On pourrait par exemple appeler `bonjour.c` le fichier contenant le code C ci-dessus.

Pour compiler ce code, revenez à la fenêtre ESIBAC et dans le même répertoire contenant le fichier source, tapez:

`gcc bonjour.c` *return*

Ceci créera dans le répertoire courant un **fichier exécutable** appelé par défaut `a.out`, qui est par la suite exécuté en tapant simplement

`./a.out` *return*

Ceci devrait produire à l'écran la sortie suivante:

```
Bonjour Monsieur le Professeur
```

Celà n'a pas l'air de grand chose, mais ce que vous avez accompli à date est déjà très substantiel: vous brancher, écrire un code source, le sauvegarder sur disque, le compiler, et l'exécuter. Bravo!

1.6 Un second code C

On continue. Créez maintenant un second code source qui a l'air de ceci:

```
#include <stdio.h>
int main(void)
{
  /* Ce programme vous dit "bonjour" a l'ecran */

  /* Declarations ------------------------------ */
  char votreNom[30] ;

  /* Executables ------------------------------- */

  printf ("SVP tapez votre nom : ") ;
  scanf  ("%s", &votreNom) ;
  printf ("Bonjour %s\n", votreNom) ;
}
```

Le code compte maintenant trois instructions exécutables, qui seront exécutées l'une après l'autre, selon l'ordre dans lequel elles apparaissent dans le code source. L'instruction `printf` vous connaissez déjà; l'instruction `scanf` fait l'inverse, elle lit quelquechose que l'usager tape

à l'écran au moment de l'exécution. Notez que chaque ligne d'instruction se termine par un point-virgule "`;`", qui indique explicitement la fin d'une instruction. Ceci permet de combiner plusieurs instructions courtes sur la même ligne, ou d'étaler une très longue instruction sur plusieurs lignes. Ceci ne change absolument rien à l'exécution du code, mais peut en améliorer la lisibilité.

Toujours dans le département des nouveautés, notez qu'ici on définit une variable appelée `votreNom`, de type caractère et pouvant contenir ici un maximum de 30 caractères (instruction commençant par `char`). Il est obligatoire en C de regrouper toute les définitions de variable au début du code, avant les instructions exécutables. Cette variable aurait pu avoir n'importe quel nom (`maman`, `bozo`, `proutschniaque`,...) sans affecter l'exécution, tant que le même nom de variable apparait aux instructions de déclaration et d'exécution. Il est cependant d'usage de donner aux variables des noms reliés à leur sens dans la logique du programme. Notez également qu'en C le nom d'une variable:

1. peut être composé des 26 caractères de base de l'alphabet, des chiffres de 0 à 9, ou du caractère "_"; cependant, pas de caractères accentués du clavier français!

2. ne doit pas cependant commencer par un chiffre; ainsi, `2freresDupondt` n'est pas un nom de variable légal en C;

3. ne peut faire plus de 32 caractères; en utiliser plus de 32 ne cause pas d'erreur de compilation, mais les surnuméraires sont perdues.

4. ne peut contenir d'espaces blancs;

5. distingue les majuscule des minuscules; ainsi, les variables `votreNom`, `VotreNom`, et `votrenom` sont considérées par le compilateur comme trois entités distinctes, devant chacune être déclarée, initialisée, etc.

Il faut finalement noter que le C réserve certaines chaines de caractères pour définir des **mots-clefs** qui jouent un rôle spécifique au niveau de la programmation; si il vous vient la brillante idée de définir une variable ou une fonction ayant le même nom, il pourrait vous arriver des choses É-POU-VAN-TA-BLES !! Voici une liste alphabétique de ces mots-clefs à éviter:

```
auto, break, case, char, const, continue, default, do, double, else, enum,
extern, float, for, goto, if, int, long, register, return, short, signed,
sizeof, static, struct, switch, typedef, union, unsigned, void, volatile,
while.
```

Revenant au petit code ci-dessus, examinez bien la seconde instruction `printf`, en la comparant attentivement avec la première. Le "`%s`" apparaissant entre les guillemets indique que la variable à imprimer, ici `votreNom`, est une chaine de caractères ("`s`" pour "string"). Nous verrons sous peu qu'il existe d'autres descripteurs de format pour d'autres types de variables.

Finalement, vous noterez que deux lignes vides ont été insérées pour démarquer les parties déclaration et exécution du code. Ce lignes ne sont pas vues à la compilation, en ne font qu'améliorer la lisibilité du code pour l'usager.

Compilez et exécutez ce programme. Une fois convaincu(e) que tout fonctionne comme prévu, expérimentez un peu:

1. Enlevez un des point-virgules quelquepart; essayez de compiler et exécuter.

2. Introduisez une faute de frappe dans le nom de la variable `votreNom` dans la section déclaration; essayez de compiler et exécuter.

3. Remplacez le "`%s`" par "`%f`" dans la seconde instruction `printf`; essayez de compiler et exécuter.

4. Enlevez le "`[30]`" dans la déclaration de la variable `votreNom`; essayez de compiler et exécuter.

5. Déplacez la seconde instruction `printf` au début de la section exécutable (i.e., *avant* l'instruction `scanf`); essayez de compiler et exécuter.

Ces divers exemples devraient vous faire réaliser qu'il y a trois types distincts "d'erreurs" qui peuvent se produire:

1. **Erreur de compilation**: votre code ne respecte pas les règles syntaxiques du langage C; le compilateur stoppe sans produire de fichier exécutable;

2. **Erreur d'exécution**: la syntaxe de votre code est valide; mais quelquechose dans la logique du code conduit à une erreur au moment de l'exécution.

3. **Erreur conceptuelle:** le programme compile et exécute sans produire de message d'erreur, mais le résultat est fautif.

Avec un peu d'expérience, le premier type d'erreur devient habituellement facile à retracer à partir des messages d'erreur produits au moment de la compilation. Les erreurs d'exécution peuvent l'être pas mal moins, et les erreurs de type conceptuel encore moins. Notez qu'une erreur durant l'exécution produira habituellement un fichier appelé `core` dans le répertoire où se trouve l'exécutable `a.out`. Ce fichier contient une image de la mémoire du processeur au moment de l'arrêt du program, et peut être utilisé en entrée à un programme dit de "déverminage" (ou "debugger" en bon anglais). Nous ne ferons pas usage de dévermineur dans le cadre de ce cours, mais ce serait une bonne habitude de toujours éliminer les fichiers `core` suite à une erreur d'exécution, car ils peuvenmt parfois avoir une taille susbtantielle.

1.7 Un troisième code C

Le dernier petit code C avec lequel vous travaillerez dans le cadre de ce premier labo vise à vous faire explorer les comportements parfois particuliers de divers opérateurs arithmétiques agissant sur divers types de variables. Relancez votre éditeur, et tapez le code suivant:

```
#include <stdio.h>
int main(void)
{
  /* Ce programme joue avec les types de variables */

  /* Declarations ------------------------------- */
   int   i, j, k ;
   double x, y, z ;

  /* Executables -------------------------------- */
  i=3 ; j=7 ; k=5 ;
  x=3.; y=7.; z=5.;

  printf ("Multiplication  i*j = %d\n", i*j) ;
  printf ("Division        i/j = %d\n", i/j) ;
  printf ("Division        j/i = %d\n", j/i) ;
  printf ("Multiplication  x*y = %f\n", x*y) ;
  printf ("Multiplication  x*y = %e\n", x*y) ;
  printf ("Combinaison   x+y/z = %f\n", x+y/z) ;
  printf ("Combinaison (x+y)/z = %f\n", (x+y)/z) ;
  printf ("Combinaison   x/y*z = %f\n", x/y*z) ;
```

```
  printf ("Combinaison x/(y*z) = %f\n", x/(y*z)) ;
  printf ("i Modulus j         = %d\n", i%j) ;
  printf ("j Modulus i         = %d\n", j%i) ;
  printf ("j Modulus i         = %f\n", j%i) ;
  printf ("Division        x/y = %d\n", x/y) ;
}
```

Les lignes d'instruction dans la section déclarations qui commencent par `int` et `float` identifient les variables dont les nom suivent comme étant des entiers et des réels, respectivement. Une fois compilé et exécuté, ceci devrait produire la sortie suivante:

```
Multiplication  i*j = 21
Division        i/j = 0
Division        j/i = 2
Multiplication  x*y = 21.000000
Multiplication  x*y = 2.100000e+01
Combinaison   x+y/z = 4.400000
Combinaison (x+y)/z = 2.000000
Combinaison   x/y*z = 2.142857
Combinaison x/(y*z) = 0.085714
i Modulus j         = 3
j Modulus i         = 1
j Modulus i         = 0.000000
Division        x/y = 1071345078
```

(où l'entier résultant de la dernière division pourrait bien avoir une autre valeur que celle reporduite ici, dépendant de l'ordi et compilateur utilisés). Il y a plusieurs choses à remarquer et comprendre ici:

1. La multiplication est effectuée via le symbole "`*`", et la division via "`/`"; ces opérateurs peuvent agir soit sur des entiers, soit sur des réels.

2. Les opérations arithmétiques s'effectuent de gauche à droite, mais la multiplication et la division ont priorité sur l'addition et la soustraction; comparez et comprenez bien les résultats du calcul de `x+y/z` versus `(x+y)/z`, et de `x/y*z` versus `x/(y*z)`, et comment l'usage judicieux des parenthèses permet de contourner ces conventions.

3. La division de deux variables déclarées comme des entiers produit un résultat étant lui-même un entier, donc tout reste de la division est perdu (ici, $3/7 = 0$ et $7/3 = 2$).

4. Le reste de la division de deux entiers peut être calculé explicitement avec l'opérateur "`%`", dit "modulo". Assurez vous de bien comprendre ici pourquoi $3\%7 = 3$ et $7\%3 = 1$.

5. À chaque type de variable doit correspondre le bon **descripteur de format**, sinon la sortie peut faire n'importe quoi (regardez bien et comparez les deux dernière ligne de la sortie avec les instructions `printf` les ayant produite).

6. Deux formats de sortie sont disponibles pour les variables réelles, soit la notation décimale habituelle (produite par le descripteur `%f`) et la notation exponentielle (produite via `%e`):

$$\texttt{2.100000e+01} \quad \equiv \quad 2.1 \times 10^1 \quad \equiv \quad \texttt{21.00000}$$

Le Tableau ci-dessous liste les différentes déclarations légales de variables en C. Notez qu'il existe trois types d'entiers et de réels, dépendant du niveau de précision requis dans la représentation numérique (voir chapitre 1 dans les notes de cours). Dans la majorité des cas,

Table 1.2: Types de variables

Déclaration	type	descripteur
`short`	entier "court"	`%d`
`int`	entier	`%d`
`long`	entier "long"	`%d`
`float`	réel	`%f`, `%e`
`double`	réel double précision	`%f`, `%e`
`long double`	réel quadruple précision	`%f`, `%e`
`char`	caractère	`%c`, `%s`

l'usage de `int` pour les entiers et `float` pour les réel devrait suffire (voir cependant le chapitre 2 des notes de cours!), et les types `short` et `long double` ne sont que très rarement utilisés.

Vous aurez peut-être remarqué que la commande `printf` agissant sur des variables de type `float` via les descripteurs de format `%e` et `%f`, ne produit que sept chiffres significatifs en sortie; pour des raisons cosmétiques (ou autres), il est souvent utile de pouvoir contrôler le nombre de chiffres significatifs en sortie. Ceci peut se faire en ajoutant un préfixe numérique aux descripteurs. Par exemple, les instructions suivantes:

```
printf ("Combinaison x/(y*z) = %10.3f\n",x/(y*z)) ;
printf ("Combinaison x/(y*z) = %10.3e\n",x/(y*z)) ;
printf ("Combinaison x/(y*z) = %20.10e\n",x/(y*z)) ;
printf ("Combinaison x/(y*z) = %12.10e\n",x/(y*z)) ;
```

vont produire en sortie:

```
Combinaison x/(y*z) =      0.086
Combinaison x/(y*z) =  8.571e-02
Combinaison x/(y*z) =     8.5714288056e-02
Combinaison x/(y*z) = 8.5714288056e-02
```

La convention des préfixes est bizarroide à souhait (et héritée du FORTRAN); `%10.3e` indique que la sortie est en notation exponentielle compte dix chiffre ou caractères dont 3 sont réservées aux décimales suivant le point; traditionnellement le 10 doit inclure ici le "`.`" et le ("`e+`", donc un descripteur `5.3e` ne produirait pas une sortie valide; de même, essayer d'imprimer le chiffre 100.2 avec le descripteur `5.3f` produirait une sortie incorrecte; on a besoin ici d'au moins 5 caractère/chiffres, puisque le "`.`" compte encore. Et dans les deux cas, si le `float` à imprimer est négatif, le "`-`" doit également être comptabilisé dans le total. La majorité des compilateurs C peuvent contourner ce problème, mais les sorties se retrouvent décalées horizontalement sur la page, comme dans l'exemple ci-dessus; ce genre d'absurdités ne devrait pas trop vous causer de problèmes, à moins d'être quelquepeu maniaque et insister sur le fait que les sorties à l'écran soient toutes joliment alignées, etc.

Vous pouvez maintenant modifier le code ci-dessus et expérimenter avec les divers types de variables, leur impact sur les opérations arithmétiques, l'utilisation des divers descripteurs de format, etc. Faites-vous la main là-dessus une dizaine de minutes, demandez à un des démonstrateur de vous tester, et ensuite voilà, le premier labo est terminé. Caramba!

Lectures supplémentaires:

Notes de cours: Chapitre 1

Delannoy: Chapitres 1, 2 et 3

Laboratoire 2

La parabole de la chute

Ceci est le second de deux laboratoires traitant principalement de divers aspects de base de la programmation en C.

2.1 Objectifs

1. Apprendre à utiliser les différents types d'instructions conditionnelles en C: `if`, `else`, etc.;
2. Apprendre à définir et utiliser des fonctions en C;
3. Apprendre à utiliser les différents types d'instructions de répétition en C: `for`, `while`, etc.;
4. Apprendre à définir et utiliser des tableaux unidimensionnels en C
5. Apprendre à faire un graphique simple "y versus x" à l'aide de la librairie graphique PLPLOT;

2.2 Rapport de laboratoire

Il n'y a encore pas de rapport à remettre pour ce labo! profitez-en, c'est la dernière fois...

2.3 Les structures conditionnelles

Dans bien des situations de programmation que nous rencontrerons dans le cadre de ce cours, nous aurons besoin d'instructions permettant de contrôler l'exécution de certaines instructions ou blocs d'instructions sur la base de critères qui dépendent de la valeur de variables calculées durant l'exécution du code même, et donc qui ne peuvent être posés *a priori* au moment de l'écriture du code source. Ce sont les instructions dites **conditionnelles**. En C, le canevas général d'une instruction conditionnelle simple a la forme:

`if ( condition )` { **instructions** }

tandis qu'une instruction conditionnelle composée aurait comme canevas:

`if ( condition )` { **instructions 1** } `else` { **instructions 2** }

Les instructions conditionnelles peuvent être imbriquées, dans le sens qu'un bloc d'instructions sujet à exécution conditionnelle peut lui-même contenir d'autres instructions conditionnelles impliquant une ou plusieurs instructions `if` et/ou `else`.

Un petit exemple simple sera plus utile qu'une longue description; le code suivant renvoie en sortie à l'écran le plus grand de deux entiers fournis par l'usager au moment de l'exécution, via l'instruction `scanf`:

```
#include <stdio.h>
int main(void)
/* Ce code choisi le plus grand de deux entiers en entree */
{
/* Declarations ---------------------------------------- */
  int n, p, maxi ;
/* Executable ------------------------------------------ */
  printf("donnez deux nombres entiers : ") ;
  scanf ("%d%d", &n, &p) ;
 if (n < p )
   { maxi = p ; }
 else
   { maxi = n ; }
 printf ("le plus grand des deux est : %d\n", maxi) ;
}
```

Ici la condition `(n < p)` peut être vue comme une variable logique qui peut valoir `VRAI` ou `FAUX`; si son évaluation est `VRAI`, alors le premier bloc d'instructions est exécuté. Si elle évalue à `FAUX`, alors c'est le bloc suivant le `else` qui est exécuté. Le tableau suivant liste les divers opérateurs de comparaison disponibles dans le langage C.

Table 2.1: Opérateurs de comparaison

Opérateur	sens mathématique
`==`	égal à
`!=`	pas égal à
`>`	plus-grand-que
`<`	plus-petit-que
`>=`	plus-grand-ou-égal-à
`<=`	plus-petit-ou-égal-à

Ces opérateurs peuvent également servir à comparer des variables de type caractères (déclaration `char`); ceci est rarement utilisé en simulation numérique, sauf peut-être pour la saine gestion des nom de répertoires et fichiers crées automatiquement à l'exécution. Allez voir votre bouquin de C si vous êtes intéressé(e)s à savoir comment ca marche.

Les conditions contrôlant le comportement des instructions conditionnelles peuvent également être composées entre elles, à l'aide des opérateurs logiques `&&` (le AND logique), `||` (le OR logique), et `!` (le NON logique). Les deux "tables de vérité" suivantes illustrent le résultat de l'utilisation de ces opérateurs logiques sur deux conditions A et B qui chacune peuvent valoir `VRAI` ou `FAUX`:

Table 2.2: Opération logique A && B

	$B =$`VRAI`	$B =$`FAUX`
$A =$`VRAI`	`VRAI`	`FAUX`
$A =$`FAUX`	`FAUX`	`FAUX`

Table 2.3: Opération logique A || B

	$B =$VRAI	$B =$FAUX
$A =$VRAI	VRAI	VRAI
$A =$FAUX	VRAI	FAUX

Le code ci-dessus souffre d'une faille évidente; si vous fournissez deux entiers identiques, le code déclarera le second plus grand le premier. Comprenez bien pourquoi, et ensuite votre premier défi est de modifier le code ci-dessus afin qu'il puisse, le cas échéant, produire une sortie du genre:

```
Les deux entiers fournis sont egaux
```

Il y a plusieurs façon d'accomplir ceci; essayez de la faire de manière "élégante", dans le sens que votre solution comporte un minimum d'instructions `if` et/ou `else`.

2.4 Les fonctions en C

Le concept mathématique d'une fonction vous est déjà familier. Conceptuellement, une fonction est une "boite noire" acceptant un ou plusieurs arguments en entrée, en produisant un résultat via une ou plusieurs opération "internes". Par exemple, la fonction trigonométrique

$$\sin\theta$$

calcule la longueur de la projection sur l'axe des y d'un point situé sur le périmètre d'un cercle de rayon unitaire, à un angle θ de l'axe des x mesurée dans le sens antihoraire. Entre autres.

Le code C ci-dessous illustre la définition et l'utilisation d'une fonction en langage C. Ce code définit une fonction acceptant en argument deux entiers, en produisant pour résultat la valeur de celui de ces deux entiers qui est le plus grand. Opérationnellement, ce code fait la même chose que celui considéré précédemment comme exemple d'introduction aux instructions conditionnelles. Comparez avec attention ces deux codes!

```
#include <stdio.h>
int main(void)
{
/* Declarations ------------------------ */
   int trouveMax(int, int) ;
   int n, p, maxi ;

/* Executables ------------------------- */
   printf("donnez deux nombres entiers : ") ;
   scanf("%d%d", &n, &p) ;
   maxi = trouveMax(n,p) ;
   printf ("le plus grand des deux est : %d\n", maxi) ;
}
/* Fonction trouvant le plus grand de deux entiers */
int trouveMax (int a, int b)
{
   int ff ;
   if (a < b )
     { ff = b ; }  /* b est le plus grand */
   else
```

```
    { ff = a ; }  /* a est le plus grand */
  return  ff ;
}
```

Il y a plusieurs choses importantes à noter ici:

1. La fonction (ici appelée `trouveMax`) est une unité fonctionelle indépendante, qui dans le code source apparait à la suite du programme principal, i.e., à l'extérieur des délimiteurs {...} de ce dernier.

2. La section de déclaration du programme principal doit inclure une instruction déclarant à la fois le nom de la fonction, le type de la valeur retournée (un `int`, ou un `float`, etc), ainsi que le nombre et type de ses variables en argument; ainsi, ici l'instruction `int trouveMax(int, int)` indique que la fonction s'appelle `trouveMax`, qu'elle prend deux entiers en argument et calcule un résultat qui est aussi un entier.

3. La fonction même doit définir le même nom et le mêmes nombre et types de variables en argument et en résultat; ainsi, vu la déclaration de `trouveMax` dans le programme principal, l'instruction de définition de la fonction elle-même *doit* prendre une forme du genre `int trouveMax(int a, int b)`, où les *noms* des variables en argument (ici `a` et `b`) peuvent évidemment être autre chose.

4. Les noms de variables définis à l'intérieur de la fonction (ici `a`, `b`, et `ff`) demeurent locaux à la fonction, et n'ont pas besoin d'être identiques aux noms données aux variables passées en argument et reçus en résultat à l'appel de la fonction dans le code principal (soit ici `n`, `p`, et `maxi`). C'est l'ordre (n, p) des arguments dans l'appel à la fonction qui établit les associations $a \equiv n$, $b \equiv p$.

5. Une instruction de type `return` doit ici être incluse à la fin de la fonction, de manière à renvoyer l'exécution au programme principal, et déterminer laquelle des variables locales est renvoyé comme résultat à l'évaluation de la fonction (ici c'est `ff`).

6. Certaines fonctions, dites "fonctions-action", ne renvoient aucune valeur au programme principal `main`; c'est le cas, par exemple de la fonction `printf`, donc l'action consiste à envoyer à l'écran l'argument qui lui est fourni.

7. Une fonction peut être invoquée directement à l'intérieur d'une expression, ou de l'argument d'une autre fonction; ainsi, dans le programme principal (`main`) ci-dessus on aurait pu éviter la définition de la variable `maxi` en écrivant simplement:

   ```
   printf("le plus grand des deux est : %d\n", trouveMax(n,p) ) ;
   ```

Une fonction peut appeller d'autres fonctions, et le langage C permet même qu'une fonction s'appelle soi-même. Ceci s'appelle une **récursion**.

L'erreur la plus commune dans l'écriture d'un code C incluant des fonctions est d'introduire une ou plusieurs incompatibilités dans le nombre et type des variables en argument ou en sortie à l'évaluation de la fonction. Vous pouvez (lire: devriez!) vous amuser à un moment ou un autre à coder le programme ci-dessus, et changer un `int` pour un `float`, le nombre de paramètres en argument, etc., et voir lesquelles de ces erreurs seront détectées par le compilateur, et lesquelles se manifesteront seulement à l'exécution.

Le Tableau ci-dessous donne une liste de fonctions prédéfinies en C que vous trouverez probablement utiles. Le fait que ces fonction soient prédéfinies implique que vous n'avez pas à les déclarer explicitement dans vos code; cependant, certaines de ces fonctions sont regroupées en **librairies**, et dépendant du détail de l'installation du compilateur C il est possible que ces librairies doivent être explicitement chargées. Par exemple, autant sur mon Mac que sur le

Table 2.4: Quelques fonctions mathématiques prédéfinies du langage C

Fonction	sens mathématique	Notez bien
`sqrt(`x`)`	$\sqrt{x}$	$x \geq 0$.
`pow(`x, y`)`	x^y	
`log(`x`)`	$\ln(x)$	$x > 0$.
`log10(`x`)`	$\log_{10}(x)$	$x > 0$.
`exp(`x`)`	e^x	
`fabs(`x`)`	$\|x\|$	
`ceil(`x`)`	Partie entière de x	
`sin(`x`)`	$\sin(x)$	x en radian
`cos(`x`)`	$\cos(x)$	x en radian
`tan(`x`)`	$\tan(x)$	x en radian
`asin(`x`)`	$\arcsin(x)$	résultat en radian
`acos(`x`)`	$\arccos(x)$	résultat en radian
`atan(`x`)`	$\arctan(x)$	résultat en radian
`sinh(`x`)`	Sinus hyperbolique	
`cosh(`x`)`	Cosinus hyperbolique	
`tanh(`x`)`	Tangente hyperbolique	
`floor(`x`)`	Plus grand entier $< x$	résultat `int`, x est un réel
`ceil(`x`)`	Plus petit entier $> x$	résultat `int`, x est un réel

(vieux) système Unix avec lequel je teste tous les petits codes de ces labos, pour avoir accès aux fonctions de type mathématique je dois inclure en toute première ligne de mon code source:

```
#include <math.h>
```

et, au moment de la compilation, cette librairie doit être explicitement chargée:

```
gcc monCode.c -lm
```

Notons finalement que la très fameuse librairie < `stdio.h` > est chargée automatiquement par tous les compilateurs C auxquels vous n'aurez jamais à faire face, bien qu'elle doivent être explicitement incluse via une instruction `#include` en en-tête au code source.

2.5 Les structures de répétition

En programmation scientifique, la **répétition** d'une instruction ou bloc d'instructions est un des éléments de programmation le plus courant. Le langage C permet de définir deux types de boucles, en fonction de la manière dont le contrôle de la répétition est effectué.

2.5.1 Boucles conditionelles: les instructions `while` et `do...while`

Une boucle conditionnelle est structurée selon des canevas ressemblant fort aux instructions conditionnelles; les deux principaux canevas diffèrent au niveau de l'ordre dans lequel la condition est évaluée, par rapport aux instructions qu'elle contrôle. La structure de base prend la forme:

```
while ( condition ) { instructions }
```

Par exemple, le petit bout de code suivant écrirait en sortie les entiers de un à dix:

```
k=0 ;
while ( k <= 9 )
  {
    k=k+1 ;
    printf ("%d\n", k) ;
  }
```

On peut aussi tester la condition après le bloc d'instructions:

`do` { **instructions** } `while (` **condition** `)`

L'équivalent du petit bout de code ci-dessus, produisant la même sortie, serait:

```
k=0 ;
do {
    k=k+1 ;
    printf ("%d\n", k) ;
   }
while ( k <= 9 ) ;
```

Comparez attentivement ces deux bouts de code; notez et comprenez bien que:

1. À la sortie des boucles, la variable `k` vaudra 10 dans les deux cas;
2. Cependant, dans le second cas, le bloc d'instruction sera toujours exécuté au moins une fois, quelle que soit la valeur à laquelle `k` est initialisée;
3. Le point-virgule suivant la condition est essentiel dans le second cas, puisqu'il indique ici la fin de boucle.

Notez ici le danger qu'une boucle se mette à tourner sans jamais stopper, si la condition de contrôle ne se retrouve jamais satisfaite, suite à une erreur conceptuelle ou de codage. Nous verrons en temps et lieux comment parer à une telle éventualité.

2.5.2 Boucles inconditionelles: l'instruction `for`

Un moment de réflexion devrait vous convaincre que les deux boucles conditionnelles ci-dessus sont, fondamentalement, des boucles **inconditionnelles** puisque que l'on a décidé *a priori*, au moment de l'écriture du code source, que l'on voulait écrire en sortie les entiers de 1 à 10, plutôt que de 3 à 17 ou de 1 à 31415926536. Dans une telle situation on peut utiliser une structure de boucle dite inconditionnelle. Le canevas d'une telle boucle a la forme:

`for (` **décompte** `)` { **instructions** }

où **décompte** est une série d'instruction contrôlant le comportement de la boucle. Par exemple, l'équivalent du bout de code ci-dessus s'écrirait maintenant comme:

```
for ( k=1 ; k<=10; k++ )
  {
    printf ("%d", k) ;
  }
```

Notez la série de trois instructions de décompte, séparées comme il se doit par des point-virgules: (1) initialisation à un d'une variable compteur, ici `k`; (2) une condition de fin de boucle; (3) une instruction d'incrément, `k++` indiquant ici que la variable `k` est incrémentée de un à chaque "tour" de la boucle, raccourci en C pour `k=k+1`.

2.6 Définir et utiliser des tableaux en C

Dans bien des situations il est pratique de pouvoir définir des variables qui sont en fait des vecteurs, ou matrices, ou tenseurs, etc. Du point de vue du langage C, tous ces objets mathématique sont des **tableaux**. Un tableau est défini par un **type**, une **dimension** et une (ou plusieurs) **longueurs**. Par exemple, la position d'un point (x, y, z) en coordonnées cartésiennes peut être définie comme une seule variable réelle (`pos`, disons); dans la section déclaration, on n'a qu'à définir `pos` comme un tableau de type `float`, de dimension 1 et de longueur trois, comme ceci:

```
float pos[3]
```

dans lequel cas on pourrait avoir, dans la section exécutable, les associations

`pos[0]`$\equiv x$, `pos[1]`$\equiv y$, `pos[2]`$\equiv z$.

NOTEZ BIEN qu'en C la numérotation des éléments d'un tableau commence à zéro, pas à un! Donc, si vous avez déclaré votre tableau de longueur 3, son premier élément se voit assigner l'indice `[0]`, le second l'indice `[1]`, et le troisième l'indice `[2]`. Ceci sera une source d'erreur commune; tenter d'invoquer par inadvertance l'élément `pos[3]` causera un **débordement de tableau**, qui se traduira par une erreur à l'exécution. Ceci sera probablement votre plus commune source d'erreur en C, après l'oubli du "`;`"...

Un exemple spécifique vaut facilement une longue discussion; le petit canevas de code C ci-dessous montre comment créer et remplir un tableau de dimension 1, qui contiendra les valeurs d'une variable —disons le temps, pour prendre un exemple comme ça au hasard— où chaque valeur est plus grande que la précédente par le même incrément, comme dans le cas d'une maille équidistante:

```
#include <math.h>
#define N 11          /* longueur du tableau */
int main(void)
{
/* Declarations ------------------------------------------- */
  int   k ;
  float t[N] ;
/* Executable --------------------------------------------- */
/* calculer une maille equidistante entre t=0 et t=10 */
    for (k=0 ; k<N ; k++) {
      t[k]=10.*fabs(k)/(N-1) ;
    }
}
```

Je vous laisse vérifier, par ajout d'une instruction `printf` judicieusement formattée, qu'à la sortie de la boucle, le tableau `t` contiendra les valeurs:

$$[0., 1., 2., 3., 4., 5., 6., 7., 8., 9., 10.]$$

Premier point extrêmement important: en C, les dimensions des tableaux *doivent* être fixées au moment de la compilation, et donc spécifiées dans la section déclaration. En C, **on ne peut pas** utiliser une variable calculée à l'exécution pour spécifier la longueur d'un tableau; si cette longueur n'est pas connue *a priori*, on doit alors spécifier une taille maximale dans la section déclaration, et il serait sage d'inclure alors dans la partie exécutable un ou plusieurs tests s'assurant qu'il n'y aura pas de débordement de tableau. Remarquez ensuite ici une nouveauté, soit l'utilisation d'une instruction `#define`, placées en en-tête au programme même; spécifie à la compilation les valeurs numériques de la variable `N` utilisée pour spécifier la dimensions des divers tableaux déclarés. On aurait tout aussi bien pu omettre ces définitions; mais alors la ligne de déclaration aurait du avoir la forme:

```
float t[11] ;
```

L'avantage de l'utilisation des instructions `#define` est qu'un seul changement dans la valeur de `N` assignée par l'instruction affecte automatiquement tous les tableaux dont une ou plusieurs dimensions sont déterminées par `N` et, pour le code tel qu'écrit ci-dessus, ajuste automatiquement le nombre d'itérations effectuées par la boucle ainsi que la taille de l'incrément. Notez finalement que, puisque les dimensions du tableau sont fixées à la compilation du code, on utilise ici une structure de répétition inconditionnelle (`for` ...).

L'utilisation de majuscules pour les noms de variables dont les valeurs sont spécifiée dans des instructions `#define` est facultative, mais c'est une bonne habitude de programmation qui permet, d'un oup d'oeil au code source, de détecter les "vraies" variables de celles initialisées à la compilation via des instructions `#define`.

Dans le cas d'un tableau de faible longueur, comme ci-dessus, il est également possible de définir explicitement le contenu du tableau en une seule ligne d'assignation (dans la partie "déclarations" du programme), comme suit:

```
float t[11] = { 0., 1., 2., 3., 4., 5., 6., 7., 8., 9., 10. }
```

Notez finalement qu'en C (comme en FORTRAN) les opérations arithmétiques sur les tableaux doivent se faire élément par élément; ainsi, si par exemple on veut doubler les valeurs des éléments du tableau `t` et placer ces valeurs dans un tableau `tx2` (préalablement déclaré `float` et de même dimensions que `t`), on doit écrire une boucle du genre:

```
for ( k=0 ; k<N ; k++ ) { tx2[k]=2.*t[k] ; }
```

plutôt que simplement `tx2=2.*t`, comme le permettent certains langages de programmation.

2.7 La trajectoire parabolique d'un corps en chute libre

Vous avez appris au secondaire que pour un mouvement rectiligne, la position x d'un mobile sujet à une accélération uniforme a est donnée par

$$x(t) = x_0 + v_0 t + \frac{1}{2} a t^2 \ , \tag{2.1}$$

où x_0 et v_0 sont les position et la vitesse du mobile à $t = 0$. Dans cette partie du labo nous allons écrire et exécuter un petit code C qui calcule $x(t)$ à une série de valeurs de t croissante, dans le contexte d'un mobile lancé vers le haut à une vitesse initiale v_0, et sujet uniquement à l'action de la gravité. L'idée ici est de calculer la trajectoire du mobile, i.e., x versus t, ainsi que le temps requis pour qu'il retombe à (mais pas plus bas) que son altitude de lancement.

Comme l'expression ci-dessus est parabolique en t, exiger que le mobile se retrouve à la position x_0 revient à exiger que

$$v_0 t + \frac{1}{2} a t^2 \equiv t \left(v_0 + \frac{1}{2} a t \right) = 0 \ ,$$

soit un polynôme à strictement parler quadratique, mais ici homogène, et donc acceptant comme solutions:

$$t = 0 \ , \qquad t = \frac{2 v_0}{a} \ .$$

La première de ces racines est évidemment la condition initiale, et la seconde représente l'intervalle de temps recherché.

Votre défi est maintenant le suivant: écrire un code C qui calcule $x(t)$ à partir d'une condition initiale $x_0 = 0$, $v_0 = 10\,\mathrm{m\ s^{-1}}$ à $t = 0$, pour $a = g = -9.8\,\mathrm{m\ s^{-2}}$. Votre code doit évaluer

l'éq. (2.1) ci-dessus et écrire le résultat en sortie à incrément de temps $\Delta t = 0.01$, stopper une fois $x(t)$ revenu à sa valeur initiale. Vous pourriez (par exemple) structurer ceci autour d'une boucle de type `while`. Vous devez calculer le temps total écoulé depuis le lancement, ainsi que le nombre de pas de temps ayant dus être effectués. De plus, vous devrez emmagasiner les valeurs de t et $x(t)$ dans deux tableaux de longueurs appropriées. Votre premier défi à ce niveau est que vous ne connaissez pas *priori* combien de valeurs de t seront requises pour revenir à $x = 0$, donc vous ne pouvez pas spécifier d'avance la taille requise des tableaux; vous pouvez cependant spécifier une taille maximale (`MNAX`, disons), et comptabiliser dans la boucle `while` le nombre de valeurs calculées. La structure de la boucle pourrait être du genre:

```
#define NMAX 500
...
double t[NMAX], x[NMAX] ;
...
t[0]=0. ;
x[0]=0. ;
n=0     ;
while ( x[n] >= 0 && n < NMAX ) {
  n=n+1 ;
  t[n] = ... ;
  x[n] = ... ;
}
if ( n == NMAX ) { printf( "NMAX atteint\n") ; }
...
```

Remarquez, et comprenez bien, les aspects suivants:

1. L'instruction `#define` régit maintenant la taille des tableaux unidimensionnels, via la valeur de la variable `NMAX`;
2. Une nouvelle variable `n` (type `int`) est initialisée à zéro et incrémentée de un à chaque passage dans la boucle `while`; à la sortie de la boucle, cette variable sera égale au nombre de points à tracer.
3. La condition de fin de boucle inclut une clause vérifiant que le nombre d'itérations de la boucle, mesuré par la variable `n`, demeure inférieur ou égal à la taille maximale `NMAX` des tableaux, telle que définie par l'instruction `define`, histoire de éviter un débordement de tableau. Si cette condition est satisfaite, un avertissement est envoyé à l'écran via un appel à `printf` afind e prévenie l'usager (vous!) que le calcul est probablement incomplet, dans le sens que vous n'avez pas encore atteint atteint $x = 0$ même si la boucle est terminée.

Mesdames et Messieurs, à vos claviers... ! Et montrez le résultat à un des TP-istes avant de continuer.

2.8 Quand ça foire...

Avec des codes incluant des boucles (ou des commandes graphiques, que nous verrons sous peu), il est possible qu'une "fausse manoeuvre" vous laisse dans une situation où le code tourne "dans le vide". Dans une telle situation, pour stopper l'exécution, il suffit de peser simultanément sur les touches "control" et "c" sur le clavier. **NE PAS FERMER LA FENÊTRE ESIBASC EN CLIQUANT SUR LE "X" ROUGE AU HAUT DE LA FENÊTRE !! COMPRIS ?!!**. Ceci tuerait la fenêtre, mais le processus tournant à vide demeure actif sur le système ESIBAC, et donc bouffe des ressources (temps CPU, mémoire, etc) qui peuvent causer un ralentissement général du système, voire même un crash si trop de ces processus fantômes en viennent à co-exister sur le système.

```
#include <stdio.h>
#include <plplot.h>
/* Ce programme est un cavenas de code C effectuant des sorties
   graphiques a l'aide de la librairie PLPLOT; plus precisement,
   ce code porte en graphique un segment de droite reliant deux points */
int main(void)
{
/* Declarations ------------------------------------------------------ */
  float x1, x2, y1, y2 ;
  float xmin, xmax, ymin, ymax ;
/* Executable -------------------------------------------------------- */
  plinit() ;                            /* Initialisation de PLPLOT */

  xmin = 0. ; xmax=2. ;                 /* intervalle en x du graphique */
  ymin = 1. ; ymax=5. ;                 /* intervalle en y du graphique */
  plenv(xmin,xmax,ymin,ymax,0,1) ;      /* Tracage du cadre */

  pllab("Axe x","Axe y","Mon titre") ; /* Titre et annotation des axes */

  x1=1.  ; y1=1.5 ;                     /* Coordonnee du premier point */
  x2=1.5 ; y2=4.  ;                     /* Coordonnee du second point */
  pljoin(x1,y1,x2,y2) ;                 /* Tracage du segment */

  plend() ;                              /* Fermeture de PLPLOT */
}
```

Figure 2.1: Canevas de code C faisant appel à la librairie graphique PLPLOT pour tracer un segment de droite sur un graphique simple.

2.9 Introduction au graphisme en C: PLPLOT

PLPLOT est une librairie de fonctions graphiques écrites en C, et utilisables directement dans tous les codes que vous développerez dans le cadre des labos. Pour un aperçu des capacités de cette librairie, voir la Page Web PLPLOT:

`http://plplot.sourceforge.net`

Vous y trouverez également un manuel de référence complet pour toutes les fonctions PLPLOT:

`http://plplot.sourceforge.net/docbook-manual/plplot-5.9.8.pdf`

L'idée est de bâtir graduellement vos habiletés graphiques au cours des labos, mais même à la fin du cours nous n'aurons touché qu'à une fraction des possibilités de PLPLOT. Pour aujourd'hui, nous nous en tiendrons au plus simple, soit le traçage d'une coube sur sur un graphique classique de type "y versus x". Un exemple valant bien un long blabla, la Figure 2.1 montre un exemple d'un code C (minimal) qui trace un segment de droite reliant deux points (x_1, y_1) et (x_2, y_2) sur un graphique, tel qu'on le voit sur la Figure 2.2. Il y a plusieurs choses à noter ici:

1. Comme pour la librairie de fonctions mathématiques discutée précédemment, la librairie PLPLOT doit être incluse explicitement en en-tête au code source, via l'instruction `#include <plplot.h>`.

2. Toute série de commandes PLPLOT produisant des graphiques dans un code C doit commencer par un appel à la fonction d'initialisation `plinit()`, et se terminer par un appel à la fonction de clôture `plend()`; remarquez que ces deux fonctions sont des fonctions-action, qui ne renvoient aucun résultat numérique au code `main`, et de surcroit ne deman-

dent aucun argument en entrée; leur exécution ne fait qu'effectuer diverses opérations au niveau du système d'exploitation de l'ordi.

3. La fonction `plenv` traces les axes et établit les échelles du graphique; ce dernier couvre ici les intervalles `xmin` $\leq x \leq$ `xmax`, et `ymin` $\leq y \leq$ `ymax`, les bornes étant spécifiées dans le code source même. Les deux derniers paramètres, entiers valant ici 0 et 1, contrôlent le style des axes; nous en explorons d'autres options plus tard, pour l'instant utilisez ces valeurs.

4. La fonction `pllab` ajoute une annotation au bas de l'axe x (première chaine de caractères en argument), à gauche de l'axe y (seconde chaine de caractère en argument), et un titre global au dessus du graphique (troisième chaine de caractères).

5. La fonction `pljoin` trace un segment de droite entre deux points (x_1, y_1) et (x_2, y_2) spécifiés en argument.

Examinez bien les Figures 2.1 et 2.2 et assurez vous de bien comprendre les appels aux fonctions PLPLOTs, et les résultats graphiques qu'ils produisent. Ensuite, il s'agira de tracer la trajectoire parabolique que vous avez calculé précédemment. Pour ce faire, vous avez en fait deux options:

1. Utiliser la fonction `pljoin` à l'intérieur de votre boucle `while` pour connecter successivement chaque paire de points (t_{n-1}, x_{n-1}), (t_n, x_n).

2. Faire un seul appel, après la fin de la boucle, à la fonction `plline`, qui prend en argument les deux tableaux `t` et `x` ainsi que le nombre `n` de points à tracer:

   ```
   plline(n,t,x) ;
   ```

   Une subtilité agacante est que la définition de `plline` présuppose que les deux tableaux 1D en argument ont été définis `double`; vous devez faire de même dans votre code `main`, sinon il y aura incompatibilité entre l'appel à `plline` et sa définition à l'intérieur de la librairie PLPLOT.

À la compilation, vous devez explicitement charger la librairie PLPLOT. Vous devez d'abord spécifier où, sur les disques ESIBAC, le compilateur peut trouver la librairie. Nos collègues de la DGTIC nous ont facilité la tâche à ce niveau, vous n'avez qu'à taper, dans la fenêtre ESIBAC, la commande suivante *avant* la première compilation de votre session de travail:

```
module load plplot
```

Ensuite, losque vous compilez avec `gcc`, il faut spécifier le chargement de la librairie:

```
gcc code.c -lplplotd
```

Notez bien le "d" à la fin du nom de la librairie PLPLOT! Quand vous taperez la commande `./a.out`, une série d'instructions `printf` et `scanf` cachées à l'intérieur de la fonction `plinit` vous demanderont, à l'exécution, de spécifier le type de sortie graphique via un code numérique. Pour une sortie graphique à l'écran, vous devez spécifier la valeur 1. Si tout s'est bien passé, admirez la belle parabole... Une fois la phase d'émerveillement passée, montrez votre beau graphique à un des démonstrateurs, et voilà le second labo terminé!

Lectures supplémentaires:

Notes de cours: Chapitre 1
Delannoy: Chapitres 4 et 5
Numerical Recipes: Section 1.2

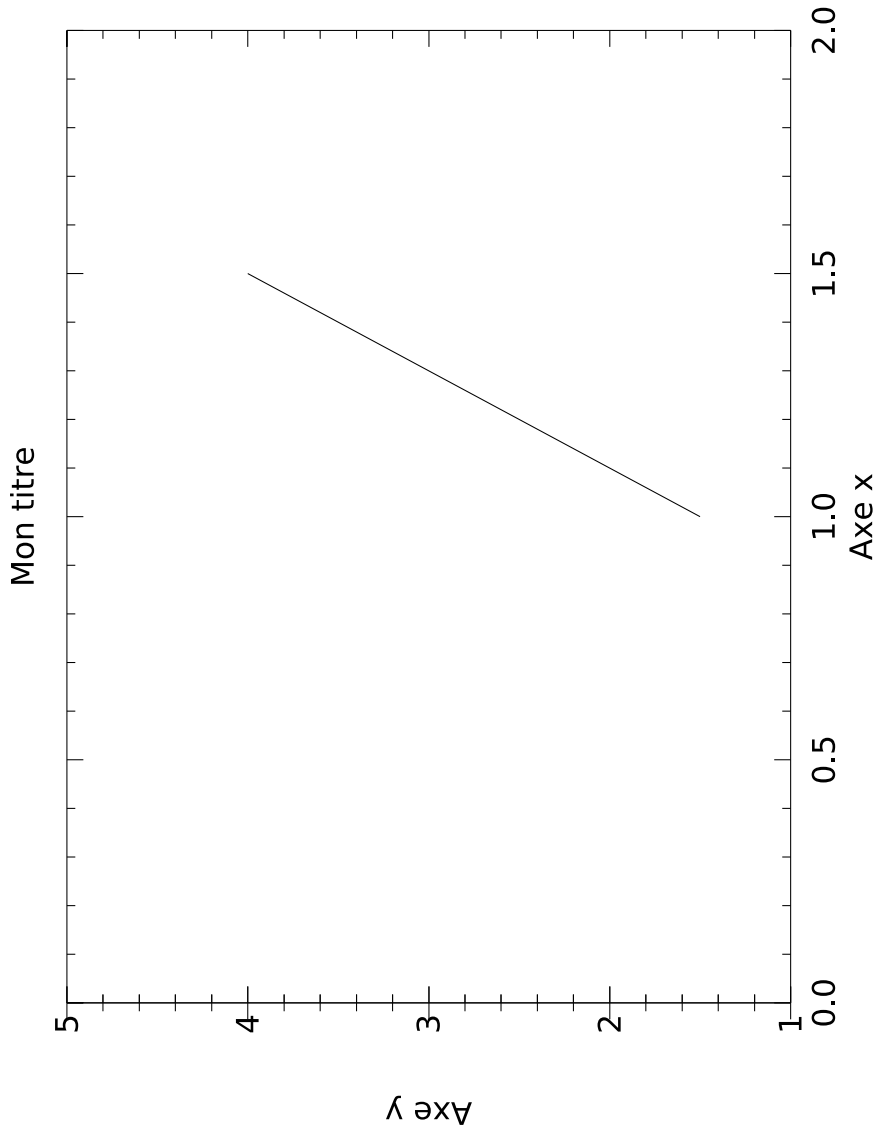

Figure 2.2: Graphique produit par le code source de la Figure 2.1. Ce graphique, tel qu'inclus dans ces notes, a été produit en spécifiant l'option graphique 4 au moment de l'initialisation de PLPLOT (appel à `plinit()`. Lorsque produit en forme fenêtre graphique à l'écran (option 1 au moment de l'initialisation), le graphique sera tracé en rouge sur fond noir... et ne sera pas pivoté de 90 degrés, comme ici.

Laboratoire 3

Potentiel gravitationnel

Ce labo vous fait calculer par intégration numérique le potentiel gravitationnel d'un objet d'étendue finie, plus spécifiquement une tige rectilinéaire. Les développements numérico-théoriques requis se retrouvent à la §2.4 des notes de cours.

3.1 Objectifs

1. Apprendre à effectuer numériquement des intégrales avec des intégrands de forme non-triviale, et à en estimer la précision;
2. Approfondir l'utilisation des fonctions en programmation C;
3. Examiner numériquement le comportement asymptotique du potentiel gravitationnel
4. Apprendre à faire des graphiques log-log avec PLPLOT.
5. Apprendre une autre manière étrange de calculer la valeur de π

3.2 Rapport de Lab

Votre rapport de Lab est à remettre la semaine prochaine **au début** de votre prochain laboratoire. Il doit inclure des réponses (qui peuvent être brèves tant qu'elles sont claires) aux questions posées aux sections 3.3 et 3.7 ci-dessous, ainsi qu'une documentation du test de validation de la §3.6. Vous devez imprimer et inclure des copies de vos codes source C, ainsi que des graphiques produits avec PLPLOT, le cas échéant.

3.3 Intégrales par la méthode du trapèze

On a déjà vu au chapitre 1 une manière bizarroide de calculer la valeur numérique du nombre irrationnel π; en voici une autre:

$$\pi = 2\int_0^1 \frac{1}{\sqrt{1-x^2}}\,\mathrm{d}x \tag{3.1}$$

Nous allons l'utiliser dans ce labo pour tester la précision des intégrales numériques par la méthode du trapèze. Nous allons tout d'abord réécrire l'expression ci-dessus sous la forme:

$$\pi = \int_0^1 f(x)\mathrm{d}x\ , \qquad \text{avec} \qquad f(x) = \frac{2}{\sqrt{1-x^2}}\ . \tag{3.2}$$

ce qui semble trivial mais, dans ce qui suit, nous allons toujours coder nos intégrants sous la forme de fonctions C, alors aussi bien l'anticiper dans nos développements "théoriques"!

Il s'agit ici d'une intégrale définie sur un intervalle $[0,1]$ en x; la première étape sera donc de se définir une maille couvrant uniformément cet intervalle. Toujours en anticipation de ce qui suivra, il sera pratique de le faire en terme d'un nombre de points de maille spécifié dans le code C même; par exemple, la série d'instructions données dans le fragment de code qui suit définit une maille équidistante en x, comprenant N points de maille répartis uniformément entre les bornes $x =$ `xi` et `xo` (voir aussi la §2.4.1 des Notes):

```
#include <stdlib.h>
#define N 100                       /* nombre de points de maille */
int main(void)
{
  float x[N] ;                      /* Tableau 1D qui contiendra la maille */
  float xi=0. ; xo=1. ;             /* bornes de la maille */
  ...                               /* autres declarations/instructions */
  for (k=0 ; k<N ; k++ ) {          /* construction de la maille... */
       x[k]=xi+(xo-xi)*k/(N-1.) ; /* ...un point a la fois */
  }
  ...                               /* la suite du code... */
}
```

Notez que discrétiser ainsi l'intervalle $[0,1]$ à l'aide de N points de maille, le premier et le dernier correspondant aux bornes de l'intégrale, définit $N-1$ sous-intervalles contigus couvrant l'intervalle total. Maintenant, suivez la procédure suivante:

1. Codez l'intégrant de l'éq. (3.2) sous la forme d'une fonction C qui accepte un paramètre en argument, soit une valeur de x, et retourne l'évaluation de la fonction $f(x)$ telle que définie ci-dessus

2. Codez la méthode du trapèze pour la solution de l'intégrale donnée par l'éq. (3.2), en utilisant évidemment votre fonction C pour calculer l'intégrant à chaque point de maille (les f_k dans l'éq. (2.45) des notes de cours).

3. Évaluez votre intégrales pour différents nombres de points de maille dans l'intervalle $[0,1]$; par exemple, $N = 3, 10, 30, 100, 300, 1000$.

4. Pour chaque valeur calculée de l'intégrale ($I(N)$, disons), calculez l'erreur par rapport à la solution attendue; on définira ici cette erreur comme:

$$\varepsilon(N) = |I(N) - \pi| \ ;$$

5. Portez en graphique $\varepsilon(N)$ versus N sur un graphique log-log, et vérifiez que l'erreur diminue bien selon $1/N^2$. Dans PLPLOT, vous n'avez qu'à utiliser la valeurs 30 du dernier paramètre dans l'appel à `plenv`. Notez bien que ceci n'affecte que le tracé des échelles sur les axes; vous devez prendre le logarithme des quantités portées en graphique (avec la fonction C `log10`) avant les appels à `pljoin` sur chaque paire successive de $(N, I(N))$. Assurez vous également de ne *pas* poser `xmin`$= 0$ ou `ymin`$= 0$ (ou encore pire, des valeurs négatives!) dans l'appel à `plenv`, parce que sur un graphique log-log ce serait vraiment pas beau...

Dernière manoeuvre: utilisez vos valeurs d'intégrales à $N = 300$ et $N = 1000$ pour obtenir une valeur améliorée (I^*) à l'aide de la méthode de Romberg (voir §2.4.3 des Notes):

$$I^* = \frac{h_2^2 I(300) - h_1^2 I(1000)}{h_2^2 - h_1^2} \ ,$$

où $h_1 = 1./299.$ et $h_2 = 1./999.$ sont les intervalles de maille associés aux intégrales calculées avec $N = 300$ et $N = 1000$, puisque l'intervalle total couvert est de longueur unitaire; de

manière plus générale, on aurait $h = (\texttt{xo} - \texttt{xi})/(N-1)$, car une discrétisation uniforme sur N points de maille définit $N-1$ intervalles contigus. Vous basant sur votre graphique de convergence, estimez quel N serait requis ici pour arriver à la même précision à l'aide de la méthode du trapèze en version "brute".

3.4 Exercice de programmation semi-avancée

Les plus enthousiastes en programmation pourront restructurer leur code de manière à ce que le code principal (`main`) appelle une première fonction (`integre`, disons), acceptant en argument deux bornes d'intégrations et le nombre de points de maille de la discrétisation souhaitée, et retourne la valeur de l'intégrale d'une fonction $f(x)$ entre ces bornes à l'aide de la méthode du trapèze. L'idée de la fonction `integre` est qu'elle fait elle-même appel à une fonction calculant l'intégrant, comme auparavant. Vous aurez donc un programme qui appelle une fonction, qui elle-même appelle une seconde fonction. Ce genre de modularité est la règle dans tout "vrai" code bien structuré.

3.5 Le potentiel gravitationnel

Il s'agit maintenant de calculer une "vraie" intégrale, soit celle décrivant le potentiel gravitationnel d'une tige rectilinéaire de longueur L et de densité uniforme. Pour une masse ponctuelle de grandeur M, vous savez déjà que le potentiel est donné par

$$\Phi(r) = -\frac{GM}{r} \ , \tag{3.3}$$

où $G = 6.67 \times 10^{-11}\,\mathrm{N\ m^2\,kg^{-2}}$ est la constante gravitationnelle de Sir Isaac. Pour une masse d'étendue finie, la stratégie consiste à l'approximer par un très grand nombre de petits éléments de masse considérés ponctuels, en faire la somme des potentiels correspondants. C'est le **principe de superposition**, auquel satisfait le champ gravitationnel (ainsi que les champs électrique et magnétique), qui permet d'aborder le problème de cette façon.

Dans ce laboratoire nous allons nous limiter à un objet géométriquement simple, soit une tige rectilinéaire de longueur L et de section A, faite d'un matériau de densité ρ (unités $\mathrm{kg/m^3}$). La tige est placée le long de l'axe-x dans le plan $[x, y]$, avec le centre de la tige coincidant avec l'origine du système de coordonnées, tel qu'illustré sur la Figure 3.1. La masse $\mathrm{d}m$ d'une petite section de tige de largeur $\mathrm{d}x'$ située à x' est alors simplement donnée par

$$\mathrm{d}m = \rho A \mathrm{d}x' \ .$$

où le produit $A\mathrm{d}x'$ n'est rien de plus que le volume de cette petite section cylindrique. La tâche consiste à calculer le potentiel gravitationnel à un point P situé à une position (x, y) mesurée par rapport à la même origine du système de coordonnées. La distance r entre ce point et notre petite section de tige est donc donnée par

$$r = \sqrt{(x-x')^2 + y^2} \ .$$

L'équivalent de l'éq. (3.3) donnant la contribution $\mathrm{d}\Phi$ de la petite section de tige au potentiel gravitationnel au point (x, y) est alors

$$\mathrm{d}\Phi(x,y) = -\frac{G\rho A}{\sqrt{(x-x')^2 + y^2}} \mathrm{d}x' \ . \tag{3.4}$$

Le potentiel gravitationnel total associée à la distribution de masse dans son ensemble est obtenu en intégrant cette expression sur la longueur de la tige:

$$\Phi(x,y) = \int \mathrm{d}\Phi(\mathbf{x}) = -\rho\, G\, A \int_{-L/2}^{+L/2} \frac{\mathrm{d}x'}{\sqrt{(x-x')^2 + y^2}} \ , \tag{3.5}$$

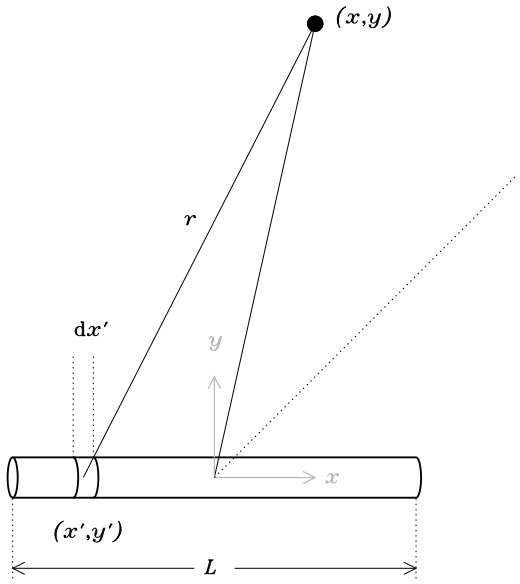

Figure 3.1: Géométrie et notation pour le calcul du potentiel gravitationnel produit par une tige rectilinéaire de section A et longueur L, placée le long de l'axe x. Il s'agit ici de calculer le potentiel gravitationnel à un point P situé à une position (x, y) mesurée dans le même système de coordonnées. La distribution de masse à l'intérieur de l'objet est mesurée via sa densité $\rho(x')$, que nous considérerons constante. De plus, on supposera que la section de la tige a un diamètre beaucoup plus petit que toutes les distances impliquées dans ce qui suit.

(voir Fig. 3.1), et on a de plus supposé que la densité est constante tout le long de la tige. Notez bien que l'intégrale se fait par rapport à la variable x', qui mesure la position de chaque élément de masse contribuant à l'intégrale; la position (x, y) à laquelle le potentiel est calculé est ici fixe, du point de vue du calcul de cette intégrale[1].

Comparez cette intégrale à celle que vous avez solutionnée précédemment dans votre calcul de π (éq. (3.2) ci-dessus). Les bornes d'intégration sont différentes, l'intégrant (la fonction $f(x)$ introduite plus haut) aussi, mais la structure générale est la même: une intégrale définie unidimensionelle, ici pour la variable d'intégration x'.

3.6 Calcul du potentiel gravitationnel

Il s'agit maintenant d'écrire un code C qui évalue numériquement l'intégrale donnée par l'éq. (3.5). En anticipation de ce qui suit, il sera utile de structurer ce code de la manière suivante:

1. Modifiez la ligne d'instruction définissant le maillage d'intégration de manière à ce qu'il couvre maintenant un intervalle $[-L/2, +L/2]$, où L est une constante dont la valeur est initialisée dans le code source.

2. Modifiez la fonction C produite précédemment pour qu'elle calcule maintenant l'intégrant de l'éq. (3.5). ATTENTION: votre fonction a besoin maintenant de TROIS arguments:

[1] Remarquez ici que l'intégrale est devenue unidimensionnelle parce qu'on a négligé le fait que la tige a une étendue finie en y, une approximation justifiable si on s'intéresse au calcul du potentiel gravitationnel à des distances de la tige beaucoup plus grandes que le rayon de sa section.

le point de maille x' où l'intégrant doit être évalué, comme auparavant, ainsi que les coordonnées cartésiennes du point (x, y) où on calcule le potentiel gravitationnel. Encore une fois, ces derniers sont des constantes du point de vue du calcul de l'intégrale, mais leurs valeurs sont requises pour calculer l'intégrant.

3. Rentrez tout ça dans un code C où vous spécifiez les paramètres de la tige, et les coordonnées d'un point (x, y), et qui calcule le potentiel par la méthode du trapèze et l'imprime à l'écran.

Vous avez en maintenant en main un code C d'une certaine complexité. Avant d'aller plus loin, il est impératif de vérifier qu'il calcule bien ce que vous penser calculer. Cette **validation** de votre code est une étape *essentielle* au traitement numérique de tout problème physique. Ici, cette validation peut s'effectuer en prenant note du fait que l'intégrale (3.5) produit une solution analytique si on l'évalue le long de la ligne $x = 0$, car elle se réduit alors à:

$$\Phi(0, y) = -\rho G A \int_{-L/2}^{+L/2} \frac{\mathrm{d}x'}{\sqrt{(x')^2 + y^2}} \ . \tag{3.6}$$

C'est une intégrale du type

$$\int \frac{\mathrm{d}x}{\sqrt{x^2 + a^2}}$$

car, rappelez-vous, la coordonnée y est une constante du point de vue du calcul de l'intégrale. Vous pouvez en trouver la solution analytique dans une table d'intégrale:

$$\int \frac{\mathrm{d}x}{\sqrt{x^2 + a^2}} = \ln\left(x + \sqrt{x^2 + a^2}\right) \ .$$

La procédure de validation consiste à évaluer numériquement le potentiel pour quelques valeurs de y (disons $y = L, 2L$ et $5L$), avec $L = 1$, le long de la ligne $x = 0$, et comparer le résultat avec l'évaluation de la solution analytique ci-dessus. Faites-le pour une discrétisation de l'intégrale en $N = 10, 30$ et 100 points de maille, et vérifiez que tout a bien l'air de converger. Aux fins de cette validation, vous pouvez poser $\rho\, G\, A = 1$.

3.7 Calcul du potentiel gravitationnel dans le plan $[x, y]$

Nous sommes enfin prêt à calculer numériquement le potentiel gravitationnel n'importe où dans le plan $[x, y]$, et à examiner un peu de quoi il a l'air. Pas de panique, nous avons déjà fait pratiquement tout le travail. En fait, nous allons nous limiter ici à examiner ce qui arrive à notre intégrale (3.5) lorsqu'on l'évalue à un point situé très loin de la tige, dans le sens que $x \gg L$ et/ou $y \gg L$. On aura alors soit $x \gg x'$, soit $y \gg x'$, et on pourrait alors approximer l'intégrale pour le potentiel gravitationnel de la manière suivante:

$$\Phi(x, y) = -\rho G A \int_{-L/2}^{+L/2} \frac{\mathrm{d}x'}{\sqrt{(x - x')^2 + y^2}} \simeq -\rho G A \int_{-L/2}^{+L/2} \frac{\mathrm{d}x'}{\sqrt{x^2 + y^2}}$$

$$= -\frac{\rho G A}{\sqrt{x^2 + y^2}} \int_{-L/2}^{+L/2} \mathrm{d}x' = -\frac{\rho G\, A\, L}{\sqrt{x^2 + y^2}} \tag{3.7}$$

où les deux dernières égalités résultent du fait qu'en faisant sauter le x' au dénominateur, plus rien dans l'intégrant ne dépend de x', et donc l'intégrale de $\mathrm{d}x'$ de $-L/2$ à $+L/2$ donne simplement L. Notant maintenant que le dénominateur est simplement la distance r entre l'origine et le point (x, y), et que le produit ρAL est la masse totale (M) de la tige, ceci peut s'écrire comme:

$$\Phi(r) = -\frac{G\, M}{r} \ , \tag{3.8}$$

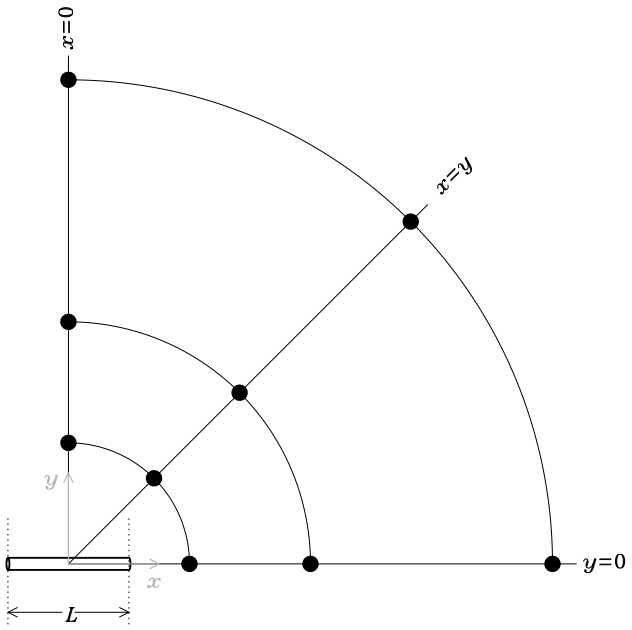

Figure 3.2: Géométrie de la configuration utilisée pour vérifier le comportement asymptotique du potentiel gravitationnel dans le plan $[x, y]$. Les points d'évaluation sont aux intersections de cercles de rayon constant (ici $2L$, $4L$ et $8L$) avec les droites $x = 0$ (axe-y), $y = 0$ (axe-x) et $x = y$ (diagonale). L'origine du système de coordonnées est toujours au centre de la tige, tel qu'indiqué.

ce qui nous ramène au potentiel gravitationnel d'une masse ponctuelle, avec lequel nous avions débuté le développement théorique de ce labo! Ceci indique quelquechose de très intéressant: suffisamment loin de la tige, sa forme n'influence plus son potentiel gravitationnel, et les équipotentielles deviennent des cercles (en 2D; des sphères en 3D), comme pour une masse ponctuelle ayant la même masse que la tige, et plaçée à son centre de gravité, soit ici à $(x, y) = (0, 0)$. Mais "suffisamment", c'est combien ? Zatiz zekweshtcheunne...

La procédure à suivre est illustrée sur la Figure 3.2. Il s'agit de calculer le potentiel $\Phi(x, y)$ pour une série de positions (x, y) situées à différents points sur quelques cercles de rayon constant. Votre code fait déjà ça, il ne s'agit que de le rouler pour répondre aux questions suivantes:

1. Sur chacune de ces trois lignes, à quelle distance radiale devez vous vous éloigner pour que le potentiel calculé numériquement ne diffère de l'éq. (3.8) que par moins d'un pourcent?

2. Cette distance dépend-t-elle de la ligne choisie? Du nombre N de points de maille utilisé pour le calcul de l'intégrale?

3.7.1 QUESTION BONUS

Il existe une explication purement géométrique au calcul de π définit par l'éq. (3.2). Quelle est-elle?

Lectures supplémentaires:

Notes de cours: Sections 2.2 et 2.4
Delannoy: Chapitres 7 et 8

Laboratoire 4

Orbites planétaires

Ce labo vous fait solutionner numériquement un système d'équation différentielles ordinaires décrivant l'orbite d'un corps céleste sous l'influence du champ gravitationnel d'une seconde masse M genre soleil. Les développement théoriques requis se retrouvent au chapitre 3 des notes de cours.

4.1 Objectifs

1. Découvrir une approche numérique aux problèmes de mécanique céleste;
2. Apprendre à solutionner des équations différentielles ordinaires couplées, et étudier le comportement et la précision de divers algorithmes;
3. Développer une appréciation pour l'accumulation des erreurs dans une solution numérique;
4. Consolider et appliquer les notions de programmation en C apprises au cours des trois premiers labos.
5. Poursuivre l'apprentissage de PLPLOT.

4.2 Rapport de Lab

Votre rapport de Lab est à remettre la semaine prochaine **au début** de votre prochain laboratoire. Il doit inclure des réponses aux questions posées aux sections 4.4, 4.5 et 4.6. SVP Inclure des copies de vos codes C, sorties graphiques, etc.

4.3 La mécanique céleste en deux pages

Revenons à Sir Isaac, c'est habituellement un bon début. La dynamique du mouvement d'une masse m sujette à l'attraction gravitationnelle d'une masse M est donnée par:

$$m\mathbf{a} = \mathbf{F} = -\frac{GMm}{r^2}\hat{\mathbf{e}}_r \ , \tag{4.1}$$

où r est la distance séparant les deux masses, et $\hat{\mathbf{e}}_r$ est un vecteur unitaire pointant le long d'un segment de droite reliant m à M, de M vers m. Dans une situation où $M \gg m$ (par exemple si M est le soleil), alors on peut considérer M fixe dans l'espace, et le problème se réduit à solutionner l'équation du mouvement ci-dessus pour la masse m. Il est important de réaliser que l'éq. (4.1) est une expression vectorielle, qui correspond en fait à trois équations, une pour

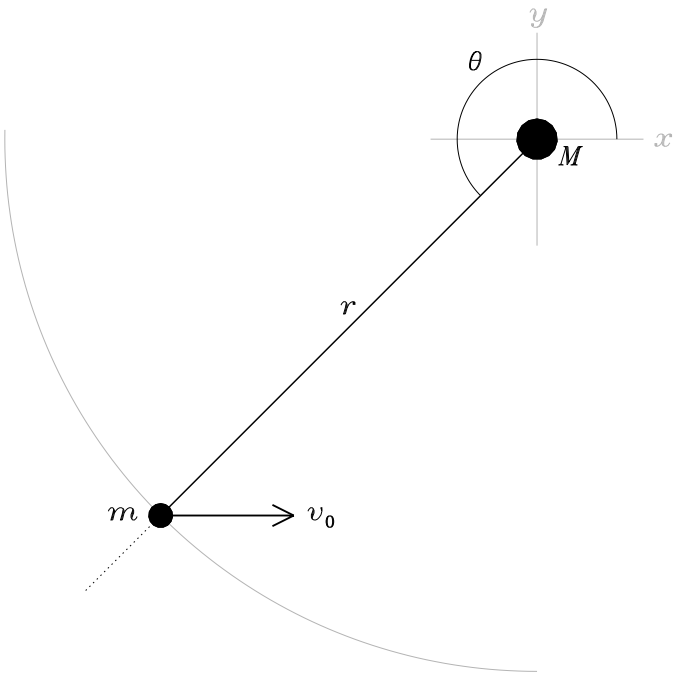

Figure 4.1: Géométrie et notation pour le calcul de la trajectoire d'un objet de masse m orbitant autour d'une masse M ($M \gg m$) sous l'influence de la force gravitationnelle. En coordonnées polaires, la position de m est décrite par les variables r et θ, où cette dernière est mesurée dans le sens antihoraire à partir de l'axe des x.

chaque composante de **a** et **F**. L'orbite recherchée étant ici dans un plan, le problème se réduit à deux composantes si l'on oriente judicieusement le système de coordonnées.

Nous allons travailler en coordonnées cartésiennes avec l'origine à M, la géométrie du problème étant illustrée à la Figure 4.1. La projection de l'éq. (4.1) dans les directions x et y conduit aux deux équations couplées suivantes:

$$a_x = -\frac{GM}{r^2}\cos\theta \ , \tag{4.2}$$

$$a_y = -\frac{GM}{r^2}\sin\theta \ , \tag{4.3}$$

où les sinus et cosinus proviennent de la décomposition de la force gravitationnelle en x et y. Il s'agit maintenant d'exprimer sous la forme d'équations différentielles en x et y uniquement. Pour l'orientation des axes cartésiens sur la Figure 4.1, le passage d'un système de coordonnées à l'autre se fait via les relations habituelles:

$$x = r\cos\theta \ , \qquad y = r\sin\theta \ , \qquad r = \sqrt{x^2+y^2} \ , \qquad \theta = \arctan y/x \ . \tag{4.4}$$

Et comme l'accélération est la dérivée seconde de la position, on obtient

$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2} = -\frac{GM}{(x^2+y^2)^{3/2}}\,x \ , \tag{4.5}$$

$$\frac{\mathrm{d}^2y}{\mathrm{d}t^2} = -\frac{GM}{(x^2+y^2)^{3/2}}\,y \ , \tag{4.6}$$

Il s'agit donc ici de deux EDOs d'ordre deux, couplées nonlinéairement via le dénominateur au membre de droite. La prochaine étape est de convertir tout ça en *quatre* EDOs d'ordre 1 par l'introduction de deux nouvelles variables dépendantes judicieusement définies:

$$\frac{\mathrm{d}x}{\mathrm{d}t} = u \ , \tag{4.7}$$

$$\frac{\mathrm{d}y}{\mathrm{d}t} = v \ , \tag{4.8}$$

ce qui transforme les éqs. (4.5) —(4.6) en

$$\frac{\mathrm{d}u}{\mathrm{d}t} = -\frac{GM}{(x^2+y^2)^{3/2}}x \ , \tag{4.9}$$

$$\frac{\mathrm{d}v}{\mathrm{d}t} = -\frac{GM}{(x^2+y^2)^{3/2}}y \ , \tag{4.10}$$

Le problème du calcul de l'orbite reviens donc à solutionner quatre EDOs d'ordre 1 et couplées, soit les éqs. (4.7)—(4.10) pour quatre variables dépendantes $x(t), y(t), u(t), v(t)$ qui sont toutes des fonctions d'une seule variable indépendante, le temps t. Pour ce faire nous utiliserons une maille temporelle équidistante:

$$t \to t_k \ , \qquad t_{k+1} = t_k + \Delta t \ , \qquad k = 0, 1, 2, ... \tag{4.11}$$

sujet à une condition initiale donnée qui sera spécifiée sous peu.

4.4 Calculer une orbite circulaire (et qui le reste...)

Commençons par le cas le plus simple, soit celui d'une orbite circulaire; dans un tel cas, vous savez déjà que la dynamique se réduit à un équilibre entre la force gravitationnelle et la force centrifuge:

$$\frac{GM}{R^2} = R\omega^2 \ , \tag{4.12}$$

où R est le rayon de l'orbite et ω (mesuré en rad s^{-1}) la vitesse angulaire de rotation, égale à $2\pi R/P$ avec P la période orbitale.

Votre première tâche consistera donc à intégrer numériquement les éqs. (4.7)—(4.10) pour une orbite circulaire, et vérifier sous quelles conditions votre orbite se retrouve vraiment circulaire, et le demeure si vous poussez l'intégration temporelle à un grand nombre d'orbites successives. Comme critère de précision par rapport à la solution exacte, vous utiliserez le fait que pour une orbite circulaire le rayon doit demeurer constant à sa valeur initiale:

$$R(t) = \sqrt{x^2(t) + y^2(t)} = \text{constante} \ .$$

Nous ferons le calcul pour l'orbite terrestre, avec la condition initiale suivante:

$$x(0) = 0 \ , \qquad y(0) = -R_0 \ , \qquad u(0) = \frac{2\pi R_0}{P} \ , \qquad v(0) = 0 \ .$$

où R_0 est égal à l'unité astronomique (distance moyenne Terre-Soleil) AU$= 1.49597892\times10^{11}$ m, P vaut un an (exprimé en secondes!). Vous aurez également besoin des valeurs de la masse du soleil $M = 1.98 \times 10^{30}$ kg et de la constante gravitationnelle $G = 6.6732 \times 10^{-11}$ N m^2 kg^{-2}. Les étapes de la modélisation sont les suivantes:

1. Écrivez un code C qui solutionne les éqs. (4.7)—(4.10) à l'aide de la méthode d'Euler explicite (voir la §3.2.1 des notes, et/ou la Fig. 3.4 toujours dans les notes de cours, si vous manquez d'inspiration pour l'écriture du code). Commencez par tout coder en simple précision (type `float` pour toutes les variables réelles).

2. Calculez une orbite circulaire pour la Terre (même si l'orbite terrestre est légèrement elliptique), utilisant la condition initiale posée ci-dessus. Commencez avec un pas de temps d'une journée terrestre (exprimé en secondes!), et tracez la progression de votre orbite avec PLPLOT (voir plus bas pour plus de détails sur vos options graphiques).

3. Examinez la variation de $R(t)$ à mesure que que l'intégration est poussée loin dans le temps.

4. Diminuez le pas de temps et examinez comment les erreurs diminuent.

5. Changez les déclarations des variables réelles de `float` à `double`, et répétez l'étape précédente. L'erreur provient-elle principalement d'un problème de troncation?

Coté graphismes, vous avez encore deux options pour porter vos orbites en graphiques, soit les fonctions PLPLOT `pljoin` et `plline`. Ici il serait naturel d'insérer un appel à `pljoin` à l'intérieur de la boucle temporelle de l'algorithme d'Euler, dans le genre:

```
pljoin(x[k],y[k],x[k+1],y[k+1]) ;
```

Votre seconde option consiste à faire un seul appel à `plline` après sortie de la boucle temporelle, une fois toutes vos valeurs de x et y calculées:

```
plline(NPAS,x,y) ;
```

Petit rappel, dans un tel cas les tableaux `x` et `y` doivent avoir été déclarés `double`!

Comme vous cherchez à tracer une orbite circulaire, il serait bien qu'elle le soit sur votre fenêtre graphique, autrement dit que PLPLOT choisisse un rapport d'aspect qui préserve le rapport d'aspect spéficifié par les valeurs de `xmin`, `xmax`, `ymin` et `ymax` dans l'appel à `plenv`. C'est là le rôle du cinquième paramètre fourni en argument à `plenv`; de manière générale:

```
plenv(xmin,xmax,ymin,ymax,just,style) ;
```

La variable `just` ne peut prendre que deux valeurs légales, soit 0 et 1; la valeur zéro laisse à PLPLOT le soit de choisir un rapport d'aspect comparable à la fenêtre graphique, tandis que `just`= 1 force un rapport d'aspect 1:1, tel que désiré ici. La valeur de la variable `style` contrôle le style des axes tracés. Expérimentez avec les valeurs -2, -1, 1, et 2, histoire de voir l'effet...

Si vous vous voulez tracer plusieurs orbites sur le même graphique, il s'agit simplement de calculer ces orbites additionnelles, et faire un appel subséquent à `plline` pour chacune. Vous n'avez pas à exécuter de nouveau `plenv` et `pllab` si toutes vos orbites sont tracées sur le même graphique! Dans un tel cas il peut aussi être utile de superposer aux courbes tracées par `plline` des symboles permettant d'identifier chaque courbe (et de visualiser la discrétisation!). Ceci peut s'effectuer via la fonction `plpoin`:

```
plpoin(NPAS,x,y,symcode) ;
```

Le dernier argument est un code numérique (type `int`) spécifiant le type de symbole tracé. PLPLOT mets à votre disposition 30 symboles prédéfinis; en voici quelques-uns, avec leurs codes numériques: $1 \equiv \cdot$, $2 \equiv +$, $3 \equiv *$, $4 \equiv \circ$, $5 \equiv \times$, $7 \equiv \triangle$; je vous laisse la joie de découvrir les autres...

Une autre possibilité est de changer la couleur du trait traçé par `plline` (ou `pljoin`, ou encore du symbole traçé par `plpoin`). Le changement de couleur se fait par un appel à la fonction `plcol0`:

```
plcol0(colcode) ;
```

où l'argument `colcode` (type `int`) est un code numérique contrôlant la couleur choisie, selon la convention présentée au Tableau 3.1 ci-dessous. La nouvelle couleur demeure active jusqu'à ce que le graphique soit terminé (via appel à `plend()`), ou qu'un nouvel appel à `plcol0` la change de nouveau.

Table 4.1: Code de couleur pour la fonction `plcol0`

`colcode`	Couleur	Commentaire
0	Noir	Valeur défaut pour arrière plan
1	Rouge	Valeur défaut pour traits et symboles
2	Jaune	
3	Vert	
4	Bleu marin	
5	Rose	
6	Orange	
7	Gris	
8	Brun	
9	Bleu	
10	Violet	
11	Cyan	
12	Turquoise	
13	Magenta	
14	Saumon	
15	Blanc	

Comme vous aurez un rapport de lab à remettre cette semaine, il vous faudra sauvegarder vos graphiques sous une forme imprimable, ou insérable dans un programme d'édition de texte, si vous en utilisez un pour vos rapports. Le choix du format graphique de sortie se fait au moment de l'exécution, en fournissant un code numérique approprié parmi la sélection proposée interactivement par `plinit`. À date vous n'avez utilisé que la sortie graphique à l'écran, mais plusieurs autres options sont disponibles. Celles qui vous seront probablement les plus utiles sont postscript noir-et-blanc (option 4) ou couleur (option 13), svg (option 14), pdf (option 16) et png (option 19). Vous aurez déjà remarqué que les couleurs de défaut, pour une sortie à l'écran (option 1), définissent un trait rouge sur fond noir; ceci demeurera le cas pour les sorties en png ou pdf; si cependant la sortie choisie est de type postscript, le tracé sera noir sur fond blanc. Pour forcer un fond blanc, vous devez appeler la fonction suivante *avant* votre appel à `plinit`:

```
plscolbg(255,255,255) ;
```

Remarquez que le nom de cette fonction commence par `pls...` plutôt que `pl...`; le "`s`" indique une fonction affectant les variables internes au système d'exploitation de l'ordi, plutôt qu'une fonction effectuant une action graphique.

4.5 Orbites elliptiques

Comme vous le verrez en PHY-1651, une orbite circulaire est caractérisée par le moment cinétique maximal pour une énergie totale (cinétique plus potentielle gravitationnelle) donnée; il serait donc fort surprenant que la formation et l'évolution du système solaire ait pu produire des orbites parfaitement circulaire, et en effet toutes les orbites planétaires sont en fait elliptiques, certaines à peine (comme la Terre), d'autres beaucoup plus (comme Mars ou Pluton).

Cette partie du laboratoire vise à vous faire calculer une orbite elliptique. Utilisant le même code C que pour l'orbite circulaire, toujours avec un pas de temps d'une journée, débutez maintenant une intégration temporelle en utilisant cette fois comme condition initiale:

$$x(0) = 0 \ , \qquad y(0) = -1\,\mathrm{AU} \ , \qquad u(0) = 0.8 \times \frac{2\pi R_0}{P} \ , \qquad v(0) = 0 \ .$$

Avec le mouvement initial orienté comme pour une orbite circulaire, mais la vitesse réduite de 20%, il n'y a plus équilibre entre la force centrifuge et l'attraction gravitationnelle. La masse m tendra donc à se déplacer vers une orbite ayant un r plus petit. Tout ca ne change cependant pas grand chose à votre à votre code C de la section précédente! Nous devons cependant nous inventer autre chose que la constance du rayon orbital pour mesurer l'erreur de nos solutions numériques. Allons-y pour l'énergie totale E de la planète en orbite:

$$E(t) = \frac{1}{2} m(u^2(t) + v^2(t)) - \frac{G\,M\,m}{\sqrt{x^2(t) + y^2(t)}} = \text{constante} \ .$$

Les étapes sont les suivantes:

1. Roulez votre code avec la condition initiale ci-dessus, sur un intervalle suffisamment long pour couvrir une demi-douzaine d'orbites. Tracez votre orbite avec PLPLOT. L'orbite est-elle vraiment elliptique? Qu'observez-vous?
2. Calculer l'énergie totale (cinétique plus gravitationnelle) de votre corps en orbite; l'énergie demeure-t-elle constante, comme il se doit en l'absence de processus dissipatifs?
3. Faites varier le pas de temps dans le but de produite une orbite qui soit le plus stable possible.

La dernière étape de ce labo est de vous faire passer à la méthode de Heun, histoire de voir jusqu'à quel point les comportements observés à date sont influencés par l'erreur de discrétisation associée à la méthode d'Euler. Vous vous rappelerez (sinon voir les §3.2.2 et 3.3.4 des notes) que ceci consistait à estimer les membres de droites des EDOs comme une moyenne à mi-pas de temps, la valeur au pas de temps $k+1$ étant calculée par extrapolation linéaire à partir du pas k. Pour la forme générale,

$$\frac{\mathrm{d}f}{\mathrm{d}t} = g(t, f) \ , \tag{4.13}$$

on a alors:

$$f_{k+1} = f_k + \frac{\Delta t}{2} \left[g(t_k, f_k) + g(t_{k+1}, f_k + h\, g(t_k, f_k)) \right] + O(h^2) \ . \tag{4.14}$$

Pour nos quatre variables x, y, u et v, cette extrapolation aurait la gueule suivante:

$$x^*_{k+1} = x_k + (\Delta t) \times \left.\frac{\mathrm{d}x}{\mathrm{d}t}\right|_{t_k} = x_k + (\Delta t) \times u_k \ , \tag{4.15}$$

$$y^*_{k+1} = y_k + (\Delta t) \times \left.\frac{\mathrm{d}y}{\mathrm{d}t}\right|_{t_k} = y_k + (\Delta t) \times v_k \ , \tag{4.16}$$

$$u^*_{k+1} = u_k + (\Delta t) \times \left.\frac{\mathrm{d}u}{\mathrm{d}t}\right|_{t_k} = u_k - (\Delta t) \times \frac{GM}{(x_k^2 + y_k^2)^{3/2}}\, x_k \ , \tag{4.17}$$

$$v^*_{k+1} = v_k + (\Delta t) \times \left.\frac{\mathrm{d}v}{\mathrm{d}t}\right|_{t_k} = v_k - (\Delta t) \times \frac{GM}{(x_k^2 + y_k^2)^{3/2}}\, y_k \ . \tag{4.18}$$

L'EDO pour u discrétisée selon la méthode de Heun conduirait alors à l'algorithme:

$$u_{k+1} = u_k - \frac{G\,M\,\Delta t}{2}\left(\frac{x_k}{(x_k^2+y_k^2)^{3/2}} + \frac{x_k+(\Delta t)u_k}{((x_k+(\Delta t)u_k)^2+(y_k+(\Delta t)v_k)^2)^{3/2}}\right) . \qquad (4.19)$$

Avant d'aller plus loin, ASSUREZ-VOUS DE BIEN COMPRENDRE comment les extrapolations (4.15)–(4.18) ci-dessus, de concert avec l'algorithme général donné par l'éq. (4.14), conduisent bien à l'expression (4.19) ci-dessus. Ensuite,

1. Calculez les algorithmes correspondants pour les trois autres variables x, y, et v.
2. Recalculez votre orbite circulaire avec le même pas de temps que votre solution avec pas de temps d'une journée; comment se comparent les variations du rayon?
3. Recalculez votre orbite elliptique avec la même condition initiale que précédemment, toujours pour un pas de temps d'une journée; comment se compare la stabilité de l'orbite? Le niveau de conservation de l'énergie totale?
4. **QUESTION BONUS:** Circulaire ou elliptiques, Euler ou Heun, vos orbites en viennent toutes à décrire un mouvement en spirale de rayon croissant inexorablement. Trouvez une explication pour cette curieuse propriété associée au comportement de l'erreur de modélisation.

4.6 Orbites liées versus non-liées

Recalculez quelques solutions utilisant la méthode de Heun et un pas de temps d'une journée, avec des conditions initiales sur u variant de $0.4 \times 2\pi R_0/P$ à $2.0 \times 2\pi R_0/P$ en sauts de 0.2. Superposez ces orbite sur le même graphique à l'aide de PLPLOT, et calculez l'énergie totale de chacune. Quelle relation pouvez-vous établir entre la forme de l'orbite et son énergie totale?

Lectures supplémentaires:

Notes de cours: Chapitre 3

Laboratoire 5

La carte logistique

5.1 Objectifs

1. Explorer le comportement chaotique de formules de récurrence nonlinéaires;
2. Apprendre à produire des diagrammes de bifurcation;
3. Apprendre à calculer des exposants de Lyapunov;
4. Apprivoiser les notions d'attracteur, d'invariance d'échelle, d'universalité et de bassin d'attraction;
5. Consolider vos habiletés graphiques avec PLPLOT;

5.2 Rapport de Lab

Le rapport de lab doit inclure des réponses aux questions posées aux sections 5.4 à 5.7 inclusivement, graphiques à l'appui.

5.3 Les cartes itératives

Plusieurs des exemples les plus classiques du chaos nous viennent de la biologie, et plus spécifiquement de la dynamique des populations en écologie, et ce sera le contexte qui servira à développer ce labo. À prime abord ca n'a pas grand chose à voir avec la physique, mais (1) la culture scientifique générale, c'est important, et (2) bien des systèmes très physiques se comportent de la même manière, incluant (entre autres) la convection thermique dans les fluides, les circuits électriques LR incluant une diode, et peut-être même le cycle d'activité du soleil, mais la mathématique sous-jacente au chaos dans ces systèmes est passablement plus lourde. Donc allons-y...

On considère une population d'organismes dont les générations successives ne coexistent pas temporellement, par exemple, des insectes saisonniers qui meurent à l'automne après avoir pondu leurs oeufs, qui n'écloront que le printemps suivant. En l'absence de catastrophe écologique hivernale (e.g., l'étang est vidé et remblayé pour faire place à une infrastructure essentielle au bien-être de l'humanité, comme un terrain de golf ou un stationnement de centre d'achat), on s'attend à ce que le nombre d'individus N_{k+1} dans la génération à venir $(k+1)$ soit proportionnel à celui de la génération précédente (k), dans le genre:

$$N_{k+1} = AN_k \ , \qquad k = 0, 1, 2, ... \ , \tag{5.1}$$

où la constante A (> 0) incorpore tous les facteurs reliés au succès reproductif, à la survie hivernale des oeufs, etc. On voit que si $A < 1$, la population décroitra d'une année à l'autre,

et à l'inverse croitra inexorablement si $A > 1$. Une telle croissance ne pourra perdurer, pour un bon nombre de raisons, dont la quantité limitée de ressources (nourriture) disponible à la population, et au fait qu'une population plus abondante augmente les chances de survie des prédateurs. En vertu de tout ça, introduisons un terme de "correction" au membre de droite de notre expression ci-dessus:

$$N_{k+1} = AN_k - BN_k^2 \ , \qquad k = 0, 1, 2, ... \ , \tag{5.2}$$

où la nouvelle constante $B > 0$. Ce nouveau terme est donc négatif, comme il se doit, puisqu'il doit tendre à faire diminuer N_{k+1}. Il est clair que le membre de droite tombera à zéro si N_k atteint une valeur $N^{\max} = A/B$. C'est la population maximale pouvant être supportée par notre "écosystème" à deux paramètres; et si $N_k = 0$ pour une génération k quelconque, alors $N_{k+1} = 0$ aussi, et c'est l'extinction (pas de génération spontanée ou d'immigration d'un étang voisin dans le modèle).

Introduisons une nouvelle variable $x_k = N_k/N^{\max}$; l'équation ci-dessus devient alors

$$x_{k+1} = Ax_k(1 - x_k) \ , \qquad k = 0, 1, 2, ... \ , \tag{5.3}$$

qui ne dépend plus maintenant que d'un seul paramètre, soit A. Donc, à partir d'une valeur x_0 décrivant la taille d'une population initiale, on peut calculer séquentiellement les populations aux générations/années subséquentes:

$$x_1 = Ax_0(1 - x_0) \ , \qquad x_2 = Ax_1(1 - x_1) \ , \qquad x_3 = Ax_2(1 - x_2) \ , \qquad ... \qquad \text{etc} \tag{5.4}$$

Comme la valeur x_k est réinjectée algébriquement à l'itération suivante, on appelle une formule de récurrence du genre de l'éq. (5.3) une **carte itérative**; et, pour des raisons historiques, celle définie spécifiquement par l'éq. (5.3) est appelée la **carte logistique**. Il faut noter que le processus est complètement déterministe; tant que les conditions écologiques ne changent pas (i.e., les facteurs numériques A et B demeurent constants), la population initiale x_0 détermine la population à toutes les générations subséquentes, $k \to \infty$. Donc, calculer l'évolution de la population d'une génération à l'autre est, en toutes apparences, la simplicité même. Mais attention, il y a une montagne de surprises cachées là-dedans...

5.4 Transient et stabilité

Votre première tâche sera de coder en C l'équation (5.3) ci-dessus. Il ne s'agit que de la transcrire en C à l'intérieur d'une boucle inconditionnelle partant à $k = 0$, avec x_0 initialisé avant le début de la boucle, et en effectuant la substitution $x_{k+1} \to x_k$ avant de passer à l'itération suivante. Pas besoin que je vous donne un canevas de code pour ça, n'est-ce-pas...? Ensuite:

1. Posez $A = 0.8$, et calculez les 100 premières itérations pour des conditions initiales $x_0 = 0.1$, $x_0 = 0.3$, $x_0 = 0.7$ et $x_0 = 0.9$. Portez le résultat en graphique en fonction de l'itération (quatre courbes sur le même graphique, bien identifiées).

2. Répétez l'étape 1 pour $A = 2$, $A = 3.2$, $A = 3.5$ et $A = 3.9$ (avec les mêmes quatre conditions initiales). Il ne s'agira ici que d'inclure un appel à la fonction PLPLOT `pljoin`, traçant un segment entre chaque paire de points (x_k, x_{k+1}).

3. Vos courbes convergent-elles toujours vers une valeur stable? Combien d'itérations prend cette convergence ? Est-ce-que ca dépend de la valeur de A choisie? De la condition initiale?

5.5 Bifurcation

Vous aurez réalisé à la section précédente que pour des valeurs de A suffisamment basses, la carte itérative en vient toujours à stabiliser sur un pattern de variation qui ne dépend pas de la condition initiale, mais peut dépendre de la valeur de A. Explorons maintenant ce comportement, en traçant un **diagramme de bifurcation**. L'algorithme est le suivant:

1. On choisit une valeur initiale de A ($A = 0.8$, disons);
2. On choisit une condition initiale quelconque ($x_0 = 0.1$, disons), et on itère la carte pendant quelques centaines d'itérations afin de bien perdre la mémoire de la condition initiale;
3. Portons en graphique les 10 dernières valeurs de x_k ainsi produites, sous la forme d'un point dans le plan $[A, x]$;
4. Augmentons A d'un incrément δA ($= 0.005$, disons), et répétons les étape 2 à 4;
5. Augmentons de nouveau A par δA, répétons les étape 2 à 4;
6. Et ainsi de suite jusqu'à ce que $A = 3.56$.

Un canevas de code C effectuant cette procédure pourrait être structuré comme suit:

```
#include <stdlib.h>
#include <plplot.h>
int main(void)
{
  float a, da, xk,xkp1 ;
  double xp[1], yp[1] ;
  int   niter=500 ;               /* nombre d'iterations de la carte */
  ...                             /* autres declarations */
  ...                             /* initialisation de PLPLOT, axes, etc. */
  a=0.8 ;                         /* initialisation de A */
  da=0.005 ;                      /* increment pour A */
  while ( a <= 3.56 ) {           /* boucle sur les valeurs de A */
      xk=0.1 ;                    /* condition initiale (arbitraire) */
      for (k=0 ; k<niter ; k++ ) { /* boucle iterative */
           xkp1=a*xk*(1.-xk) ;    /* nouvelle valeur de x */
           xk=xkp1 ;              /* la nouvelle population devient l'ancienne */
           if (k >= niter-10) {   /* Porter en graphique les dernieres 10 iterations */
              xp[0]=a ; yp[0]=xk ;
              plpoin(1,xp,yp,1) ; /* Trace le point (A,xk) */
           }
      }
      a+=da ;                     /* on incremente A de delta A */
  }
  ...                             /* fermer PLPLOT, etc. */
}
```

Si tout s'est bien passé, vous devriez obtenir un graphique ressemblant à la Figure 5.1 ci-dessous. Remarquez d'abord que si $A < 1$, x_k se retrouve à zéro, comme on s'y attendrait en vertu de notre discussion initiale à la §5.3. Notez ensuite la séquence de bifurcations à mesure que A augmente; chacune de ces bifurcations cause ce qu'on appelle un **dédoublement de la période**. Finalement, notons le bizarre usage de la fonction `plpoin` de PLPLOT; comme cette fonction, par son design, trace des points pour une série de paires de coordonnées (x, y) contenues dans deux tableaux en argument, même si ici on ne désire ne porter en graphique

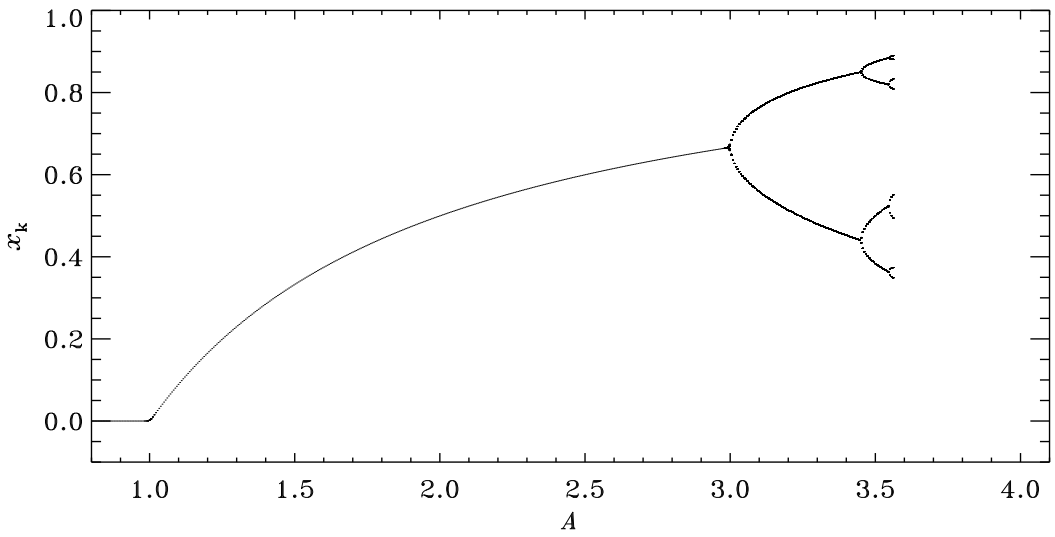

Figure 5.1: Diagramme de bifurcation pour la carte logistique. Notez comment, après la bifurcation initiale à $A = 1$, les branches se dédoublent à mesure que A augmente.

qu'un seul point, les coordonnées du point à tracer doivent être fournies en argument à la fonction `plpoin` sous la forme de deux tableaux de longueur 1, de type `double` de surcroit. D'où la bizarre manoeuvre à l'intérieur du `if` consistant à assigner la valeur de a (abcisse) à l'élément 0 du tableau `x`, et le x_k à `y[0]`. Un appel plus "logique", de la forme `plpoin(1,a,xk,1) ;` produira un avertissement cryptique à la compilation, et un résultat erroné sur le graphique! C'est bête mais c'est comme ça...

Comprenez bien le lien entre ce diagramme et les graphiques produits à la §5.4, en comprenez pourquoi il est important de porter en graphique plusieurs valeurs de x_k pour chaque valeur de A utilisée dans la construction du diagramme.

5.6 Régime chaotique

On voit bien sur la Figure 5.1 que la séquence de dédoublement de la période $1 \to 2 \to 4 \to 8...$ se produit de plus en plus rapidement à mesure que A augmente. Où celà se terminera-t-il? Reprenez le calcul du diagramme de bifurcation de la section précédente, mais cette fois poussez la valeur de A jusqu'à $A = 4.0$. De plus, portez en graphique cette fois une cinquantaine de valeurs de x_k par valeur de A, plutôt que 10. Vous devriez obtenir quelquechose ressemblant à la Figure 5.2 ci-dessous. Au delà de $A \simeq 3.56$, on assiste à une transition vers un régime où les valeurs successives x_k semblent se répartir dans un ou plusieurs intervalles d'étendues finies en amplitudes x_k, plutôt que sur des branches bien définies. Bien que bornée entre des valeurs inférieure et supérieure qui changent avec A, la séquence des x_k ne présente plus de pattern régulier (retournez voir vos graphiques de la §5.4 pour $A = 3.9$). Nous sommes maintenant dans le **régime chaotique** de la carte logistique, qui marque le point final de la séquence de dédoublement de période.

Il s'agit maintenant de vérifier si le régime $3.56 \lesssim A \leq 4$ est *vraiment* chaotique, dans le sens mathématique du terme. On verra en classe (et au chapitre 4) que la signature du chaos est la divergence exponentielle de deux solutions ne différant que très peu au niveau de la condition initiale. Facile à simuler avec la carte logistique; posez $A = 3.99$, et effectuez deux simulations de 10^3 itérations, à partir des conditions initiales suivantes:

$$x_0^{(1)} = 0.1 \ , \qquad x_0^{(2)} = 0.1 + 10^{-6} \ . \tag{5.5}$$

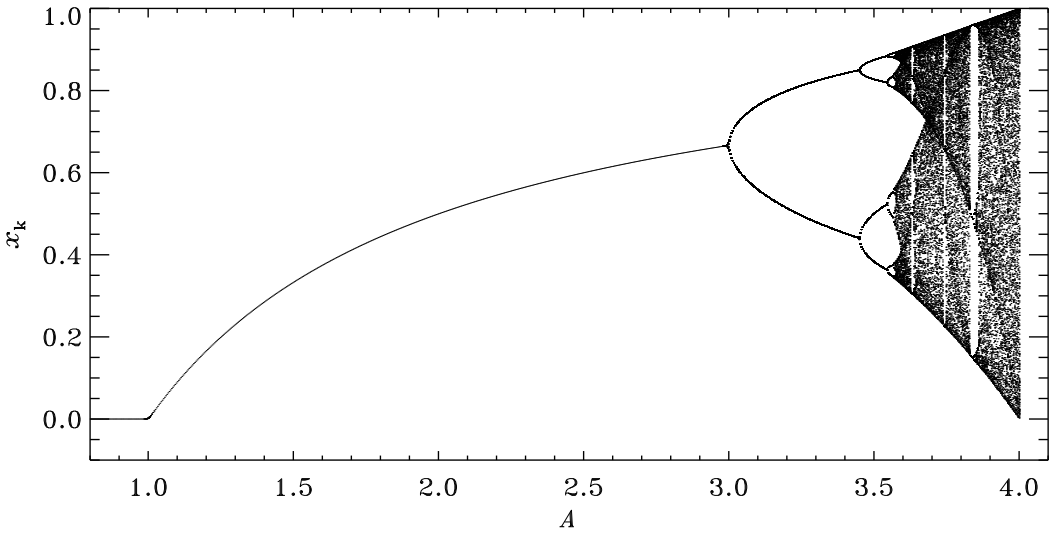

Figure 5.2: Diagramme de bifurcation complet pour la carte logistique, incluant la région chaotique à $3.56 \lesssim A \leq 4$.

à chaque itération, calculez l'écart normalisé:

$$\delta x_k = 2\left|\frac{x_k^{(1)} - x_k^{(2)}}{x_k^{(1)} + x_k^{(2)}}\right| , \qquad k = 0, 1, 2, ... \tag{5.6}$$

Si le système est véritablement chaotique, on devrait avoir:

$$\delta x_k = \delta x_0 \exp(\Lambda k) , \tag{5.7}$$

où Λ est l'**exposant de Lyapunov**, et l'indice k est interprété comme une mesure temporelle. L'idée est donc de portez en graphique $\log(\delta x_k)$ versus k, identifier l'intervalle de temps où la variation prend la forme d'une croissance linéaire sur ce graphe, mesurer la pente et en déduire ainsi la valeur de l'exposant de Lyapunov.

5.7 Attracteur, invariance d'échelle, universalité, et bassin d'attraction

Repensez à vos résultats de la §5.4. Quelle que soit la valeur de la condition initiale x_0, à une valeur de A donnée la carte logistique converge toujours sur la même valeur (ou groupes de valeurs) numériques de x_k, et ces valeurs sont celles représentées sur la Figure 5.2. Cet objet géométrique est donc un **attracteur** de la dynamique de la carte logistique. Il a cependant une forme géométrique beaucoup plus complexe que les points ou tores avec lesquels nous avions fait connaissance dans nos études du pendule nonlinéaire (chapitre 4 des notes de cours); en fait, l'objet géométrique de la Figure 5.2 est un **attracteur étrange**, étrange à cause de certaines particularités géométriques et dynamiques que nous explorerons un peu dans cette dernière section du Labo.

5.7.1 Invariance d'échelle

Bien que ce ne soit pas tout à fait évident à prime abord, le diagramme de bifurcation complet (i.e., incluant le régime chaotique) que vous avez produit ła §5.6 est caractérisé par une propriété

géométrique particulière, appellée **invariance d'échelle** ou **autosimilarité**. Explorons un peu ceci, de la manière suivante:

1. Modifiez votre code de la section 5.5 pour ne calculer que l'intervalle $3.5 \leq A \leq 3.6$, maintenant avec $\delta A = 10^{-3}$. Faites un diagramme de bifurcation comme précédemment, mais en limitant l'axe vertical en PLPLOT à l'intervalle $0.4 \leq x_k \leq 0.6$; vous pouvez accomplir ceci en via les valeurs `xmin`, `xmax`, etc, fournies en argument à la fonction `plenv` (voir second labo). Assurez vous de réduire votre δA de manière à avoir suffisamment de résolution pour avoir un beau diagramme. Il sera probablement utile maintenant de porter en graphique une centaine de valeurs de x_k à chaque valeur de A.

2. Répétez l'expérience maintenant avec A dans l'intervalle $3.56 \leq A \leq 3.59$, avec $\delta A = 2 \times 10^{-4}$. Sur votre graphique PLPLOT, ajustez l'échelle verticale pour couvrir l'intervalle $0.53 \leq x_k \leq 0.58$. Ajustez le δA et le nombre de valeurs x_k écrites sur disque afin d'avoir une densité de points similaire à ce que vous aviez précédemment.

3. Discutez les similarités et différences entre vos trois diagrammes de bifurcation. Jusqu'à quel point les diagrammes sont-ils self-similaires?

5.7.2 Universalité

Considération une variation "cubique" de notre carte logistique (5.3):

$$x_{k+1} = A x_k^2 (1 - x_k) \ , \qquad k = 0, 1, 2, ... \ , \tag{5.8}$$

Tracez un diagramme de bifurcation pour cette nouvelle carte itérative, et discutez-en les similarités et différences avec celui produit à la §5.6 pour la carte logistique. Vous n'avez qu'une ligne de code à modifier, et cette modification est presque triviale. Notez cependant que l'attracteur de cette carte n'existe pas nécessairement exactement dans le même intervalle de A que celui de la carte logistique, et donc que vous aurez à tâtonner avec les valeurs de A pour "retrouver" l'attracteur.

Le fait que des cartes itératives mathématiquement distinctes produisent des diagrammes de bifurcation topologiquement semblables est une manifestation de l'**universalité** de la transition au chaos dans cette classe de cartes.

5.7.3 Bassin d'attraction

La carte cubique définie par l'éq. (5.8) possède une autre propriété intéressante que vous explorerez dans cette dernière partie du labo. Pour une série de valeurs de A équidistantes ($\delta A = 0.1$, disons), calculez une série de séquences itératives pour des conditions initiales équidistantes en x_0, e.g., de $x_0 = 0$ à $x_0 = 1$ en saut de 0.1. Tracez, dans le plan $[A, x_0]$, un symbole à chaque point $[A, x_0]$ ainsi testé, et utilisez deux types de symboles distincts, un pour les itérations qui convergent vers zéro, et un symbole différent pour celles qui convergent vers des valeurs non-nulles. Vous devriez ainsi produire une carte du **bassin d'attraction** de votre attracteur étrange; ce bassin est défini comme l'ensemble des conditions initiales qui convergent vers l'attracteur.

Notons finalement que certains des concepts explorés ici, notamment l'invariance d'échelle et l'universalité, ne sont pas exclusives aux systèmes chaotiques; nous les croiserons de nouveau dans les mois qui viennent, dans des contextes fort différents. Pour l'instant, le labo 5 est terminé!

Lectures supplémentaires:

La carte logistique est discutée dans tous les bouquins traitant de chaos. Voir la bibliographie à la fin du chapitre 4 des notes de cours pour quelques bonnes suggestions. Une relecture des dernières sections du chapitre 4 des notes de cours sera également de rigueur, histoire de se rappeller comment est défini l'exposant de Lyapunov.

Laboratoire 6

Intégration Monte Carlo

Avant de vous lancer dans ce Labo, relisez bien les sections 2.5.4, 5.1, 5.2 et 5.4 des notes de cours. Rien de nouveau coté PLPLOT cette semaine.

6.1 Objectifs:

1. Vous familiariser avec la production de nombres aléatoire en C;
2. Apprendre à calculer des intégrales multidimensionnelles par Monte Carlo;
3. Apprendre à estimer l'erreur sur les résultats numériques de calculs de type Monte Carlo.

6.2 Rapport de Laboratoire

La rapport doit contenir des réponses, codes et résultats numériques à l'appui, aux questions posées aux §6.4 et 6.5, et doit être remis le lundi suivant le retour de la semaine de relâche, avant 12:00 (midi).

6.3 Utiliser `rand()` pour générer des nombres aléatoires

Le langage C inclut un générateur de nombres aléatoires prenant la forme d'une fonction appelée `rand()`. Nous ne nous préoccupperons pas trop de ce qui se passe à l'intérieur de cette boite noire. Il suffit de savoir que cette fonction produit un entier (i.e, type `int`) prenant une valeur entre zéro et 2147483647 (sur un système monté en 64 bits comme ESIBAC). Un appel à `rand()` est l'équivalent de rouler un dé à autant de faces.

On peut produire une nombre aléatoire (r disons) de type `float` uniformément distribué dans l'intervalle unitaire $[0, 1]$ à l'aide d'une instruction du genre:

```
r = 1.*rand()/RAND_MAX ;
```

où `RAND_MAX`= 2147483647 est une constante de type `int` préféfinie inclue dans la librairie `<stlib.h>`, et la variable `r` doit préalablement avoir été déclarée `float`, bien entendu. La pré-multiplication par `1.` est essentielle ici pour convertir l'entier retourné par `rand()` en type `float` **avant** la division par `RAND_MAX`, sinon c'est garanti que r vaudra pratiquement tout le temps zéro puisque `rand()` $\leq$ `RAND_MAX`= 2147483647.

La fonction `rand()` utilise un germe prédéfini valant 1; il est possible de changer ce germe à l'aide d'une autre fonction C prédéfinie appelée `srand`, contenue dans `<stdlib.h>`. Cette fonction-action accepte un germe (variable de type `int` fournie par l'usager), et ne renvoie aucune valeur numérique au code l'ayant appellé. On doit donc l'invoquer à l'aide d'une instruction du genre:

```
   int mongerme=1234 ;   /* Declaration et initialisation du germe */
   ...
   srand(mongerme) ;
   ...
```

IMPORTANT: `srand()` ne doit être appelée qu'une fois avant le premier appel à `rand()`. Tout appel à `srand()` entre deux appels à `rand()` détruira les propriétés statistiques de votre séquence de nombres aléatoires. Vous n'avez pas vraiment à toucher au germe aux fins de ce Labo. En guise d'exemple d'utilisation de `rand()` et `srand()`, le petit bout de code suivant produit 10 nombres aléatoires entre zéro et un, et les renvoie à l'écran:

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
/* Declarations ---------------------------------------- */
   float r ;
   int i, n=10 ;
   int mongerme=1234 ;
/* Executable ------------------------------------------ */
   srand(mongerme) ;
   for (i=0 ; i<n ; i++) {
      r=1.*rand()/RAND_MAX ;
      printf ("Nombre aleatoire %d = %f\n",i,r) ;
   }
}
```

Tapez ce petit code, compilez le et assurez vous que tout se passe comme prévu.

6.4 Calcul du volume d'une hypersphère en D-dimensions

La première partie de ce labo vise à vous faire calculer le volume d'une hypersphère à l'aide de l'approche Monte Carlo. La logique est la même que celle expliquée au début du chapitre 5 des notes de cours dans le cas de l'exemple de l'étang. On considère un hypercube de coté $L = 2$ dans un espace 5-dimensionnel (v, w, x, y, z). Ce cube est centré sur l'origine $(0, 0, 0, 0, 0)$, et s'étend donc de -1 à 1 dans chacun des cinq axes de coordonnées (cartésiennes) de l'espace 5D. Le volume de l'hypercube est donc $2 \times 2 \times 2 \times 2 \times 2 = 32$! Une hypershère de rayon 1 centrée à l'origine du système de coordonnée sera donc complètement contenue dans l'hypercube, le touchant seulement au centre de ses faces du cube (combien de faces a un hypercube en cinq dimensions spatiales ...?).

Il s'agit maintenant de générer des points (v, w, x, y, z) uniformément distribués dans notre espace 5-dimensionnel; chaque coordonnée, devant être distribuée entre -1 et $+1$, sera donc produite par un appel à un générateur de nombre aléatoires distribué uniformément dans l'intervalle $[0, 1]$; par exemple,

```
float u, v, x, y, z ;
...
u=-1.+2.*rand()/RAND_MAX ;
v=-1.+2.*rand()/RAND_MAX ;
x=-1.+2.*rand()/RAND_MAX ;
y=-1.+2.*rand()/RAND_MAX ;
z=-1.+2.*rand()/RAND_MAX ;
```

Avant d'aller plus loin, assurez vous de bien comprendre pourquoi, si $0 \leq \texttt{1.} * \texttt{rand}()/\texttt{RAND\_MAX} \leq 1$, alors $-1 \leq \texttt{-1.} + \texttt{2.} * \texttt{rand}()/\texttt{RAND\_MAX} \leq 1$. Une fois ça compris, pour chacun de ces points, on calcule la distance d à l'origine; même dans un espace à cinq dimensions, cette distance est néanmoins donnée par:

$$d = \sqrt{v^2 + w^2 + x^2 + y^2 + z^2} \ . \tag{6.1}$$

Si et seulement si $d \leq R$, alors on ajoute 1 à une variable compteur, (préalablement initialisée à zéro); sinon on passe au point aléatoire suivant. Un canevas de code C pour cet algorithme pourrait avoir l'allure suivante:

```
#include <stdio.h>
#include <stdlib.h>
#define N 1000                        /* Nombre de points 5D generes */
int main(void)
{
/* Declarations ---------------------------------------- */
   float d, u, v, x, y, z, som ;
   int i ;
/* Executable ------------------------------------------ */
   som=0. ;                           /* initialisation de la variable-compteur */
   for (i=0 ; i<N ; i++) {
      u= ...                          /* generer un point 5D /*
      d = ...                         /* calculer la distance a l'origine */
      if (d < 1.) { som += 1. ; }  /* incrementation de la variable compteur */
   }
}
```

Remarquez qu'il est préférable ici de définir la variable `som` comme un `float`, au cas où `N` se retrouverait très grand. Il ne reste plus qu'à diviser `som` par `N` et multiplier par le volume de l'hypercube contenant la sphère pour obtenir un estimé Monte Carlo du volume de l'hypersphère. Je vous laisse imaginer de quoi devrait avoir l'air l'instruction en C effectuant ce dernier petit calcul...

La procédure pour le reste de cette partie du labo est maintenant la suivante:

1. Écrivez un code C utilisant l'approche Monte Carlo décrite ci-dessus pour calculer le volume d'une sphère en trois dimension spatiales;

2. Calculez le volume par Monte Carlo pour `N`= 10, 100, 1000, 10000, et 100000;

3. Estimez combien de points (x, y, z) vous devrez utiliser pour arriver au résultat connu ($V = 4\pi R^3/3$) à mieux de 1%. Surprise, si tout s'est bien passé vous venez de compléter l'étape de validation pour ce labo!

4. Répétez maintenant le calcul pour une hypersphère en cinq dimensions spatiales, toujours pour `N`= 10, 100, 1000, 10000, et 100000. Examinez comment la valeur calculée du volume varie en fonction de `N`.

5. Basé sur les résultats de votre calcul précédent en 3D, estimez la valeur de `N` requise pour arriver à une valeur du volume précise à mieux de 1%.

6. Pouvez vous "deviner" la valeur exacte du volume de l'hypersphère en 5D, en terme d'une fraction entière de π?

6.5 Calcul d'une intégrale multidimensionnelle par Monte Carlo

Nous allons revenir à un problème qui nous a déjà tenu fort occuppé(e), soit le calcul du potentiel gravitationnel à l'extérieur d'un objet d'étendue finie. Cette fois, plutôt qu'une tige, nous allons considérer un monolithe (noir) de forme rectangulaire parallépipédique, ayant un rapport d'aspect $x : y : z = 1 : 4 : 9$. La dernière partie de ce Labo consiste à vous faire calculer le potentiel gravitationnel à l'extérieur de ce mythique monolithe, de manière "standard", soit l'intégration par la règle du trapèze, ainsi que par intégration Monte Carlo, et de comparer les résultats.

Considérons le calcul du potentiel à un point (x, y, z) situé quelquepart à l'extérieur du monolithe. Si l'on fait coincider son centre avec l'origine de notre système de coordonnées cartésienne, le potentiel gravitationnel à (x, y, z) sera donné par l'expression:

$$\Phi(x,y,z) = -G\rho \int_{-4.5}^{4.5}\int_{-2}^{2}\int_{-0.5}^{0.5} \frac{\mathrm{d}x'\mathrm{d}y'\mathrm{d}z'}{\sqrt{(x-x')^2+(y-y')^2+(z-z')^2}} \ , \tag{6.2}$$

où les intégrales se font sur les coordonnées "prime" qui mesurent la position des petits éléments de volume infinitésimaux composant le volume du monolithe. Définissant un maillage cartésien approprié sur ces coordonnées (dimensions $N_x \times N_y \times N_z$), l'application de la méthode du trapèze en trois dimensions spatiales conduisait à:

$$\begin{aligned}
\Phi(x,y,z) = -G\rho \sum_{i=1}^{N_x}\sum_{j=1}^{N_y}\sum_{k=1}^{N_z} \frac{1}{8}\Big[& \quad f(x,y,z,x'_i,y'_j,z'_k) + f(x,y,z,x'_{i+1},y'_j,z'_k) && (6.3)\\
& + \quad f(x,y,z,x'_{i+1},y'_{j+1},z'_k) + f(x,y,z,x'_i,y'_{j+1},z'_k) && (6.4)\\
& + \quad f(x,y,z,x'_i,y'_j,z'_{k+1}) + f(x,y,z,x'_{i+1},y'_j,z'_{k+1}) && (6.5)\\
& + \quad f(x,y,z,x'_{i+1},y'_{j+1},z'_{k+1}) + f(x,y,z,x'_i,y'_{j+1},z'_{k+1}) \Big] && (6.6)\\
& \times \quad (x'_{i+1}-x'_i)(y'_{j+1}-y'_j)(z'_{k+1}-z'_k) && (6.7)
\end{aligned}$$

où la fonction f mesure la distance entre le point (x, y, z) où le potentiel est calculé, et le point (x', y', z') du petit élément de masse contribuant au potentiel; en trois dimension spatiales:

$$f(x,y,z,x',y',z') = \frac{1}{\sqrt{(x-x')^2+(y-y')^2+(z-z')^2}} \ , \tag{6.8}$$

La triple somme étant ici une réflexion directe de la tridimensionalité du maillage requis pour échantillonner le volume du monolithe. La fonction C à la page suivante effectue cette intégrale triple sur un maillage cartésien régulier de pas $\Delta x = \Delta y = \Delta z = 0.1$. Avant de continuer, assurez-vous de bien comprendre la correspondance entre cette fonction et l'éq. (6.3).

Il s'agit maintneant pour vous de produire un équivalent où l'intégration est effectuée par Monte Carlo. Vous vous rappelez certainement que l'idée de l'intégration Monte Carlo est d'échantillonner l'intégrand à un grand nombre (N) de positions choisies *aléatoirement* à l'intérieur du monolithe, et d'estimer le potentiel comme la moyenne de ces évaluations. Plutôt que l'éq. (6.3), on a maintenant:

$$\Phi(x,y,z) = -G\rho \frac{(L_x L_y L_z)}{N} \sum_{r=1}^{N} f(x,y,z,x_r,y_r,z_r) \ , \tag{6.9}$$

où la fonction f est la même qu'auparavant, $L_x = 1$, $L_y = 4$ et $L_z = 9$ sont les mesures du monolithe, et les x_r, y_r et z_r sont des nombres aléatoires tirés de distributions uniformes couvrant les intervalles:

$$x_r \in [-0.5, 0.5] \ , \qquad y_r \in [-2, 2] \ , \qquad z_r \in [-4.5, 4.5] \ . \tag{6.10}$$

```
#include <math.h>
#define NX 11          /* taille du maillage dans direction-x */
#define NY 41          /* taille du maillage dans direction-y */
#define NZ 91          /* taille du maillage dans direction-z */
float potgrav( float x, float y, float z)
{
/* Declarations ------------------------------------------- */
  float f( float, float, float, float, float, float ) ;
  int   i, j, k ;
  float xp[NX], yp[NY], zp[NZ], somme, pot, dx, dy, dz ;
  float G=6.67e-11, rho=1. ;
/* Executable --------------------------------------------- */
/* 1. calculer la maille 3D sur le monolithe */
  for (i=0 ; i<NX ; i++) { xp[i]=-0.5+1.0*fabs(i)/(NX-1) ; }
  for (j=0 ; j<NY ; j++) { yp[j]=-2.0+4.0*fabs(j)/(NY-1) ; }
  for (k=0 ; k<NZ ; k++) { zp[k]=-4.5+9.0*fabs(k)/(NZ-1) ; }
/* 2. calculer potentiel par integration, trapeze 3D */
  somme=0. ;
  for (i=0 ; i<NX-1 ; i++ ) {
    dx=xp[i+1]-xp[i] ;
    for (j=0 ; j<NY-1 ; j++ ) {
      dy=yp[j+1]-yp[j] ;
      for (k=0 ; k<NZ-1 ; k++ ) {
         dz=zp[k+1]-zp[k] ;
         somme+=(f(x,y,z,xp[i],  yp[j],  zp[k])
                +f(x,y,z,xp[i+1],yp[j],  zp[k])
                +f(x,y,z,xp[i],  yp[j+1],zp[k])
                +f(x,y,z,xp[i+1],yp[j+1],zp[k])
                +f(x,y,z,xp[i],  yp[j],  zp[k+1])
                +f(x,y,z,xp[i+1],yp[j],  zp[k+1])
                +f(x,y,z,xp[i],  yp[j+1],zp[k+1])
                +f(x,y,z,xp[i+1],yp[j+1],zp[k+1]))*dx*dy*dz ;
      }
    }
  }
  pot=-G*rho*somme/8. ; /* la valeur du potentiel... */
  return pot;           /* ...est renvoyee au programme principal */
}
float f( float x, float y, float z, float xp, float yp, float zp)
{
  float ff ;
  /* calcule l'inverse de la distance entre (x,y,z) et (x',y',z') */
  ff=1./sqrt( (x-xp)*(x-xp)+(y-yp)*(y-yp)+(z-zp)*(z-zp) ) ;
  return ff ;
}
```

Figure 6.1: Fonction C pour le calcul du potentiel gravitationnel produit à une position (x, y, z) par un parallépipède (noir) de rapport d'aspect 1:4:9 et de densité constante. Le calcul de l'intégrale est ici effectué à l'aide de la généralisation tridimensionnelle de la méthode du trapèze, avec l'origine du système de coordonnées (x', y', z') au centre du monolithe. Remarquez que les trois coordonnées (x, y, z) du point auquel le potentiel est calculé sont passées en argument à cette fonction.

Votre tâche est maintenant de créer une fonction (appelée `potgravmc`, tiens), qui fait ce calcul Monte Carlo. Cette nouvelle fonction est encore une fois appelée avec trois arguments, soit les valeurs x, y, z auxquelles on veut calculer le potentiel, et devrait pouvoir être substituée directement à la place de la fonction `potgrav` calculant la même chose par la méthode du trapèze (voir code page précédente). Cependant, le calcul effectué par la fonction même est maintenant passablement différent. Il n'y a plus de maillage en coordonnées "prime" qui doivent être défini à l'intérieur de `potgravmc`, plus de triples boucles, plus d'éléments de volume. Il n'y a plus que N évaluations de l'intégrand (essentiellement la fonction f ci-dessus), donc une seule boucle se répétant N fois. Vous pouvez néanmoins partir du code C de la page précédente, et le modifier en conséquence. Et pas de panique, tous les éléments requis ont déjà été couverts dans la première partie de ce Labo! Une fois la fonction écrite, les étapes à exécuter sont les suivantes:

1. Évaluez d'abord le potentiel à l'aide de la méthode du trapèze, à la position $(x, y, z) = (10, 10, 10)$; vous pouvez utiliser la fonction listée à la page précédente, avec un programme "`main`" approprié qui appelle simplement cette fonction aux coordonnées $(x, y, z) = (10, 10, 10)$. Assurez-vous de bien définir la fonction `potgrav` et ses arguments dans votre programme principal.

2. Faites de même pour la version Monte Carlo, toujours à $(x, y, z) = (10, 10, 10)$, et comme auparavant avec $N = 10$, $N = 100$, 1000, 10^4 et 10^5. Normalement, votre "`main`" devrait demeurer essentiellement identique au premier, sauf qu'il appellera cette fois la fonction `potgravmc` plutôt que `potgrav`;

3. Répétez les étapes 1 et 2, mais cette fois pour le potentiel à la position $(x, y, z) = (100, 100, 100)$;

4. Votre calcul Monte Carlo fonctionne-t-il mieux près ou loin du monolithe? Pourquoi?

Finalement, avant de quitter le lab vous devriez vous convaincre que votre calcul du volume de l'hypersphère revenait essentiellement au calcul d'une intégrale du genre:

$$I = \int_{-1}^{+1} \int_{-1}^{+1} \int_{-1}^{+1} \int_{-1}^{+1} \int_{-1}^{+1} g(u, v, x, y, z) \mathrm{d}u \, \mathrm{d}v \, \mathrm{d}x \, \mathrm{d}y \, \mathrm{d}z \ , \tag{6.11}$$

où l'intégrand $g(u, v, x, y, z)$ vaut un dans l'hypersphère et zéro à l'extérieur. Une fois convaincu(e).., le labo 6 est terminé.

Laboratoire 7

Interaction radiation-matière

Assurez-vous de relire attentivement les §5.2, 5.3 et 5.5 des Notes de cours avant de commencer ce lab.

7.1 Objectifs

1. Vous familiariser avec la description physique du passage de la radiation à travers la matière;
2. Apprendre à utiliser des nombres aléatoires pour définir des tests probabilistes.
3. Effectuer une véritable simulation Monte Carlo d'un processus physique stochastique.

7.2 Rapport de Lab

A remettre dans une semaine, comme d'habitude. Le rapport doit répondre aux questions posées aux sections 7.5 et 7.6.

7.3 Passage de la radiation corpusculaire à travers la matière

Que ce soit dans le cadre de la radioprotection ou de la radiothéraphie, l'étude expérimentale et la modélisation du passage de la radiation dans la matière a fait vivre plusieurs générations de physicien(ne)s depuis maintenant près d'un siècle. Le terme "radiation" est utilisé de manière tout à fait générique ici, et inclut donc les radiations électromagnétique et corpusculaire (faisceaux de particules élémentaires). La Figure 7.1 illustre l'idée générale. Un faisceaux de particules arrive ici de la gauche selon une trajectoire coincidant avec la ligne en tirets, et pénètre une plaque d'une substance quelconque (en gris). Les interactions avec les atomes composant la plaque peuvent causer des déviations de la trajectoire des particules incidentes, parfois suffisamment marquée pour les faire ressortir de la plaque du coté d'où elles étaient entrées. D'autres particules parviennent à traverser la plaque, parfois après plusieurs déviations, et d'autres se retrouvent absorbées dans la plaque (trajectoires se terminant sur un point noir). En pratique, on cherche à déterminer la fraction des particules qui (1) traverseront la plaque, (2) seront absorbées à l'intérieur de la plaque, et (3) qui seront réfléchies par la plaque. On définit donc les **transmissivité** (T), **absorptivité** (A) et **réflectivité** (R) comme les fractions de l'intensité du faisceau incident qui est transmise, absorbée et réfléchie, respectivement. Vous pouvez facilement imaginer que ces coefficients dépendent de plusieurs facteurs, dont évidemment l'épaisseur de la plaque et sa densité, mais aussi —et c'est là où physique du problème peut devenir très complexe— de la nature et énergie de la radiation incidente et des atomes absorbeurs.

Dépendant de la situation physique considérée, les objectifs peuvent être très divers:

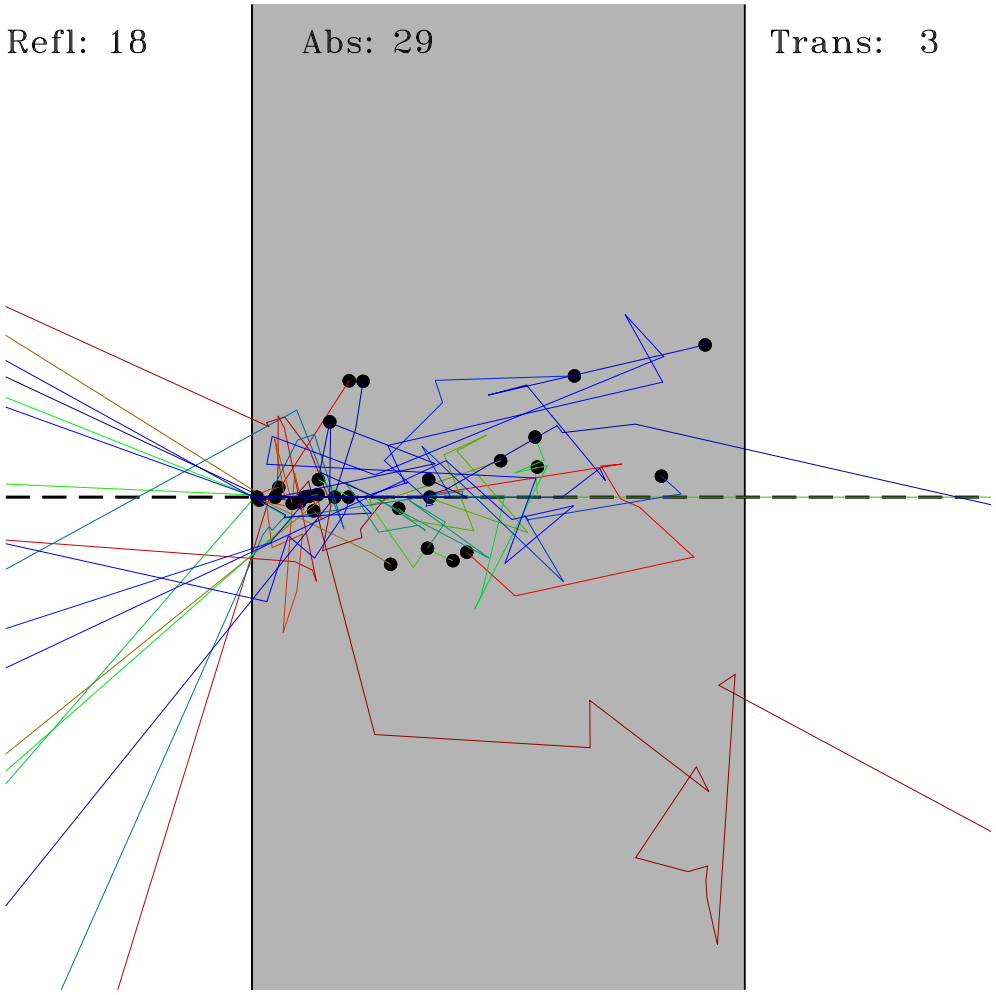

Figure 7.1: Réflexion, absorption et transmission d'un faisceau de neutrons traversant une plaque (en gris). Les neutrons arrivent de la gauche, selon la trajectoire indiquée en tirets. Les points noirs à l'intérieur de la plaque indiquent les positions où des neutrons ont été absorbés. Notez qu'ici que seuls trois neutrons ont réussi à traverser la plaque, dont un en ligne droite (trajectoire verte).

1. Blindage de protection autour d'un réacteur nucléaire: on veut maximiser la réflectivité, minimiser la transmissivité, et dans la mesure du possible l'absorption (afin de pas endommager ou rendre radioactif le blindage);

2. Barres de contrôle d'un réacteur nucléaire: produire une absorption qui soit à la fois élevée, et le plus homogène possible spatialement;

3. Transmutation nucléaire: maximiser absorption, et minimiser réflexion et transmission pour ne pas endommager le matériel ou les techniciens;

4. Radiothérapie: maximiser l'absorption, tout en contrôlant le mieux possible sa localisation spatiale.

Dans ce laboratoire vous développerez une simulation de type Monte Carlo visant à modéliser le passage d'un faisceau de neutrons à travers une plaque. Ce sera effectivement la même simulation que j'ai utilisée pour produire la Figure 7.1. Le choix du neutron comme particule du faisceau incident est motivé par la simplicité (relative) avec laquelle celui-ci interagit avec la matière. N'étant pas électriquement chargés, les neutrons n'interagissent qu'au moment de leur "collision" avec les noyaux atomiques[1]. L'absence de force électrostatique implique alors qu'entre ces interactions, les neutrons se déplacent en ligne droite. Ce n'est qu'à très courte distance du noyau que les forces nucléaires entrent en jeu, et que les neutrons peuvent être déviés ou absorbés.

7.4 Modélisation Monte Carlo

7.4.1 Formulation statistique

Un peu comme le problème de l'approche à l'équilibre (§5.5 des notes de cours), le problème du passage de la radiation à travers la matière peut se formuler en équations différentielles, mais nous utiliserons ici une approche de type Monte Carlo. Celle-ci est en fait souvent utilisée dans les calculs impliquant des faibles intensités du faisceau incident, des courts temps d'irradiation, ou encore des substances de faible absorptivité.

Dans le cadre de notre approche Monte Carlo, l'interaction neutron-matière sera quantifiée par seulement trois paramètres:

1. Le **libre parcours moyen** (symbole λ): c'est la distance moyenne traversée (en ligne droite) par un neutron entre deux interactions;

2. La **probabilité d'absorption** (symbole p_a, avec $0 \leq p_a \leq 1$): c'est la probabilité qu'un neutron soit capturé par un noyau au moment de l'interaction;

3. La **probabilité de dispersion** (symbole p_s, avec $0 \leq p_s \leq 1$; "scattering" en anglais... et dans bien des bouquins français!): c'est la probabilité qu'un neutron soit dévié de sa trajectoire au moment de l'interaction avec le noyau;

Notons qu'en général $p_a + p_s < 1$, car il est tout à fait possible qu'un neutron s'approche assez près d'un noyau pour ressentir la force nucléaire, mais sans interagir. Vous ferez connaissance avec ce genre de subtilités dans le cadre de vos études de la mécanique quantique de la structure nucléaire; dans le contexte de ce labo, nous considérerons ces trois paramètres comme données en entrées à la simulation.

[1] "Collision" a été délibérement mis entre guillemets, car l'interaction neutron-noyau ne peut vraiment pas être représentée comme une collision de type boules-de-billiard.

7.4.2 Le libre parcours moyen

Pour un solide amorphe (plutôt que cristallin), les atomes sont distribués localement aléatoirement (mais globalement uniformément), donc nous devrons extraire d'une distribution statistique appropriée la distance ℓ parcourue entre chaque interaction. On peut montrer que pour une distribution spatialement aléatoire mais uniforme globalement (i.e., même densité de noyaux partout dans la plaque), le libre parcours moyen assume une distribution exponentielle. Comme on l'a vu à la §6.3.1 des notes de cours, une telle distribution peut facilement être générée à partir d'une distribution uniforme selon la transformation:

$$\ell = -\lambda \ln r \ , \qquad r \in [0,1] \ , \tag{7.1}$$

où r est un nombre aléatoire entre 0 et 1; en C ça aurait l'air de ceci:

```
ell = -lambda* log( 1.*rand()/RAND_MAX  ) ;
```

où `lambda` est le libre parcous moyen introduit précédemment.

7.4.3 L'absorption

L'absorption se calcule facilement; on produit un nombre aléatoire r extrait d'une distribution uniforme dans l'intervalle $[0,1]$; si $r < p_a$, le neutron est absorbé; sinon il ne l'est pas. Facile:

```
if ( 1.*rand()/RAND_MAX <= pa ) { sa+=1 ; }
```

où `pa` est la probabilité de capture définie ci-haut, et `sa` est une variable-compteur comptabilisant le nombre de neutrons capturés.

7.4.4 La dispersion

La dispersion est un processus en deux étapes. On commence par un test probabiliste du même genre que décrit précédemment pour l'absorption, afin de décider si le neutron sera dévié ou non. Si c'est le cas, nous devons calculer un angle de déviation. Dans le cas de l'interaction d'un neutron avec un noyau atomique non-excité, tous les angles sont équiprobables. En trois dimensions spatiales, et travaillant en coordonnées sphériques avec comme axe de symétrie la direction horizontale (disons), la probabilité de dévier le neutron vers les angles polaire et azimutal (θ, ϕ) (genre latitude-longitude) est donnée par:

$$p(\theta,\phi) = \frac{\sin\theta}{4\pi} \tag{7.2}$$

(voir également la Figure 7.2); je vous laisse également le soin de vérifier qu'intégrer cette distribution sur la sphère conduit bien à $p = 1$. Comme la déviation azimutale ne nous intéresse pas particulièrement, il s'agit maintenant d'intégrer sur ϕ pour calculer la probabilité de déviation pour un angle polaire θ est:

$$p(\theta) = \int_0^{2\pi} p(\theta,\phi)\mathrm{d}\phi = \frac{1}{2}\sin\theta \ . \tag{7.3}$$

Comme on l'a vu à la §5.3 des notes, on a alors

$$r = \frac{1}{2}\int_0^{\theta} \sin(\theta')\mathrm{d}\theta' \ . \tag{7.4}$$

Ce qui permet de générer un angle θ distribué selon l'éq. (7.2) à partir d'un nombre aléatoire uniforme $r \in [0,1]$ selon la relation:

$$\cos\theta = 1 - 2r \ . \tag{7.5}$$

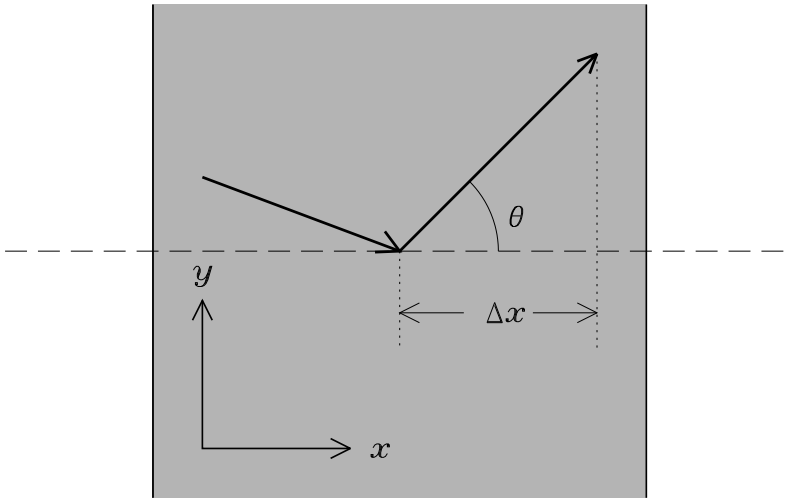

Figure 7.2: Définition de l'angle de dispersion. Ce dernier est toujours mesuré par rapport à la direction du faisceau incident, ici l'horizontale (tirets); si tous les angles de dispersion sont équiprobables, le choix de l'axe est arbitraire, mais le choix fait ici a l'avantage que l'avance du neutron dans la direction-x est alors simplement donnée par $\Delta x = \ell \cos\theta$, où ℓ est la distance parcourue entre l'interaction présente et la suivante.

Ultimement, ce qui nous intéresse est le déplacement du neutron dans la direction x; pour une distance de parcours ℓ et un angle de déviation θ, l'incrément de position dans la direction-x est donné par

$$\Delta x = \ell \cos\theta \ , \tag{7.6}$$

donc il n'est pas nécessaire de calculer l'arccosinus de l'éq. (7.5). Tout ça, en C, pourrait avoir l'air de:

```
if ( 1.*rand()/RAND_MAX <= ps )       /* dispersion se produit */
   {
      costheta = ( 1.- 2.*rand()/RAND_MAX  ) ; /* nouvel angle */
   }
```

où `ps` est la probabilité de dispersion, et la valeur de $\cos\theta$ ainsi calculée demeurera la même jusqu'à la prochaine fois où le test de dispersion sera satisfait pour le neutron dont on calcule la trajectoire.

7.4.5 L'algorithme

L'idée de la simulation est de "lancer" successivement N neutron à travers la plaque, et de faire un décompte du nombre s_r de neutrons réfléchis, du nombre s_c de neutron capturés, et du nombre s_t de neutron transmis; les réflectivité, absorptivité et transmissivité seront alors données par les expressions:

$$R = \frac{s_r}{N} \ , \qquad A = \frac{s_c}{N} \ , \qquad T = \frac{s_t}{N} \ . \tag{7.7}$$

L'algorithme de calcul est le suivant:

1. Initialisation des compteurs: $s_r = 0, s_c = 0, s_t = 0$.

2. Initialisation: le neutron est à $x = 0$ (surface de la plaque) et se déplace horizontalement ($\theta = 0$) vers la droite

3. Déplacement: Calculer ℓ via l'éq. (7.1), et la nouvelle coordonnée $x \to x + \Delta x$, avec Δx donné par l'éq. (7.6);

4. Test pour réflexion: si $x < 0$, ajouter 1 au compteur de réflexion s_r et passer au neutron suivant (étape 7);

5. Test pour transmission: si $x > d$, ajouter 1 au compteur de transmission et passer au neutron suivant (étape 7);

6. Tests de capture/dispersion: si $0 \leq x \leq d$:

   (a) Test de capture: si $r < p_a$, ajouter 1 au compteur capture et passer au neutron suivant (étape 7);

   (b) sinon, test de dispersion: si $r < p_s$, calculer un nouvel angle θ via l'éq. (7.5), et remonter à l'étape 3;

7. Répétez pour un nombre de neutron N.

Vous pouvez donc bien imaginer que le canevas de votre code sera construit sur deux boucles imbriquées, soit une boucle extérieure inconditionnelle sur le nombre N de neutrons envoyés contre la plaque, et une boucle intérieure conditionnelle (étapes 3 à 6) qui tournera tant et aussi longtemps que le neutron n'a pas traversé, ou n'a pas été absorbé ou réfléchi; ceci pourrait avoir l'air de:

```
#include <stdlib.h>
int main(void)
{
  int itermax=1000 ;
  ...
  sa=0 ; st=0 ; sr=0 ;        /* initialisation des compteurs */
  for (k=0 ; k<N ; k++ )      /* boucle sur les N neutrons */
    {
      x=0.  ;                 /* position initiale du neutron */
      costheta=1. ;           /* direction initiale du neutron */
      fin=0 ;                 /* variable-marqueur */
      j=0   ;                 /* compteur de securite */
      while ( fin == 0 && j<itermax ) /* deplacements dans la plaque */
        {
           ...                /* calculer et effectuer deplacement */
           ...                /* test pour reflexion */
           ...                /* test pour transmission */
           ...                /* test de capture/dispersion */
           j+=1
        }
    }
  printf (" Neutrons captures:  %d\n",sa) ;
  printf (" Neutrons transmis:  %d\n",st) ;
  printf (" Neutrons reflechis: %d\n",sr) ;
}
```

Notez ici la variable-marqueur `fin`, qui est initialisée à zéro au lancement de chaque neutron; l'idée sera d'inclure une instruction `fin=1 ;` dans le bloc d'instruction contrôlé par chaque énoncé conditionnel vérifiant si le neutron a été transmis, absorbé ou réfléchi. Ceci mettra alors fin à la boucle intérieure, et on passera "automatiquement" au neutron suivant. Notez également la variable-compteur `j`, qui est incrémentée à chaque itération de la bouble intérieure, et qui forcera cette boucle à stopper après 1000 itérations, par mesure de sécurité afin d'éviter les boucles infinies.

Vous pouvez utiliser le générateur générique `rand()` (suivant la procédure décrite à la §6.1 des notes de cours), ou encore le générateur `ran0` de *Numerical Recipes*, téléchargeable depuis la page web du cours.

7.4.6 Quelques tests de validation

Je vous le répéterai toute la session: vous devez valider vos codes sur des problèmes pour lesquels la réponse est connue! Allons-y donc pour quelques tests simples:

1. Si $p_c = 0$ et $p_s = 0$, alors tous les neutrons devront être transmis. Vérifiez que votre code les laisse tous passer!

2. Si $p_c = 1$, alors tous les neutrons devront être capturés. Vérifiez que votre code les capture tous!

3. Posez $p_s = 0.5$, $p_c = 0.25$, et $\lambda = 0.1$; roulez votre simulation pour 100 neutrons, et vérifiez que $s_r + s_a + s_t = 100$!

7.5 Absorption pure

Passons maintenant à un calcul Monte Carlo plus sérieux. Posons $p_s = 0$; les neutrons ne peuvent maintenant qu'être absorbés, ou traverser la plaque en ligne droite. Le nombre n de neutrons dans la plaque à une position $x + \lambda$ doit alors être égal au nombre à la position x moins le nombre capturés entre x et $x + \lambda$; on pourrait donc écrire:

$$n(x + \lambda) - n(x) = -p_c n(x)$$

Ceci ressemble fort à une discrétisation par différence finie avant d'une EDO du genre:

$$\frac{\mathrm{d}n}{\mathrm{d}x} = -\frac{p_c}{\lambda} n(x)$$

où l'on a associé le pas spatial au libre parcours moyen λ, Cette EDO a comme solution:

$$n(x) = N \exp\left(-\frac{p}{\lambda} x\right) \ .$$

où N est le nombre total de neutron entrant dans la plaque du coté gauche. Cette expression nous permet de prédire que la fraction de neutron transmis à travers une plaque d'épaisseur d doit être donnée par

$$\frac{s_t}{N} = \exp\left(-\frac{p_c}{\lambda} d\right) \ .$$

Posez maintenant $p_s = 0$, $p_c = 0.2$, $\lambda = 0.2$, et $d = 1$, et déterminez combien par essai-erreur combien de neutrons (N) vous devez lancer pour arriver à $N/s_t = e = 2.7182818285...$ à mieux de 1%. Vous êtes maintenant en possession d'un algorithme Monte-Carlo vous permettant de calculer le nombre irrationnel e, à ajouter à l'algorithme du chapitre 1 des notes de cours, qui calcule π par Monte Carlo!

7.6 Effet de la dispersion

Il s'agit maintenant d'étudier l'influence de la dispersion. La procédure est la suivante:

1. Toujours avec $\lambda = 0.2$ et $p_c = 0.2$, imbriquez votre simulation dans une boucle extérieure faisant varier p_s de zéro à un en incrément $\Delta p_s = 0.1$.
2. Pour chacune de ces valeurs de p_s calculez les coefficients T, P et A, et portez le tout en graphique en fonction de p_s. Comment expliquez vous l'allure de ces courbes?
3. Répétez l'étape précédente pour $\lambda = 0.4$.
4. Basé sur vos résultats, si vous aviez à fabriquer une blaque de blindage neutronique devant minimiser la transmissivité (en priorité) et au mieux possible l'absorption, quelles seraient les valeurs optimales de p_s et p_c, si vous avez de plus la possibilité de choisir un matériau ayant soit $\lambda = 0.2$ ou 0.4?

Voilà, fini pour le Labo 7. Il n'en reste plus que trois!

Lectures supplémentaires:

Ce labo est inspiré en partie de la section 11.6 de l'ouvrage de Gould & Tobochnik cité en bibliographie au chapitre 1 des notes de cours. Vous bénéficierez également d'une relecture posée et attentive du chapitre 5 des Notes de cours.

Laboratoire 8

Irradiance solaire

8.1 Objectifs

1. Apprendre des notions de base en traitement de signal
2. Commencer à apprivoiser les séries de Fourier
3. Approfondir votre compréhension du contrôle des boucles et énoncés conditionnels en C
4. Apprendre la manipulation de fichiers en entrée/sortie en langage C

8.2 Rapport de Lab

Votre rapport de Lab est à remettre la semaine prochaine **au début** de votre prochain laboratoire. Il doit inclure des réponses (plus graphiques et listing des codes source) des questions posées aux sections 8.5 et 8.6

8.3 L'irradiance solaire

L'irradiance solaire est définie comme le flux énergétique total (Watt par mètre carré) de radiation électromagnétique solaire, intégré sur toute les longueurs d'ondes, incident au haut de l'atmosphère terrestre quand la Terre est à sa distance moyenne nominale du soleil, soit une unité astronomique (149,598,599 km). Déjà au dix-neuvième siècle, les mesures prises du sol de cette "constante solaire" avaient démontré que l'irradiance est affectée par l'atmosphère terrestre, et qu'une mesure précise ne pouvait être effectuée que de l'espace. Depuis 1978, une série de satellite en orbite aurtour de la Terre mesurent cette quantité de façon à peu près continue. La Figure 8.1A montre les moyennes journalières ainsi mesurées, de 1978 à 2007. La constante solaire ne l'est en fait pas du tout, montrant plutôt des variations sur des échelles journalières atteignant $4\,\mathrm{W\ m^{-2}}$, et une modulation décennale d'environ $1.5\,\mathrm{W\ m^{-2}}$.

La partie B de la Figure 8.1 montre un zoom sur une petite partie de la séquence, indiquée par l'encadré en A. On y remarque deux chutes marquées de l'irradiance, aux dates ~ 2000.73 et ~ 2001.25. Ces chutes, d'environ 2 Watt par mètre carré par rapport au niveau "moyen", sont nettement plus substantielles que les "fluctuations" d'ordre $\pm 0.5\,\mathrm{W\ m^{-2}}$ caractérisant les autres périodes de la séquence. Leur origine peut être retracée au passage d'un groupe de grandes taches solaires sur la partie du disque solaire illuminant la Terre. Le Soleil, comme la Terre, est imbu d'un mouvement de rotation, mais un "jour" solaire correspond à environ 28 jours terrestres. "Environ", car la surface du soleil est fluide, et il s'avère que ses régions équatoriales tournent environ 30% plus rapidement que les régions polaires; la période de rotation dépend donc de la latitude à laquelle on la mesure! Quoiqu'il en soit, vu de la terre une tache solaire demeurant à une latitude/longitude héliosphérique fixe semblera dériver de gauche

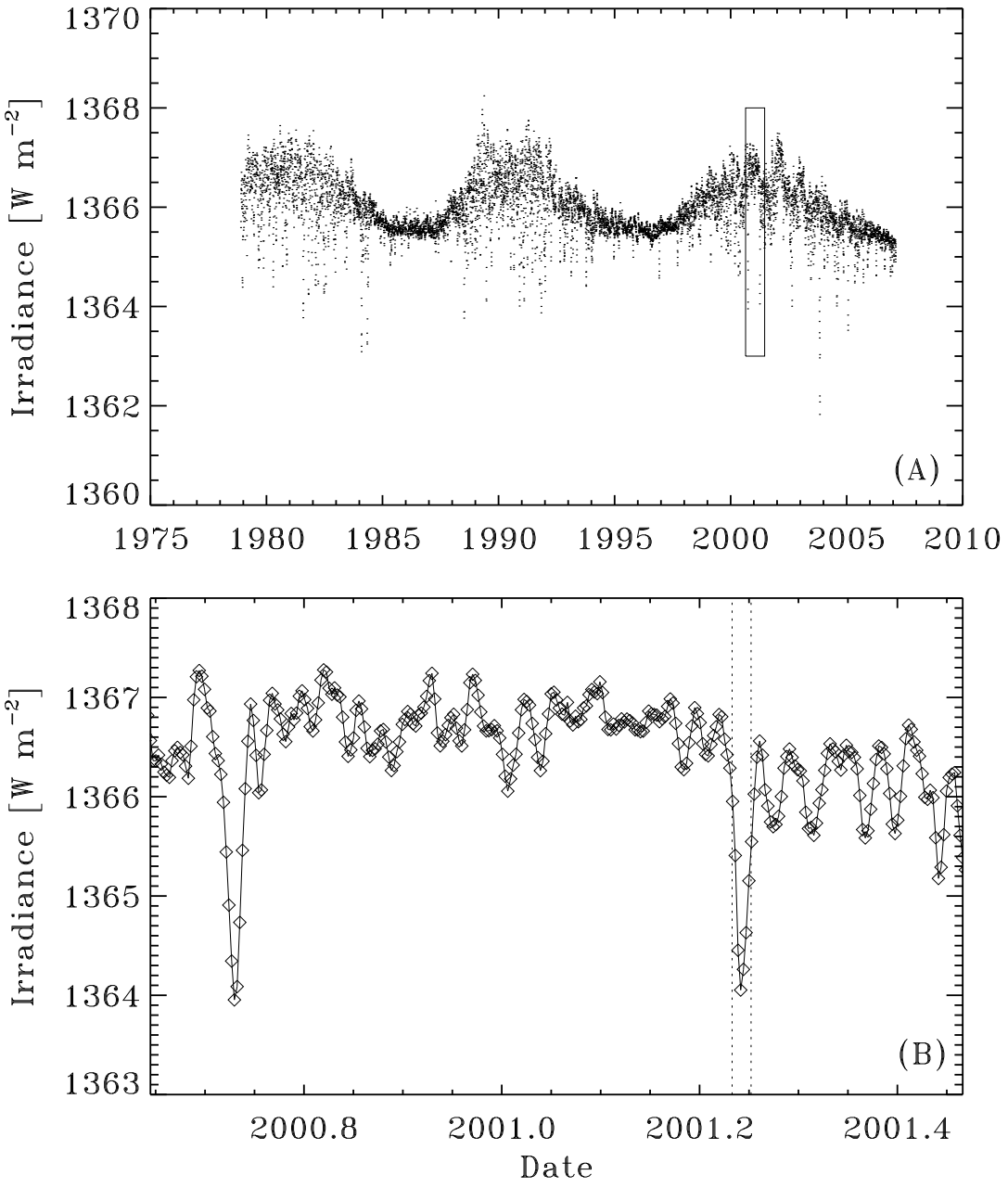

Figure 8.1: (A) Variations de l'irradiance solaire totale en fonction du temps, de 1978 au début 2007. Il s'agit ici du composite `d_41_61_0702` produit et distribué par le Physikalisches-Meteorologisches Observatorium à Davos, en Suisse. Cette séquence temporelle de l'irradiance solaire est composée de milliers de points de données, soit un point par jour, chacun correspondant à la moyenne d'une dizaine de mesures distinctes, et la précision relative associée à chaque point est inférieure à l'épaisseur du trait formant le cadre du graphique. (B) Zoom sur la partie de la séquence encadrée en (A). Les deux chutes marquées de l'irradiance aux dates $\sim$ 2000.73 et $\sim$ 2001.25 sont produites par le passage, sur la partie du disque solaire faisant face à la Terre, d'un groupe de grandes taches solaires (voir aussi Fig. 8.2).

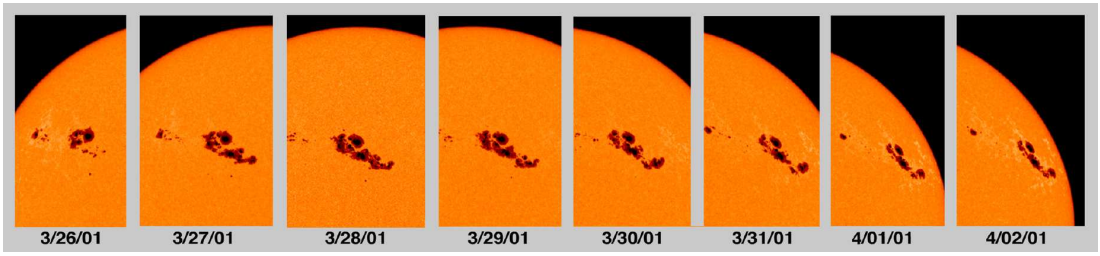

Figure 8.2: Déplacement d'Est en Ouest d'un groupe de taches solaires causé par la rotation axiale du soleil. La séquence couvre une période s'étendant du 26 mars au 2 avril 2001, à une cadence d'une journée. L'intervalle de temps correspondant est indiqué par deux traits verticaux pointillés sur la Figure 8.1B. Image disponible publiquement sur le site web `http://sohowww.nascom.nasa.gov`, merci à la NASA.

à droite d'une journée à l'autre, comme l'illustre la Figure 8.2. C'est d'ailleurs en observant ce déplacement que Fabricius et Galilée, déjà en 1610, avaient découvert la rotation du Soleil.

Comme les taches solaires sont plus sombres que l'atmosphère environnante, le passage d'un groupe de taches causera une baisse de l'irradiance solaire. D'ailleurs, c'est précisément le passage du groupe de taches visible sur la Figure 8.2 qui est responsable de la chute de l'irradiance par $\sim 2\,\mathrm{W\ m^{-2}}$ si prononcée à la date 2001.25 sur la Figure 8.1B. Comme on peut déjà le constater sur la Fig. 8.1B, la durée typique de tels épisodes de baisse de l'irradiance est d'environ la la moitié de la période de rotation solaire, soit 14 jours. De surcroit, tout groupe de tache survivant plus d'une rotation solaire produira un "signal" récurrent sur une période de 28–30 jours, dépendant de sa latitude.

Nous avons donc déjà détecté un "temps caractéristique" dans les variations de l'irradiance, mais même un coup d'oeil rapide à la Figure 8.1 révèle des variations se développant sur des échelles de temps beaucoup plus longues. Il s'agira, dans ce laboratoire, de développer des techniques nous permettant de "détecter" de telles variations. Mais, d'abord et avant tout, il faut apprendre comment lire dans un programme en C des données numériques contenues dans un fichier, pour manipulations subséquentes; avec plus de 12000 points de données sur la Figure 8.1A, l'entrée interactive à l'aide de l'instruction `scanf` serait absolument débile, donc il faut procéder autrement.

8.4 Lire des données numériques sur fichier dans un code C

La première chose que vous devrez faire est de copier les données de la Figure 8.1 dans votre répertoire local pour le labo 8. Ces données sont disponibles sur la partition Web de Mononcle Paul, et la manière la plus rapide de les ramasser est de taper:

```
wget http://www.astro.umontreal.ca/~ paulchar/phy1234/labos/tsi.dat
```

Ce qui devrait copier le fichier dans le répertoire où vous tapez la commande. Vous devriez aussi copier le fichier `tsiREADME.txt`, qui donne quelques informations intéressantes sur l'origine de ces données.

Vous pouvez examiner le contenu de ce fichier via la commande Linux `more`. Il consiste en trois colonnes de nombres. Le premier est un entier de six chiffres encodant la date en format AAMMJJ (deux derniers chiffres de l'année, le mois, le jour). La second est un numéro de jour, de type `float` avec le 1 janvier 1980 comme point zéro; donc, si t est la valeur lue, la date annuelle décimale (d) est donnée par:

$$y = 1980. + t/365.25$$

en unités d'années. Le troisième nombre est l'irradiance mesurée (S), aussi de type `float` et en unités de Watt par mètre carré. NOTEZ BIEN. Il y a certains jours pour lesquels aucune mesure n'est disponible (panne d'instrument, etc.); à ces jours, on a assigné à l'irradiance une valeur arbitraire de -99.0000.

Il s'agit maintenant de voir comment, en langage C, on peut lire ce fichier et emmagasiner les valeurs de t et S dans des tableaux unidimensionnels. Le petit bout de code C qui suit effectue ceci, et présuppose que le fichier de données est déjà copié dans le répertoire courant sous le nom `tsi.dat`:

```
#include <stdio.h>
#define NDATAMAX 12000
int main(void)
{
/* Ce programme est un exemple de lecture de fichier */
/* Declarations ----------------------------------  */
   int  ndata=0 ;
   float t[NDATAMAX], s[NDATAMAX] ;
   int   dummy ;
   FILE *fd ;
/* Executable ------------------------------------  */
   fd=fopen("tsi.dat","r") ;
   while ( !feof(fd) && ndata < NDATAMAX )
     {
       fscanf (fd, "%d %f %f\n", &dummy, &t[ndata], &s[ndata] ) ;
       ndata+=1 ;
     }
   if ( ndata == NDATAMAX ) { printf ("NDATAMAX atteint\n") ; }
   fclose(fd) ;
}
```

Il y a plusieurs choses à noter ici:

1. L'ouverture d'un fichier de données en mode lecture commence par la spécification d'un descripteur de fichier avec l'instruction `FILE`; attention, le "`*`" devant le nom du descripteur de fichier (ici `fd`) est important, il indique que `fd` est un *pointeur*, un type particulier de variable en C; ouverture avec `fopen`, avec le paramètre `"r"` pour "read" (ce serait `"w"`, pour "write", dans le cas d'un fichier où le code devrait écrire des données); et fermeture du fichier avec `fclose` une fois la lecture terminée.

2. La lecture même se fait avec l'instruction `fscanf`, le pendant de `scanf` pour la lecture de fichier plutôt que du clavier. Le premier argument de `fscanf` **doit** être le descripteur de fichier `fd` défini par `FILE` initialisé par l'instruction `fopen`. Si vous lisez des données provenant de fichiers différents, chacun doit être ouvert avec `fopen` mais en assignant un descripteur distinct pour chaque fichier.

3. Comme on ne sait pas d'avance, en général, combien de lignes de données on va lire, on définit les longueurs des tableaux à une taille raisonnablement élevée (ici 12000), et on ajoute un test à la condition de lecture pour empêcher les débordement de tableaux. On ajoute également un test, après la boucle de lecture, visant à prévenir l'usager si cette taille maximale est atteinte (dans lequel cas il manquera des données, et on devrait ici augmenter la valeur de `NDATAMAX` dans l'instruction `#define` en en-tête au programme, recompiler et re-exécuter.).

4. Examinez bien la condition contrôlant la boucle `while` lisant les données. La fonction-macro C `feof(fd)` évaluera à `FAUX`, donc `!feof(fd)` à `VRAI`, jusqu'à ce qu'un caractère

"fin-de-fichier" ait été rencontré à la lecture du fichier identifié par le descripteur `fd`. Ce caractère est ajouté automatiquement chaque fois que vous sauvegardez un fichier.

5. À la sortie de la boucle de lecture, la variable d'incrément `ndata` aura une valeur égale au nombre de points de données dans la séquence temporelle lue (mais le dernier élément est le `[ndata-1]`, car C commence sa numérotation des éléments de tableaux à 0, pas 1; encore une fois, ATTENTION!). La valeur de `ndata` sera utile par la suite...

6. La première colonne du tableau est lue dans une variable "fantôme" appelé `dummy` (type `int`), plutôt que dans un élément de tableau; puisqu'elle ne sera pas utilisée par la suite, pas besoin de l'emmagasiner.

Votre première tâche est de reproduire la Figure 1. Il s'agira de modifier le petit bout de code ci-dessus afin de:

1. Convertir le numéro de jour en date décimale, comme décrit plus haut;

2. Insérer les instructions `PLPLOT` appropriées.

Bien des choses bizarres peuvent se passer quand on lit des données d'un fichier, donc il est primordial, avant de commencer de jouer avec les données, de bien vérifier ce qu'on a lu; porter les variables lues en graphiques est souvent une façon aisée de détecter des aberrations introduites au moment de la lecture. Une autre approche courante est d'insérer une instruction `printf` qui donne à l'écran un "écho" des variables lues à mesure que ces variables sont lues, ligne par ligne. On n'est jamais trop prudent...

8.5 Lissage

Une première approche "'visuelle" à la détection de tendances à long terme dans une séquence temporelle consiste à effectuer un lissage qui élimine les variations rapides, quelles que soient leurs amplitudes, afin de laisser émerger des variations de plus faibles amplitudes se développant sur de longues échelles de temps. L'approche la plus simple consiste à **filtrer** la séquence. Le filtre le plus simple est la **moyenne courante** (*boxcar average* en anglais). Considérons le k-ième point (S_k) de la séquence; on remplace la valeur de ce point par une moyenne (S_k^*) des W/2 points précédents et suivants:

$$S_k = \frac{1}{W+1} \sum_{j=k-W/2}^{k+W/2} S_j \ , \qquad k = W/2, ..., N - W/2$$

où N est le nombre de points dans la séquence temporelle. On parle alors d'un filtre carré de largeur W (bien que la moyenne se fasse sur $W + 1$ points!).

Il s'agir maintenant pour vous de filtrer de cette façon les données de la Figure 8.1A. Le processus de filtrage s'effectue via deux boucles imbriquées, la boucle extérieure balayant les N données originelles et la boucle intérieur calculant la moyenne via l'expression ci-dessus; en C ca pourrait avoir l'air de quelquechose comme:

```
W=100 ;                                /* largeur du filtre de moyennage */
for ( k=W/2 ; k<N-W/2 ; k++ ) {        /* boucle sur les donnees */
   sum=0. ;
   for ( j=k-W/2 ; j<=k+W/2 ; j++ ) {  /* calcul de la moyenne */
      sum+=S[j] ;
   }
   Setoile(k)=sum/(W+1) ;              /* moyenne au temps k */
}
```

Notez les points suivants:

1. Comme le filtre doit "reculer" de W points de données pour calculer la moyenne au temps k, il n'est pas possible de calculer la moyenne courante pour les points de données ayant $k < W/2$ sans causer de débordement de tableau (aller chercher un élément avec un k négatif); tintin pour les derniers $W/2$ points de la séquence; c'est pourquoi la boucle extérieure ne tourne que de $k = W/2$ à $k = N - W/2$, plutôt que de 0 à $N - 1$;

2. La boucle intérieure doit centrer sa somme sur le point k; c'est pourquoi k apparait explicitement dans le contrôle de la boucle intérieure (qui utilises j comme indice).

3. La variable d'accumulation `sum` doit impérativement être réinitialisée à zéro chaque fois qu'on passe à un nouveau point de données; l'instruction `sum=0.` est par conséquent incluse dans la boucle extérieure.

4. La valeur moyennée à k doit être entreposée dans un tableau différent (ici le tableau `Setoile`) plutôt que d'être réinsérée directement à la position k de celui contenant les données originelles (ici `S`). Sinon le calcul de la moyenne au temps k utiliserait les $W/2$ valeurs moyennées précédemment, plutôt que les données véritables.

Dans le cas des données d'irradiance de la Figure 8.1A, il y a une complication supplémentaire: à certains jours il n'y a pas de données ($S = -99.000$ dans le fichier); ces valeurs doivent être exclues du calcul de la moyenne, à l'aide d'une instruction `if` appropriée.

Il s'agit maintenant de calculer une version lissée des données de la Figure 8.1; faites ça pour une fenêtre de lissage de largeur $W = 30$, $W = 81$ et $W = 365$, et superposez ces versions lissées (traits en couleurs) aux données originelles (points). Ceci vous permet-il de relever certaines périodicités? Lesquelles?

8.6 Analyse de Fourier

Parmi les diverses techniques permettant d'extraire des périodicités d'une séquence temporelle, la **Transformée de Fourier** est certainement en tête de liste. Cette dernière partie du labo vise donc à vous introduire à cette technique d'applicabilité très générale.

Comme vous aurez l'occasion de l'explorer à fond en PHY-1620, toute fonction présumée continue d'une variable (t, disons) peut être représentée comme une somme de fonctions harmoniques, appelée **série de Fourier**:

$$f(t) = \frac{a_0}{2} + \sum_{k=1}^{\infty} \left(a_k \cos(\omega_k t) + b_k \sin(\omega_k t) \right) \ , \tag{8.1}$$

où

$$\omega_k = k\omega_0 \ , \qquad \omega_0 = \frac{2\pi}{T} \ , \tag{8.2}$$

et la fréquence fondamentale ω_0 est déterminée par l'intervalle $[0, T]$ sur lequel on veut ainsi représenter $f(t)$. Les coefficients numériques a_k et b_k sont donnés par les intégrales suivantes:

$$a_k = \frac{2}{T} \int_0^T f(t) \cos(\omega_k t) \mathrm{d}t \ , \tag{8.3}$$

$$b_k = \frac{2}{T} \int_0^T f(t) \sin(\omega_k t) \mathrm{d}t \ , \tag{8.4}$$

avec le coefficient $a_0/2$ dans l'éq. (8.1) correspondant alors à la valeur moyenne de $f(t)$ sur l'intervalle temporel considéré, soit $t \in [0, T]$. Notez que du point de vue de la variable d'intégration (t), la fréquence ω_k est une constante. Mais on associe un a_k et un b_k à chaque fréquence ω_k.

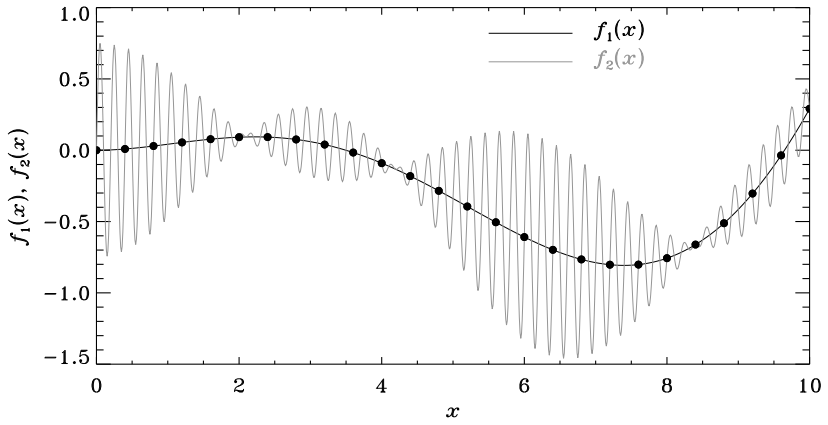

Figure 8.3: Illustration de l'idée de la fréquence de Nyquist. On peut toujours ajouter à une fonction quelconque n'importe quelle fonction harmonique de fréquence égale à un multiple de la fréquence de Nyquist associée au pas d'échantillonnage, sans rien changer aux valeurs échantillonnées de la fonction (voir texte).

Considérons maintenant une situation où $f(t)$ est échantillonnée à un nombre fini de points t_i:

$$f(t) \to f(t_i) \equiv f_i \ , \qquad t_i \in [0, T] \ ; \tag{8.5}$$

Dans le cas qui nous préoccupe, les paires (t_i, f_i) correspondent à chaque point de données sur la Figure 8.1. La fréquence fondamentale est toujours donnée par $\omega_0 = 2\pi/T$, mais il est maintenant superflu de calculer la somme dans l'éq. (8.1) jusqu'à $k \to \infty$; en effet, la discrétisation de t impose une fréquence maximale qui puisse être mesurée dans les données. Ceci est illustré sur la Figure 8.6. Les points noirs correspondent à une série de "données" dont on veut calculer la transformée de Fourier. Imaginons maintenant que ces données représentent un échantillonnage discret à pas constant h d'une fonction continue. Cette fonction pourrait bien avoir l'air du trait noir sur la Figure, qui est le genre de truc que vous pourriez obtenir via interpolation par splines cubiques sur les données discrètes. Cependant, les données pourraient tout aussi bien avoir été produites par la fonction correspondant au trait gris, qui offre une représentation tout aussi exacte des données que le trait noir. Il est essentiel de bien apprécier que *les données ne contiennent pas d'information nous permettant de distinguer entre ces deux possibilités*. Plus précisément, toute fonction harmonique ayant une période plus petite que *deux* points de maille ne laissera aucune trace cohérente dans les données. Cette fréquence de coupure est donc donnée par

$$\omega_c = \frac{\pi}{h} \ , \tag{8.6}$$

et est appellée **fréquence de Nyquist**. C'est la fréquence à laquelle la somme doit être tronquée dans l'éq. (8.1); la valeur correspondante de k est facile à calculer (faites-le!):

$$k_c = \frac{T}{2h} \ . \tag{8.7}$$

Dans votre cas, le h correspond à l'intervalle de temps entre deux points de données successifs, soit 1 journée pour les observations de la Fig. 8.1.

Mais quel peut bien être l'avantage de représenter nos données en termes d'une série de Fourier? L'attrait vient du fait que pour un signal multipériodique qui ne contient que quelques modes ayant des périodes bien définies, seul quelques coefficients a_k et b_k auront une amplitude

substantielle, et les fréquences correspondantes seront celles présentes dans le signal. Une telle **analyse de Fourier** offre donc une approche très efficace pour identifier les périodes "naturelles" présentes dans un signal multipériodique.

Nous allons évidemment commencer par une validation utilisant un signal bi-périodique. Utilisant les t_k (en version originale mesurés en jours) associés aux données de la Figure 8.1A, construisez un tableau `bidon[N]` contenant les valeurs:

$$B(t_k) \equiv B_k \equiv \texttt{bidon}[k] = 5\sin\left(\frac{2\pi}{30}t[k]\right) + 3\cos\left(\frac{2\pi}{100}t[k]\right) \ , \qquad k = 0, ...N-1$$

Il s'agira de calculer la transformée de Fourier de ce signal artificiel, et de vérifier que les deux fréquences qui en ressortent sont bel et bien $2\pi/30$ et $2\pi/100$. Les étapes sont les suivantes:

1. Calculez la fréquence fondamentale ω_0, la fréquence de Nyquist ω_c, et le nombre critique k_c pour les données de la Figure 8.1.

2. Travaillant avec vos données bidon, calculez les coefficients a_k et b_k définis par les éqs. (8.3) et (8.4). Utilisez la méthode du trapèze (§5.5 des Notes de cours, et plus spécifiquement l'éq. (5.25)) pour calculer ces intégrales. Comme il s'agit ici d'intégrales d'une seule variable (ici le temps t), le calcul de ces intégrales numérique n'impliquera qu'une seule boucle. Vous devrez cependant calculer les a_k et b_k pour chaque multiple de la fréquence fondamentale, jusqu'au k_c correspondant à la fréquence de Nyquist. Vous aurez donc une boucle extérieure sur les fréquences, et une boucle intérieure calculant les intégrales.

3. Calculez maintenant la quantité:

$$P(\omega_k) \equiv P_k = a_k^2 + b_k^2 \ .$$

4. Portez en graphique $P(\omega_k)$ versus ω_k; ceci s'appelle un **spectre de puissance**.

5. Identifiez et notez bien les fréquences correspondantes aux pics observés dans votre spectre de puissance. Correspondent-ils bien aux deux fréquence angulaires attendues?

Il s'agit finalement de répéter le procesuss ci-dessus, mais sur les données d'irradiance de la Figure 8.1. Vous devrez d'abord calculer la valeur moyenne de S sur l'ensemble de la séquence, et soustraire cette moyenne à chaque point de donnée S_k; ceci revient à se débarasser du terme a_0 dans l'éq. (8.1). Attention à bien éliminer du calcul de la moyenne et des intégrales les points pour lesquels il n'y a pas de mesures ($S_k = -99.0000$). Un `if` y suffit... Enfin, répondez aux questions suivantes:

1. Pouvez vous identifier, dans votre spectre de puissance, un pic correspondant au temps caractéristique de ~ 14 jours associé au passage d'un groupe de taches sur la partie visible du disque?

2. Pouvez vous identifier d'autres périodicités préalablement identifiées par lissage?

3. Pouvez vous identifier des périodicités associées à la rotation du soleil (28 jours à l'équateur solaire, vu de la Terre).

Et voilà, le labo 8 se termine ici!

Lectures supplémentaires:

Notes de cours: Section 2.4

Delannoy: Sections 7.6, 7.7, 8.7

Laboratoire 9

Diffusion et agrégation

9.1 Objectifs:

1. Explorer le lien entre la marche aléatoire sur réseau et la diffusion.
2. Approfondir vos habiletés à manipuler et utiliser des tableaux multidimensionnels en C
3. Bâtir une structure fractale par agrégation limitée par la diffusion

9.2 Rapport de Laboratoire

La rapport doit contenir des réponses, codes et résultats numériques à l'appui, aux questions posées aux §9.3 et 9.4, et doit être dans une semaine, avant le début du prochain lab.

9.3 Marche aléatoire 2D sur réseau

Nous avons vu au chapitre 7 des notes de cours que la marche aléatoire est la représentation à l'échelle microscopique des processus qu'on nomme diffusion du point de vue de l'échelle macroscopique. Du point de vue pratique la situation se corse si l'on désire simuler des systèmes diffusifs où des interactions entre les constituants microscopiques doivent être prises en considération. De telles situations incluent les problèmes d'agrégation couverts à la §7.5 des notes, mais aussi les réactions chimiques en milieux fluides (entre autres). Dans de tels systèmes, les interactions dépendent souvent des distances interparticules, et le calcul de ces distances pour un ensemble de marcheurs aléatoires "classiques" peut devenir très couteux en temps de calcul, quand le nombre de marcheurs devient grand. La **marche aléatoire sur réseau** (voir §7.4 des notes) contourne ce problème en restreignant la position des particules à un réseau cartésien, et une interaction ne se produit que si deux particules se trouvent sur des sites voisins, ce qui est très facile et rapide à calculer.

Ce Labo vise (entre autre) à vous faire explorer les limites de l'équivalence entre la marche aléatoire sur réseau et la diffusion. Vous accomplirez ceci en simulant de telles marches, et en examinant les propriétés statistiques des déplacements parcourus, un peu comme nous avions fait pour la marche alétoire en 1D à la §7.2 des notes. Le résultat-clef était l'augmentation du déplacement quadratique moyen d'un groupe de marcheurs selon la relation

$$\left\langle D_n^2 \right\rangle = n\, s^2 \ , \tag{9.1}$$

où n est le nombre de pas effectués. Il s'agit donc de simuler la marche aléatoire sur réseau 2D. Le canevas global de votre code de simulation est semblable à celui déjà introduit à la §7.2 des notes: une boucle extérieure sur le nombre (`NM`) de marcheurs, dans laquelle est imbriquée une boucle sur le nombre (`NPAS`) de pas effectués par chaque marcheur:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define NM   100
#define NPAS 1000
/* Marche aleatoire sur reseau 2D: NM marcheurs font NPAS pas     */
int main(void)
{
/* Declarations ------------------------------------------------- */
  int pas, x, y, j, k, dfin[NM] ;
/* Executable --------------------------------------------------- */
  for (j=0 ; j<NM ; j++)              /* boucle sur les marcheurs */
    {
      x=0 ; y=0 ;           /* Tous les marcheurs partent a x=y=0 */
      for (k=0 ; k<NPAS ; k++)              /* boucle sur les pas */
        {
          ...       /* choix d'un pas vers un des 4 sites voisins */
          ...                           /* deplacement du marcheur */
        }
      dfin[j] = x*x+y*y ;    /* calcul du deplacement quadratique */
    }
}
```

Comme on lance ici un marcheur à la fois, on peut considérer que les sites voisins sont toujours libres, donc pas besoin de tester si le déplacement est valide (cf. la Fig. 7.13 des notes), il le sera toujours. La seule difficulté pratique ici est de choisir aléatoirement un de 4 sites voisins; l'approche la plus simple est probablement d'utiliser un test probabiliste (genre `if (1.*rand()/RAND_MAX <= 0.5) {...}`) pour décider si le déplacement se fera en x ou en y, et une fois ce choix fait, utiliser le truc décrit à la §7.2 des notes pour choisir aléatoirement -1 ou $+1$ de manière équiprobable, et effectuer ce pas dans la direction déterminée précédemment. Remarquez que le canevas ci-dessus inclut la définition d'un tableau `dfin` qui calcule le déplacement quadratique à la fin de la marche, pour chacun des marcheurs. Ceci sera utile pour les analyses que vous devrez faire sous peu. Votre protocole de simulation est le suivant:

1. Lancez 1000 marcheurs faisant chacun 1000 pas sur un réseau dans le plan $[x, y]$, et débutant tous à $(x, y) = (0, 0)$. Tracez, avec des appels appropriés aux fonctions PLPLOT, la position de vos 1000 marcheurs aux pas 10, 30, 100, 300 et 1000 (genre, un point-symbole par marcheur, une couleur différente pour chacun des 5 pas de temps).

2. Votre nuage de points a-t-il une forme et une évolution temporelle "raisonnable" pour un processus diffusif? Justifiez votre réponse en quelques lignes.

3. Si la marche aléatoire sur réseau 2D est équivalente à la diffusion, alors le déplacement quadratique moyen de l'ensemble de vos marcheurs devrait varier comme $\sqrt{n}$. Modifiez votre code pour calculer, à chaque pas de temps de la marche, le $\langle D_n^2 \rangle$, et portez-moi ça en graphique versus n ($1 \leq n \leq 1000$). Est-ce-que ça ressemble bien à la Figure équivalente au chapitre 7 des Notes de cours? Discutez des similarités et différences.

4. Calculez maintenant le déplacement moyen dans la direction x (i.e., la moyenne des positions en x de vos marcheurs), $\langle x_n \rangle$, et portez ceci en graphique en fonction de n, comme précédemment. De toute évidence, $\langle x_n \rangle$ a une gueule tout à fait différente de $\sqrt{\langle D_n^2 \rangle}/2$; comment expliquez-vous ceci?

Finalement, sur la base de tous vos résultats, discutez (un paragraphe d'au plus une douzaine de lignes) dans quelles circonstances la marche 2D sur réseau est ou n'est pas une représentation adéquate de la diffusion, au même niveau que le serait la marche aléatoire classique.

9.4 Agrégation

Dans la seconde partie de ce labo, vous vous bâtirez des dendrites fractales par marche aléatoire sur réseau couplée à un processus d'agrégation. L'idée est la même que celle introduite à la §7.5 des notes: un (ou plusieurs) marcheurs se voient assigner un statut "collant-et-immobile"; lorsqu'un autre marcheur arrive sur un noeud situé au voisinage immédiat (± 1 dans chaque direction), il devient collant et immobile son tour.

Imaginez maintenant que vos marcheurs sont maintenant contenus dans une "boite" de dimension 256×256. Donc, chaque fois qu'un marcheur tente de faire un pas à $x < 1$ ou $x > 256$, il doit "rebondir" dans la direction inverse; et tintin dans la direction y. De plus, on devra maintenant être en mesure de vérifier quand un marcheur situé à un noeud (j, k) a ou non de la compagnie sur un site voisin $(j \pm 1, k \pm 1)$. Pour ce faire il sera utile de définir, en plus de deux tableaux contenant les coordonnées discrètes (j, k) de chaque marcheur, un tableau 2D de taille égale à celle du réseau, dont les éléments auront une valeur '0" pour un site vide et "1" pour un site occuppé par un marcheur. Lorsqu'un marcheur se déplace, il faudra donc ajuster ce tableau en conséquence. On assignera aussi une valeur "2" à un marcheur 'collant-et-immobile". Le code C listé à la page suivante effectue ceci. Étudiez bien ce code car vous aurez à le modifier par la suite. Notez en particulier les points suivants:

1. Le canevas global du code inclut encore une fois deux boucles imbriquées, mais cette fois la boucle sur les marcheur est intérieure à la boucle temporelle;

2. L'initialisation consiste à distribuer aléatoirement les marcheurs sur le réseau, et à insérer une valeur "1" à la position correspondante dans le tableau 2D `grille`, pour indiquer que ce site est occuppé, les autres éléments de `grille` ayant été préalablement initialisés à zéro. Remarquez comment des élément de tableaux de type `int`, ici `x[j]` et `y[j]`, sont utilisés pour identifier un élément dans le tableau 2D `grille`, opération tout à fait légale en C mais pas dans tous les langages de programmation.

3. Le calcul débute en assignant le statut "collant" à la rangée de N sites situé à à $y = 1$; vous aurez à modifier ceci dans ce qui suit.

4. Un tableau `statut` assigne un code à chaque marcheur, valant 0 si le marcheur est mobile et 1 si il est "collé". Dans la boucle sur les marcheurs, le calcul du pas et le test "voisin" n'est effectué que pour les marcheurs encore mobiles (`status[j]=0`).

5. La boucle temporelle de votre code tourne jusqu'à ce que tous les marcheurs soient "collés" (i.e., `while( ncolle < M ...)`, ou qu'un maximum préétabli d'itérations (`NITERMAX`) soit atteint.

6. Le test voisin utilise deux tableaux `dx`, `dy` de dimension 1 et longueur 8, identifiant la position relative en x et y respectivement des 8 voisins immédiats (haut-bas-gauche-droite et diagonales correspondantes). Ceci permet d'accéder aisément aux huit voisins du marcheur testé avec un seul `if` dans une boucle sur les 8 voisins. Remarquez comment des opérations arithmétiques sur les éléments de `x`, `dx`, etc, servent directement à identifier les éléments du tableau `grille`.

7. Bien que le réseau soit de taille $N \times N$, le tableau 2D `grille` est de taille $(N+2) \times (N+2)$; les rangées et colonnes 0 et $N - 1$ sont des couches "fantômes" requises pour éviter les débordement de tableaux lors du test "voisin" sans avoir à ajouter un tas d'intructions `if`.

```
#include <stdio.h>
#include <stdlib.h>
#define N 256               /* taille du reseau */
#define M 5000              /* nombre de marcheurs */
#define NITERMAX 250000     /* nombre maximal d'iterations temporelles */
int main(void)
{
/* Declarations ============================================================ */
  int x[M], y[M], statut[M], grille[N+2][N+2] ;
  int ncolle, iter, ifound, i, j, k, jj, xnew, ynew ;
  int dx[8]={-1,0,1,1,1,0,-1,-1} ;              /* Stencil des voisins */
  int dy[8]={-1,-1,-1,0,1,1,1,0} ;
/* Executable ============================================================== */
  for (i=0 ; i<N+2 ; i++) { for (j=0 ; j<N+2 ; j++) { grille[i][j]=0 ; }}
  for (j=0 ; j<M ; j++) {                       /* M marcheurs sur le reseau */
    x[j]     =1+floor(1.*N*rand()/RAND_MAX) ;
    y[j]     =1+floor(1.*N*rand()/RAND_MAX) ;
    statut[j]=0 ; grille[x[j]][y[j]]=1 ;        /* initialisation grille */
  }
  for (j=0 ; j<N ; j++) { grille[j][1]=2 ; }    /* sites collants a y=0 */
  ncolle=0 ; iter=0 ;                           /* variables compteur */
  while ( ncolle < M && iter < NITERMAX ) {     /* Boucle temporelle */
    for (j=0 ; j<M ; j++ ) {                    /* boucle sur les marcheurs */
      if ( statut[j] < 1 ) {                    /* les non-colles bougent */
        if ( 1.*rand()/RAND_MAX < 0.5 ) {       /* on bouge en x */
          xnew= x[j] + 2* (floor( 2.*rand()/RAND_MAX ))-1 ;
          if ( xnew < 1 ) { xnew=1 ; }          /* On ne quitte pas le reseau */
          if ( xnew > N ) { xnew=N ; }
          grille[xnew][y[j]]=1 ;
          grille[x[j]][y[j]]=0 ;
          x[j]=xnew ;                           /* deplacement du marcheur */
        } else {                                /* sinon on bouge en y */
          ynew= y[j] + 2* (floor( 2.*rand()/RAND_MAX ))-1 ;
          if ( ynew < 1 ) { ynew=1 ; }          /* on ne quitte pas le reseau */
          if ( ynew > N ) { ynew=N ; }
          grille[x[j]][ynew]=1 ;
          grille[x[j]][y[j]]=0 ;
          y[j]=ynew ;                           /* deplacement du marcheur */
        }
        ifound=0 ;
        for (k=0 ; k<8 ; k++) {                 /* On regarde les 8 voisins */
          if (grille[x[j]+dx[k]][y[j]+dy[k]] == 2 && ifound == 0) {
            grille[x[j]][y[j]]=2 ;              /* ce marcheur colle...  */
            statut[j]=1 ;
            ncolle+=1 ; ifound=1 ;              /* un colle de plus */
          }
        }
      }
    }
    printf ("iteration, nombre de colles %d %d\n",iter,ncolle) ; iter+=1 ;
  }
}
```

Figure 9.1: Code C de base pour l'agrégation limitée par la diffusion.

8. Afin d'accélérer le calcul, deux marcheurs peuvent ici occupper le même site, ce qui n'est pas habituellement le cas avec la marche aléatoire sur réseau.

Je vous laisse comprendre le pourquoi et le comment de la variable `ifound`... et m'expliquer dans votre rapport pourquoi son rôle est essentiel ici au bon fonctionnement du code! Votre première tâche consiste à ajouter les commandes PLPLOT appropriées pour porter en graphique les positions finales de vos marcheurs, et voir la jolie fractale ainsi produite. Ensuite:

1. Ajoutez au code un tableau 1D comptabilisant le nombre de marcheurs s'étant collés à chaque itération temporelle. Portez ceci en graphique en fonction de l'itération temporelle;

2. Le graphique a l'air un peu fou n'est-ce-pas... c'est parce qu'il y a beaucoup d'itérations où aucun marcheur ne colle; donc, calculez un lissage de largeur ± 50 itérations, un peu comme au labo 8, et retracez le graphique.

3. Répétez le calcul pour $N = 2500$, $N = 10000$ et $N = 20000$ marcheurs, toujours sur un réseau $N \times N = 256 \times 256$. Quelles sont les similarités et différences entre les agrégats ainsi produits? Pouvez-vous intuiter lequel parmi ces quatre agrégat a l'indice fractal le plus élevé? le plus petit?

4. Revenez à $M = 5000$ marcheurs, et changez maintenant la condition initiale du code de manière à ce que le processus d'agrégation ne débute qu'à partir d'un seul site collant situé au centre du réseau; bâtissez ainsi une nouvelle fractale, et calculez de nouveau sa courbe de croissance avec lissage ± 50 itérations; bien que la dynamique d'agrégation-diffusion sous-jacente soit la même, cette courbe a une allure passablement différente de la précédente, n'est-ce-pas? Expliquez pourquoi!

5. Finalement, inventez-vous une condition initiale personnelle (distribution spatiale de sites collants), et voyez quelle fractale résulte du processus d'agrégation. N'hésitez pas à travailler sur un plus grand réseau, ou un réseau non-carré, ou un nombre plus ou moins élevé de marcheurs, etc., si l'envie vous prend. Certains de vos collègues de l'an passé ont obtenu des résultats spectaculaires en utilisant une distribution initiale de forme gaussienne centrée au milieu du réseau, avec le bas du réseau "collant", essayez ça en premier si vous manquez d'inspiration...

6. BONUS I: La plus jolie fractale produite à l'étape précédente sera utilisée l'an prochain en arrière-plan aux pages titres des notes de cours et de labo. Si vous voulez participer au concours, glissez une copie sous ma porte de bureau, avec votre nom écrit au verso.

7. BONUS II: Les vraiment enthousiastes peuvent coder la méthode du décompte des boites et calculer ainsi l'indice fractal de leur agrégat personnel.

Et voilà pour le Labo 9. Il ne reste plus qu'un labo... ah, comme la neige a neigé...

Lectures supplémentaires:

Le chapitre 7 des notes de cours devrait être relu attentivement avant de commencer ce labo.

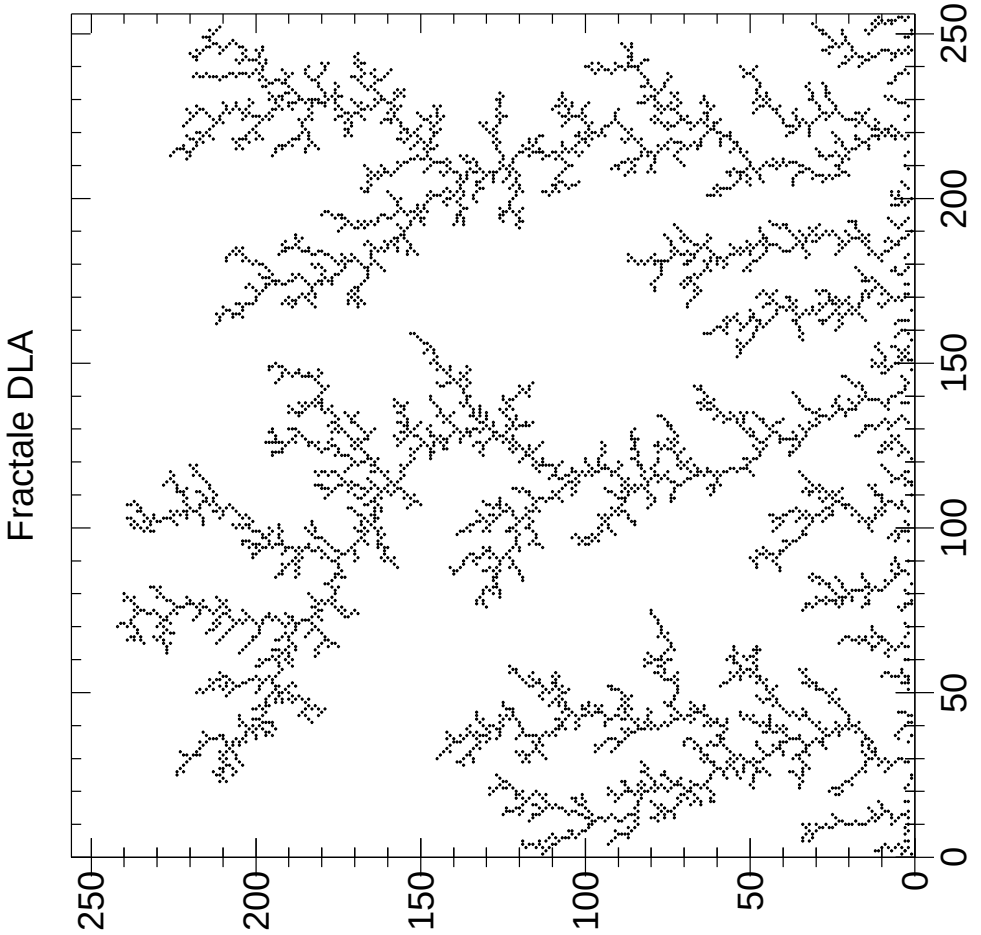

Figure 9.2: Structures dendritiques produites par agrégation sur une surface horizontale "collante" située à $y = 1$. Agrégation de $M = 5000$ marcheurs sur un réseau $N \times N = 256 \times 256$, avec le code C de la Figure 9.1.

Laboratoire 10

Pandémie!

10.1 Objectifs

1. Simuler l'évolution d'un système complexe.

2. Comprendre comment extraire des résultats généraux de simulations dont chaque réalisation individuelle peut produire un résultat fort différent.

10.2 Rapport de Lab

À remettre dans une semaine, avant le début du projet final. Le rapport doit inclure les réponses aux questions posées aux sections 10.4 et 10.5, codes tableaux et graphiques à l'appui, ainsi qu'une discussion critique (mais brève) de vos résultats.

10.3 Une simulation épidémiologique simple

L'idée ici est de combiner la marche aléatoire sur réseau (§8.4) et le modèle Feux-de-Forêt (§9.4). Un nombre M de marcheurs se déplacent sur un réseau 2D de dimensions $N \times N$. La seule différence avec la marche sur réseau décrite à la §8.4 est qu'on permet maintenant à deux marcheurs d'occuper le même site, ce sera essentiel dans ce qui suit. Supposons maintenant qu'un marcheur, choisi aléatoirement parmi les M, commet l'erreur de siéger pendant quelques minutes de trop sur d'un bol de toilette contaminé, et devient infecté par une horrible maladie, genre peste noire. Il n'en meurt pas immédiatement cependant; il survit pendant L itérations, et transmet sa maladie à tout autre marcheur avec qui il vient à occuper le même site sur le réseau. Ces marcheurs nouvellement infectés pourront eux aussi en infecter d'autres durant les L itérations suivant leur infection, et ainsi de suite. Ceci peut donc, sous certaines conditions que vous éluciderez plus loin, conduire à une avalanche d'infection se propageant à travers le réseau. Après L itérations suivant leur infection, les marcheurs malades meurent et cessent immédiatement d'être contagieux.

Le code C de la Figure 10.1 ci-dessous implémente cet "algorithme" de propagation de la maladie. Il sera important de bien comprendre comment fonctionne ce code, car vous aurez à le modifier dans ce qui suit. Notez bien les choses suivantes:

1. La simulation est globalement structurée selon une boucle conditionnelle (`while`) qui tournera jusqu'à ce que le nombre de marcheurs infectés retombe à zéro, ou qu'un maximum prédéfini (`NITERMAX`) ait été atteint. À la sortie de cette boucle temporelle, la valeur de la variable `iter` vous donne donc la durée de l'épidémie.

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#define N 128          /* taille du reseau */
#define M 4000         /* nombre de marcheurs */
#define L 20           /* duree de survie des malades */
#define NITERMAX 5000  /* nombre maximal d'iterations temporelles */
int main(void)
{
/* Declarations ============================================================ */
  int x[M], y[M], infect[M], survie[M] ;
  int ninfect, nmort, iter, j, k, jj, xnew, ynew, d2 ;
/* Executable ============================================================== */
  for (j=0 ; j<M ; j++) {                       /* M marcheurs sur le reseau */
    x[j]     =floor(1.*N*rand()/RAND_MAX) ;
    y[j]     =floor(1.*N*rand()/RAND_MAX) ;
    infect[j]=0 ; survie[j]=0 ;
  }
  jj        =1.*M*rand()/RAND_MAX ;             /* infection d'un marcheur */
  infect[jj]=1 ; survie[jj]=L ; ninfect=1 ; nmort=0 ; iter=0 ;
  while ( ninfect > 0 && iter < NITERMAX ) {    /* Iteration temporelle */
    for (j=0 ; j<M ; j++ ) {                    /* boucle sur les marcheurs */
      if ( infect[j] == 1 ) { survie[j] -=1 ; } /* le temps qui passe... */
      if ( infect[j] == 1 && survie[j] <= 0 ) { /* ...les malades qui meurent */
         infect[j]=2 ; ninfect -=1 ; nmort   +=1 ;
      }
      if ( infect[j] < 2 ) {                    /* seuls les vivants bougent */
        if ( 1.*rand()/RAND_MAX < 0.5 ) {       /* on bouge en x */
          xnew= x[j] + 2* (floor( 2.*rand()/RAND_MAX ))-1 ;
          if ( xnew < 0 )   { xnew=0   ; }      /* On ne quitte pas le reseau */
          if ( xnew > N-1 ) { xnew=N-1 ; }
          x[j]=xnew ;                           /* deplacement du marcheur */
        } else {                                /* sinon on bouge en y */
          ynew= y[j] + 2* (floor( 2.*rand()/RAND_MAX ))-1 ;
          if ( ynew < 0 )   { ynew=0   ; }      /* on ne quitte pas le reseau */
          if ( ynew > N-1 ) { ynew=N-1 ; }
          y[j]=ynew ;                           /* deplacement du marcheur */
        }
      }
      if ( infect[j] == 1 ) {                   /* ce marcheur est malade...  */
        for ( k=0 ; k<M ; k++ ) {               /* ...et peut contaminer les autres */
          if ( k != j && infect[k] == 0 ) {     /* ...qui ne le sont pas deja */
            d2=pow( (x[j]-x[k]),2 )+pow( (y[j]-y[k]),2 ) ;
            if ( d2 == 0 ) {                    /* sur le meme site: contagion */
              infect[k]=1 ; ninfect +=1 ; survie[k]=L ;
            }
          }
        }
      }
    }
    printf ("iter, malades, morts: %d, %d, %d\n",iter,ninfect,nmort) ; iter+=1 ;
  }
}
```

Figure 10.1: Code C de base pour le modèle épidémiologique décrit dans le texte.

2. À chaque itération temporelle, une boucle inconditionnelle `for` sur les M marcheurs déplace chaque marcheur (encore vivant) et mets à jour son statut médical (tableau `infect[M]` décrit plus bas).

3. Quatre tableaux undimensionnels, chacun de longueur M, contiennent l'information relative aux marcheurs: `x[M]`, `y[M]`, `infect[M]`, et `survie[M]`. La position (x, y) sur le réseau du j-ième des `M` marcheur est entreposée dans les éléments `x[j]` et `y[j]`; et un statut épidémiologique correspondant à l'élément `infect[j]`, où une valeur de 0 indique un marcheur en santé, 1 indique un marcheur contaminé (et contagieux!), et 2 un marcheur décédé (et immobile!).

4. L'initialisation consiste à distribuer aléatoirement les marcheurs sur le réseau, et une fois ceci fait en choisir un seul aléatoirement et lui assigner le statut "malade" (`infect[jj]=1`).

5. Seuls les marcheurs vivants (sains ou malades, `infect[j]<2`) se déplacent sur le réseau. La marche aléatoire sur réseau 2D se fait de la manière habituelle, mais ici on a ajouté des tests (les 4 `if`) pour s'assurer que les marcheurs situés aux bords du réseau ne peuvent pas faire un pas en x ou y qui les ferait sortir du réseau.

6. Une fois le marcheur j infecté, l'élément `survie[j]` est initialisé à L, et cette valeur décroit de un à chaque itération temporelle subséquente (le premier `if` dans la boucle principale sur les `M` marcheurs); quand `survie[j]` atteint zéro, le marcheur j est déclaré mort (`infect[j]=2`; second `if` dans la boucle sur les marcheurs).

7. Pour chaque marcheur infecté (dernier `if` dans la boucle temporelle) on boucle sur les autres marcheurs, on calcule leur distance (`d2`), et tout autre marcheur sain (`infect[k]==0`) se trouvant sur le même site (`d2==0`) tombe malade.

8. Les variables-compteurs `ninfect` et `nmort` comptabilisent, à chaque itération temporelle, le nombre de marcheurs infectés et le nombre de morts sur le réseau.

Les Figures 10.2 et 10.3 illustrent différents aspects d'une simulation typique, ici sur réseau de taille 128×128 occupé initialement par $M = 4000$ marcheurs, et une durée de contagion $L = 20$ itérations. Comme la simulation débute avec un seul marcheur contaminé introduit sur le réseau, la contamination est plutôt lente au début (voir Figure 10.2). La maladie se propage sous la forme d'un "front", d'épaisseur finie et initialement quasi-circulaire, qui croit graduellement en rayon, laissant derrière lui une densité réduite de marcheur... et des cadavres. Comme dans le cas du modèle Feu-de-forêt, ce front développe une forme plus complexe, et se fragmente éventuellement en divers foyers d'infection spatialement distincts (voir Fig. 10.3). L'épidémie comme telle se développe en "vagues" successives, mais comme elle "brûle" graduellement son "carburant" elle en vient finalement à s'éteindre, ici après un peu moins de 700 itérations. Dans le cas de cette simulation, près de 60% de la population initiale a été éliminée, ce qui serait tout à fait réaliste dans le contexte des épidémies de peste noire ayant frappé l'Europe au Moyen Âge.

10.4 Comprendre le comportement du modèle

Avant de vous lancer dans l'étude des effets de la vaccination, il vous sera impératif de développer une bonne compréhension du comportement du modèle.

10.4.1 Vérification et validation

En guise de test de validation, votre première tâche consistera à reproduire les résultats des Figure 10.2 et 10.3. Évidemment, vu les multiples éléments stochastiques du modèle, vous ne pouvez pas vous attendre à retrouver *exactement* les mêmes résultats, cependant vous devriez pouvoir obtenir des résultats statistiquement comparables.

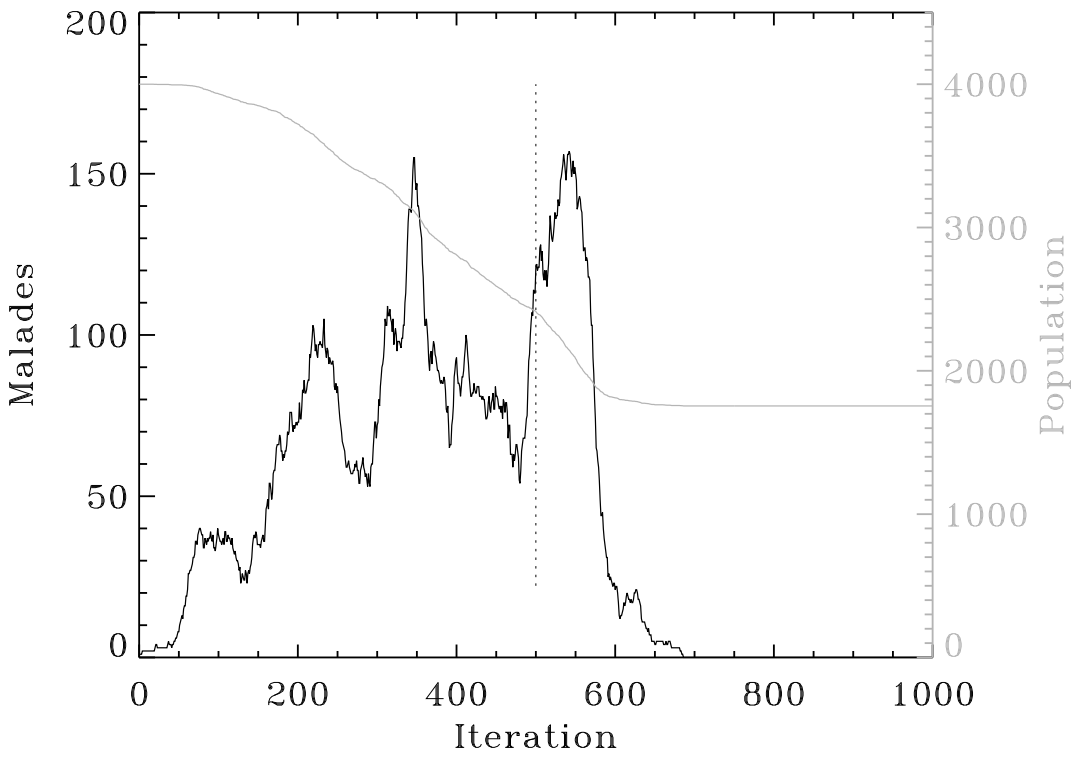

Figure 10.2: Évolution temporelle du nombre de marcheurs sains (trait gris) et malades (trait noir) pour une simulation sur réseau 128×128 avec $M = 4000$ et $L = 20$. Ici, la population initiale a été réduite par près de 60%. Notez les multiples "vagues" dans l'évolution de l'épidémie, une caractéristique souvent observée dans la réalité.

L'idée est donc de répéter la même simulation ($N = 128$, $M = 4000$, $L = 20$) 10 fois, chaque fois avec une initialisation différente du générateur de nombre aléatoire (via la fonction C `srand`, vous vous rappelez...?). Étudiez attentivement vos résultats, et répondez aux questions suivantes:

1. La durée de l'épidémie et le nombre de décès varient-ils beaucoup d'une simulation à l'autre?

2. Le résultat de la Fig. 10.2 est-il en fait représentatif du comportement "moyen" de la simulation pour ces valeurs des paramètres?

3. Les diverses vagues épidémiques visible à la Figure 10.2 semblent avoir une période assez bien définie; pouvez vous imaginer ce qui établit la durée de cette période?

10.4.2 Effet de la densité de population

Pas besoin de réfléchir très longtemps pour réaliser qu'un facteur important dans la propagation de l'épidémie est la densité de population (ρ), soit le nombre M de marcheurs se promenant sur le réseau divisé par le nombre de sites occupables ($N \times N$):

$$\rho = \frac{M}{N^2} \ . \tag{10.1}$$

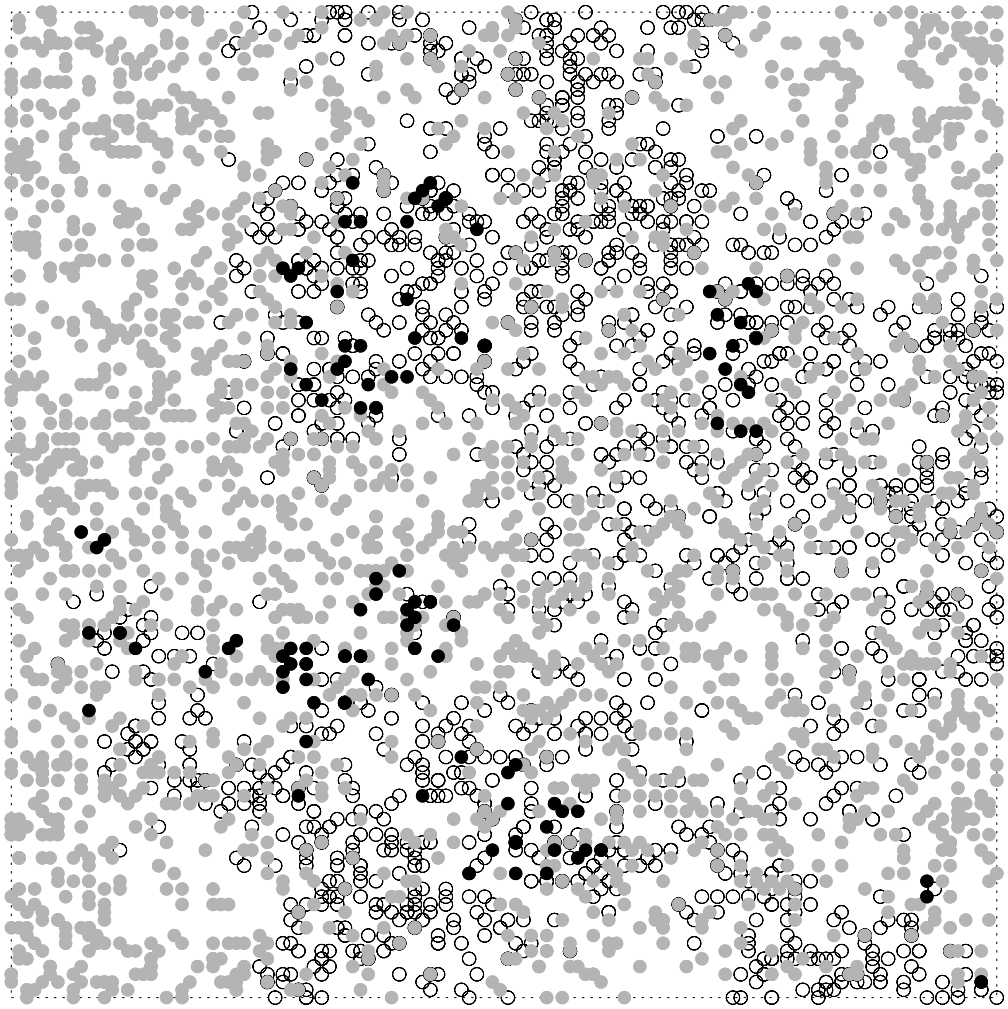

Figure 10.3: Instantané de la distribution des marcheurs dans la phase ascendante de la dernière grande vague épidémique (itération 500, trait vertical pointillé sur la Fig. 10.2). Les marcheurs non-contaminés sont en gris, les marcheurs contaminés en noir, et les cadavres (immobiles) en blanc. Notez la présence de multiples "foyers d'infection" d'étendues spatiales finies, à l'intérieur desquels marcheurs malades et sains cohabitent.

Il s'agit donc de calculer une séquence de simulations avec $\rho = M/N^2$ croissant, de 0.05 à 0.5 en sauts de 0.05. Encore une fois, assurez vous d'évaluer la robustesse de vos résultats en répétant chaque simulation plusieurs fois (genre, 10) avec une initialisation différente du générateur de nombres aléatoires. Définissez le taux de mortalité $0 \leq \mu \leq 1$ comme le nombre de décès divisé par la population initiale. Ensuite,

1. Pour chaque groupe de valeurs de ρ calculez le taux moyen de mortalité ($\langle\mu\rangle$) et la déviation quadratique moyenne (σ) par rapport à ce taux moyen; pour K réalisations de la simulation:

$$\langle\mu\rangle = \frac{1}{K}\sum_{k=1}^{K}\mu_k \ , \qquad \sigma = \left(\frac{1}{K}\sum_{k=1}^{K}(\mu_k - \langle\mu\rangle)^2\right)^{1/2} \ .$$

2. Portez en graphique $\langle\mu\rangle$ versus ρ, et utilisez la valeur correspondante de σ comme barre d'erreur, i.e., ajoutez à chaque point $(\rho, \langle\mu\rangle)$ de votre graphique un trait vertical couvrant l'intervalle $\langle\mu\rangle \pm \sigma$.

3. Produisez un graphique semblable, cette fois pour la durée moyenne ($\langle T\rangle$) des épidémies en fonction de ρ, avec encore une fois une barre d'erreur calculée à partir de la déviation quadratique moyenne.

4. Portez en graphique μ_k versus T_k, en utilisant un symbole de couleur différente pour chaque groupe de K simulations associées à une valeur de ρ. Quelle information pouvez-vous extraire de la distribution des points ainsi produite?

Peut-on dire qu'il existe ici une densité critique au dessus de laquelle l'infection balaye tout le réseau (ou presque) ? Considérant vos résultats, est-il possible de prédire précisément l'évolution d'une épidémie?

10.5 Modéliser la vaccination contre la grippe A (H1N1)

La peste noire, c'était au Moyen Âge. Revenons en décembre 2010. Suite à la gestion désastreusement soviétique de la pandémie de grippe A (H1N1) de la saison 2009-2010, le *Kominkom* à Québec a décidé de PRENDRE DES MESURES! Un budget est alloué à la recherche sur la propagation des épidémies. Une étude statistique préliminaire mandatée en mai 2010 par Notre Bon Gouvernement a compilé les informations suivantes, suite à la grippe 2009-2010:

1. Probabilité de transmission de la grippe par contact personne-à-personne: 0.66 (à chaque contact).

2. Durée de la phase contagieuse: $L = 30$ itérations

3. Citoyen atteint par la grippe: coût net 1 "unité" (U) (jours de travail perdus, consultations médicales, etc.);

4. 5% des citoyens atteints par la grippe développent des complications conduisant à une hospitalisation: coût moyen 50 U par personne concernée.

5. Coût total moyen pour la vaccination: 0.75 U par citoyen (inclut coûts des vaccins, coûts associés à l'organisation de la vaccination de masse, temps double pour le personnel, heures de travail perdues par les citoyens faisant la queue, augmentation de salaire pour les ministres, campagnes publicitaires, consultants externes, préparation des communiqués de presse, etc.)

Hésitant à augmenter les impôts parce qu'il y a de l'élection dans l'air, Notre Bon Gouvernement aimerait bien minimiser les coûts totaux (i.e., $1U$ par malade + $50U$ par malade développant des complications + $0.75U$ par citoyen vacciné) la prochaine fois que la possibilité d'une épidémie se présentera. Votre tâche est donc la suivante: déterminer quelle fraction de la population devrait être vaccinée dans une région de forte densité de population (fraction d'occupation du réseau $\rho = 0.5$), afin de minimiser les coûts totaux de l'opération. On supposera que les malades ne meurent pas de la grippe, donc la question de minimiser les décès n'est plus un enjeu.

Vous devez donc modifier votre code afin d'inclure les effets d'une probabilité finie de transmission de la maladie, et de la vaccination. Pour la probabilité de transmission, il s'agit simplement d'introduire un test probabiliste basé sur une probabilité de transmission p_t (< 1) lorsque deux marcheurs (dont un contagieux) se retrouvent sur le même site du réseau. Pour la vaccination, le plus simple est d'assigner une variable (type `int`) à chaque marcheur, valant 1 si le marcheur est vacciné et 0 sinon. La transmission de la maladie à ce marcheur n'est possible que si cette variable vaut zéro. Vous pouvez supposer que le vaccin confère une immunité complète et absolue aux marcheurs vaccinés. Après qu'un marcheur j ait été infecté pendant L itérations, il guérit et devient immunisé; vous pouvez donc lui associer le même statut médical (immunité= 1) qu'un marcheur vacciné.

Présentez vos résultat sous la forme d'un graphique ou d'un tableau documentant le coût moyen d'une épidémie en fonction du pourcentage de vaccination, accompagné d'une discussion des incertitudes associées à vos résultats. Quel pourcentage-cible de vaccination recommandez-vous?

Lectures supplémentaires:

La section 9.4 des notes de cours devraient être relue attentivement avant de commencer ce labo.

Aussi fortement recommandé sur ce sujet général: le magnifique petit bouquin *Rats, Lice and History*, par Hans Zinsser (1935), ainsi que le chapitre intitulé "Les Fugger de Cologne" dans *l'Oeuvre au Noir*, de Marguerite Yourcenar.

Laboratoire 11

Projet final: embouteillage!

Novembre 2010; le projet de réaménagement de l'échangeur Turcot est finalement dévoilé. A la surprise générale, on avait surestimé la quantité de beignes et de café consomés par les divers consultants et spécialistes en communication et marketing engagés pour pondre le projet. Comme il est hors de question, par principe, de remettre cet excédent aux payeurs de taxes, le Ministère des Transport a décidé d'utiliser le $5000 en question pour engager un(e) stagiaire d'été pour étudier le problème es embouteillages, et vous êtes l'heureux(se) élu(e)... Bravo, et Courage, le KÉBEK compte sur vous!!

11.1 Définition du modèle

Des millions d'autos, "...des millions d'êtres humains qui se battent pour un pouce d'autoroute, sans trop se demander c'qui y est au boutte...", ca vous rappelle (entre autres) les systèmes complexes: un grand nombre de composantes (les autos) n'interagissant qu'avec les quelques composantes voisines (l'auto devant et celle derrière) selon des règles simples (ralentir si l'auto d'en avant ralentit, accélérer si elle accèlère, etc.). Vous vous définissez donc le système idéalisé suivant:

1. Les voitures se déplacent sur une route à une seule voie, à sens unique; donc, pas de dépassement.
2. Les positions et vitesse de la k-ième voiture au temps t_n sont dénotée par x_k^n et v_k^n
3. à chaque pas de temps (n), chaque chauffeur (k) ajuste sa vitesse en fonction de la distance le séparant de l'auto le précédant:
$$\delta = x_{k+1}^n - x_k^n$$
4. Si $\delta < 5$, on ralentit: $v_k^n \rightarrow v_k^n - 3$
5. Si $\delta > 5$, on accéllère: $v_k^n \rightarrow v_k^n + 1$
6. Vitesse minimale: zéro (on ne recule PAS dans un sens unique, n'en déplaise aux chauffeurs de taxi...);
7. Vitesse maximale: 10 (sinon la SQ s'en mêle vite, avec son enthousiasme habituel pour la chose)
8. Les voitures se déplacent suivant la prescription habituelle reliant le déplacement à la vitesse (ici considérée constante durant une itération temporelle):
$$x_k^{n+1} = x_k^n + v_k^n \times \Delta t \ .$$

Dans tout ce qui suit, on posera $\Delta t = 1$ sans aucune perte de généralité. Le code C de la Figure 11.1 ci-dessous implémente cet "algorithme" de déplacement du trafic, avec une importante addition discutée plus bas. Il sera important de bien comprendre comment fonctionne ce code, car vous aurez à le modifier au cours de ce projet. Notez bien, et comprenez, les aspects suivants:

1. La simulation est globalement structurée selon des boucles imbriquées, la boucle extérieure étant l'itération temporelle et une séquence de trois boucles inconditionnelles intérieures sur les `N` voitures;

2. L'initialisation des positions se fait selon des *incréments* de taille aléatoire mais toujours positifs; ici $3 \leq x_{k+1} - x_k \leq 17$, pour un intervalle moyen de 10 unités; cette façon de faire assure que $x_1 < x_2 < x_3 < x_4 < ... < x_{\rm N}$;

3. Le changements de la vitesse des voitures est tout d'abord calculé pour toutes les voitures, et ensuite une seconde boucle modifie le tableau des positions de manière synchrone, en une étape distincte du calcul des changements de vitesse.

4. Une fonction incluant un test assure que la vitesse ne peut chuter sous zéro;

5. Une fonction incluant un test assure que la vitesse ne peut dépasser 10;

6. Un test assure que chaque voiture ne peut pas s'approcher à moins d'une unité de la voiture la précédant;

7. La voiture de tête, qui n'a pas de voiture la précédant, ajuste sa vitesse de manière particulière, en fonction de la distance de la voiture la suivant;

8. Et voici la clef de la simulation: occasionnellement, sans raison particulière autre que l'arrivée d'un texto, le changement d'un CD, le cellulaire qui sonne, un écureuil traversant la chaussée, ou même vraiment pour rien, un chauffeur aléatoire freine... Ici cet ajustement est effectué après l'ajustement déterministe contrôlé par les distances inter-voitures.

Les Figures 11.2 et 11.3 illustrent différents aspects d'une simulation typique, ici pour un groupe de 300 voitures initialement réparties aléatoirement avec une distance inter-voiture moyenne de 10; il s'agit en fait des valeurs de paramètres utilisées dans le code de la Figure 11.1. La figure 11.2 montre les trajectoires de chacune des 300 voitures, i.e., 300 tracés de x versus t. Une déviation horizontale des tracés, impliquant que x demeure constant quand t augmente, indique une vitesse zéro, soit un embouteillage! La figure 11.3 pousse trois fois plus loin dans le temps, et se borne à indiquer les positions où les voitures sont au repos ou presque (vitesse ≤ 1). Remarquez que même une fois le système stabilisé, des embouteillages pouvant impliquer un nombre très variable de voitures se développent de manière en toute apparence erratique. La plupart de ces embouteillages sont causés par le freinage aléatoire d'une voiture, mais ce qui est remarquable est qu'un tel freinage individuel puisse produire (parfois) un embouteillage impliquant une substantielle fraction du groupe de voitures en mouvement. L'encadré illustre la trajectoire d'une voiture particulière s'étant tout juste dépêtrée d'un embouteillage majeur ayant affecté tout le réseau, et rejoignant, puis quittant, deux embouteillages secondaires en tentant d'accélérer de nouveau. Plusieurs caractéristiques de cette simulation se doivent être notées déjà à ce stade:

1. Le trafic se met en branle en deux phases plus ou moins distinctes: un transient initial farci d'embouteillages majeurs, suivi d'une seconde phase où toutes les voitures se déplacent à une vitesse moyenne plus ou moins constante. Ici la transition semble se produire à $t \simeq 1300$ (cf. Fig. 11.3), mais on verra plus loin que le système atteint un état véritablement statistiquement stationnaire passablement plus tard, soit vers $t \simeq 2000$.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define NT    2000                              /* Nombre de pas de temps */
#define N     300                               /* Nombre d'autos */
#define PBOZO 0.1                               /* Probabilite freinage aleatoire */
int main(void) {
/* Declarations ============================================================ */
  int tmin(int, int) ;
  int tmax(int, int) ;
  int x[N], v[N], del, k, iter ;
/* Executable ============================================================== */
  x[0]=1 ; v[0]=0 ;
  for (k=1 ; k<N ; k++) {                       /* repartitions initiale des autos */
    x[k]=x[k-1]+floor(3.+14.*rand()/RAND_MAX) ;
    v[k]=0 ;                                    /* Vitesse initiale nulle */
  }
  for ( iter=0 ; iter<NT ; iter++ ) {           /* boucle temporelle */
    for ( k=0 ; k<N-1 ; k++ ) {                 /* boucle 1 sur voitures: vitesse */
      del = x[k+1]-x[k] ;                       /* calcul distance */
      if ( del < 5 ) { v[k]=tmax(0 ,v[k]-3) ; } /* trop pres: on ralentit... */
      if ( del > 5 ) { v[k]=tmin(10,v[k]+1) ; } /* assez loin: on accelere... */
    }
    if (x[N-1]-x[N-2] <= 10 ) {                 /* Cas special: la voiture de tete */
      v[N-1]=tmin(10,v[N-1]+1) ; }
    for ( k=0 ; k<N ; k++ ) {                   /* boucle 2 sur voitures: freinage */
      if ( 1.*rand()/RAND_MAX <= PBOZO ) {      /* un bozo ralentit parfois pour rien */
        v[k]=tmax(0,v[k]-3) ; }
    }
    for ( k=0 ; k<N-1 ; k++ ) {                 /* boucle 3 sur voitures: on roule */
      x[k]=tmin(x[k]+v[k],x[k+1]-1) ;           /* Deplacement (borne)  */
    }
    x[N-1]=x[N-1]+v[N-1] ;                     /* Cas special: la voiture de tete */
  }                                             /* fin boucle temporelle */
}
int tmin ( int a, int b )                       /* Trouve le plus petit de a,b */
{
  int ff ;
  if ( a < b ) { ff = a ; }
  else         { ff = b ; }
  return  ff ;
}
int tmax ( int a, int b )                       /* Trouve le plus grand de a,b */
{
  int ff ;
  if ( a < b ) { ff = b ; }
  else         { ff = a ; }
  return  ff ;
}
```

Figure 11.1: Code C de base pour le modèle du trafic automobile décrit dans le texte.

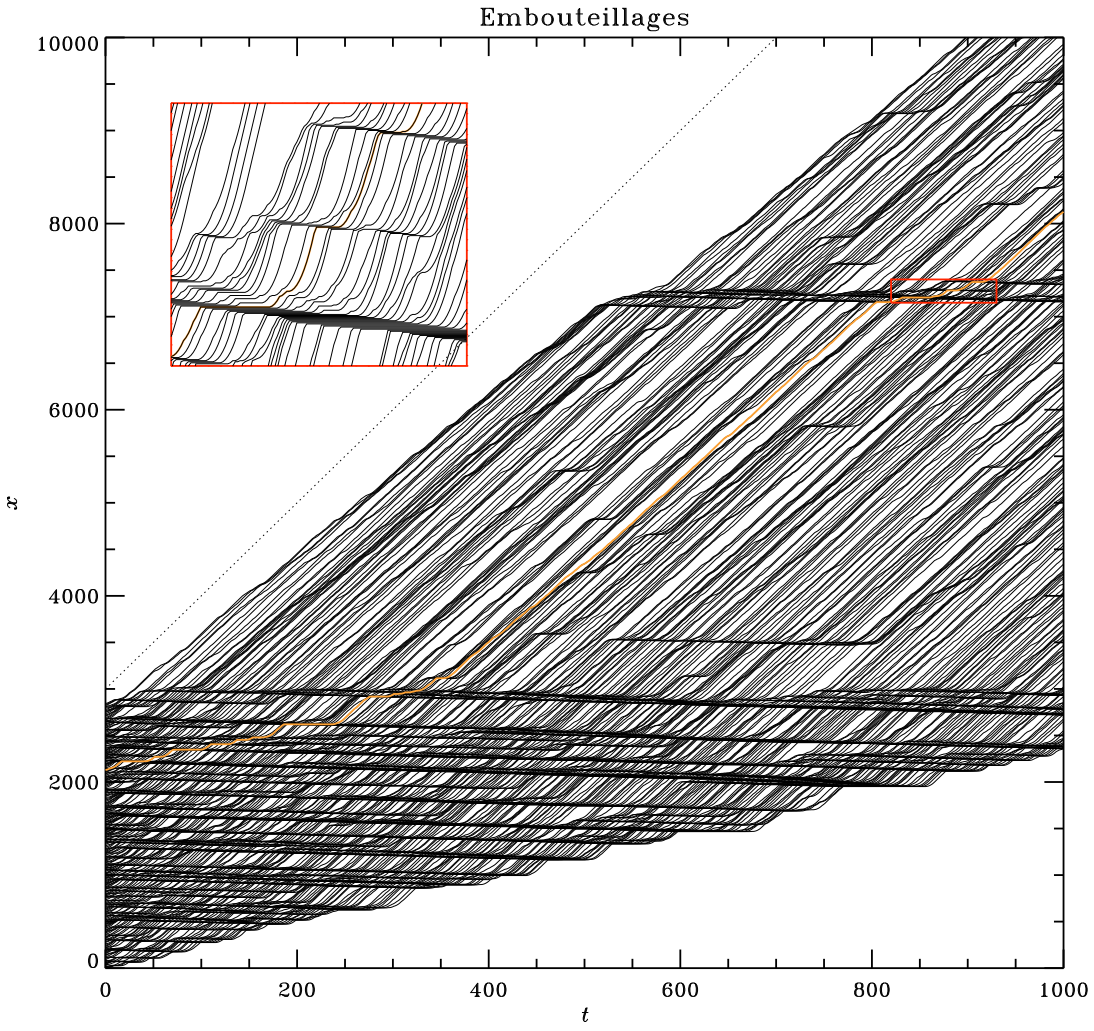

Figure 11.2: Évolution de la position des voitures (axe vertical) en fonction du temps (axe horizontal). On note deux phases distinctes, la première durant laquelle la condition initiale relaxe, par une série d'embouteillages majeurs, vers un état où la vitesse de toutes les voitures est approximativement constante, mais où des embouteillages peuvent néanmoins se produire. Le trait orange indique la trajectoire d'une voiture spécifique, située au trois quart du peloton. Les lignes pointillées indiquent la pente associée à une voiture se déplaçant à la vitesse maximale $v = 10$. L'encadré montre un zoom sur un embouteillage; on y constate que les voitures ne se croisent jamais (comme il se doit puisque les dépassements sont interdits!). Cette simulation, impliquant 300 voitures, a été effectuée avec la condition initiale et les valeurs de paramètres tels que spécifiés dans le code de la Figure 11.1.

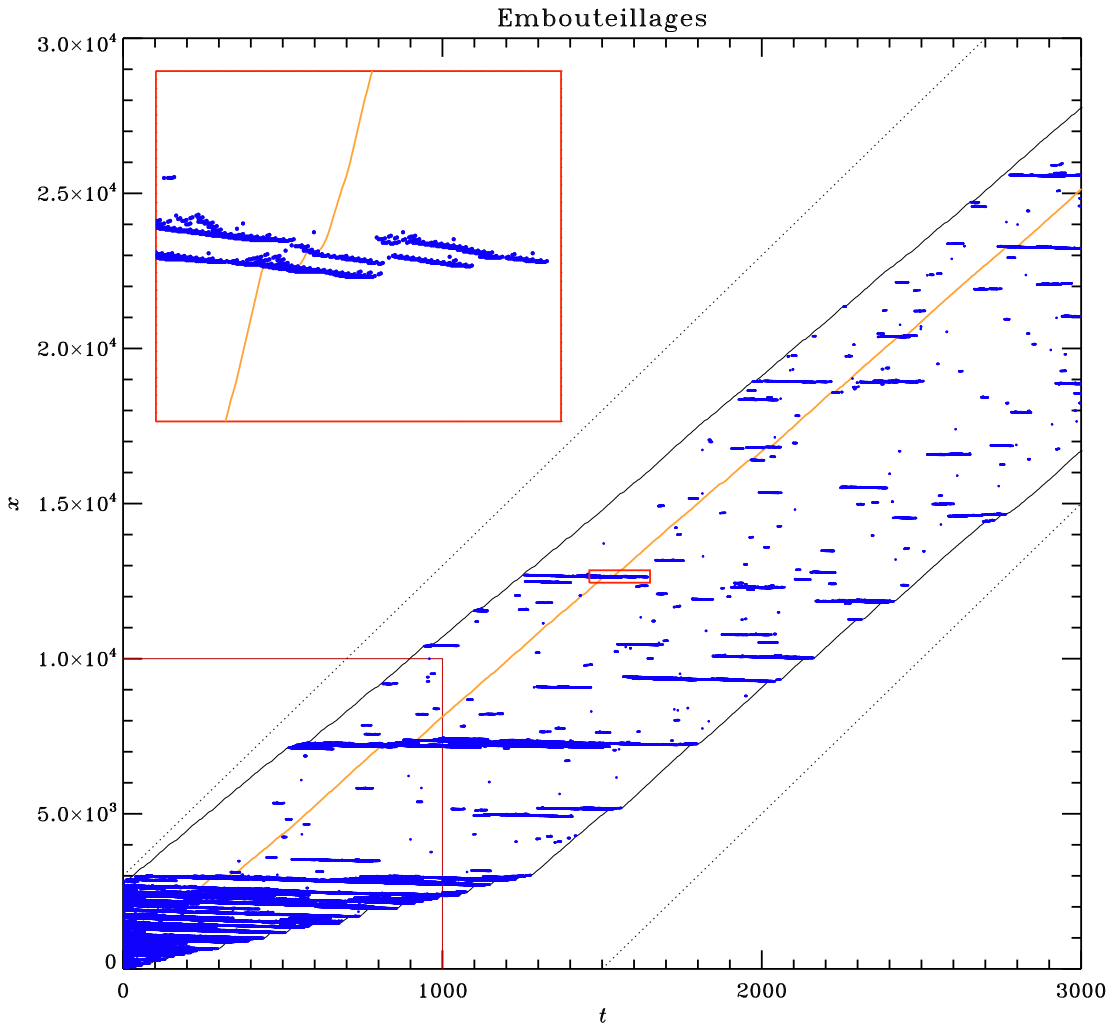

Figure 11.3: La même simulation que sur la Figure 11.2, mais cette fois couvrant un plus grand intervalle temporel et montrant la distribution spatiotemporelle des embouteillages. Chaque point bleu correspond à une voiture au repos ou presque ($v \leq 1$). Le carré inférieur gauche définit l'intervalle couvert sur la Figure 11.2, et les lignes pointillées indiquent encore une fois la pente d'une trajectoire à vitesse maximale $v = 10$. Les trajectoires des première et dernière voitures sont tracée en noir, et la voiture-test en orange.

2. Même durant la seconde phase, de nombreux embouteillages se développent et disparaissent, pouvant n'impliquer que quelques voitures, ou une grande fraction du peloton (e.g., l'embouteillage débutant à $(x,t) \simeq (7000, 500)$ sur la Fig. 11.2).

3. Durant la seconde phase, une voiture quelconque tend soit à se déplacer presque à vitesse maximale, soit à être coincée dans un embouteillage (voir trajectoire en orange).

4. L'étendue temporelle de l'embouteillage est substantiellement plus longue que le temps passé par une voiture s'y empêtrant (trajectoire orange); les voitures à la tête de l'embouteillage peuvent accélérer et s'en dépêtrer graduellement, une voiture à la fois, tandis que les voitures atteignant la queue de l'embouteillage s'y empilent. C'est pourquoi l'embouteillage, une fois déclenché, "recule" en x en fonction du temps (voie encadré sur Fig. 11.3).

Deux quantité intéressantes à suivre, dans ce qui suit, sont la vitesse moyenne des voitures:

$$\langle v \rangle = \frac{1}{N} \sum_{k=1}^{N} v_k \ , \tag{11.1}$$

et la distance moyenne entre voitures:

$$\langle \delta \rangle = \frac{1}{N-1} \sum_{k=1}^{N-1} (x_{k+1} - x_k) = \frac{x_N - x_1}{N-1} \tag{11.2}$$

(je vous laisse le soin de démontrer la seconde égalité dans cette dernière expression). La densité moyenne de voitures (ρ; nombre de voitures par unité de distance) est simplement l'inverse de cette expression:

$$\rho = \frac{N-1}{x_N - x_1} \tag{11.3}$$

Connaissant ces deux quantités, le *flux* (Φ) de voitures, soit le nombre moyen de voiture traversant une position x^* quelconque par unité de temps, se calcule facilement:

$$\Phi = \rho \times \langle v \rangle \ ; \tag{11.4}$$

C'est une quantité que vous aurez à calculer —et maximiser!— dans ce qui suit. La Figure 11.4 montre la variation dans le temps de $\langle v \rangle$ et ρ, pour la simulation de la Figure 11.2. On y remarque que les valeurs numériques de ces deux quantités varient rapidement jusqu'à $t \sim 1300$, ce qui correspond au changement marqué dans la structure des embouteillages visibles sur la Fig. 11.2, cependant la densité de voitures —et donc aussi le flux— ne se stabilise vraiment que vers $t \simeq 2000$.

11.2 Vérification et validation

En guise de test de validation, votre première tâche consistera à reproduire les résultats des Figure 11.2 et 11.4. Évidemment, vu les multiples éléments stochastiques du modèle, vous ne pouvez pas vous attendre à retrouver *exactement* les mêmes résultats, cependant vous devriez pouvoir obtenir des résultats statistiquement comparables. Il s'agit donc ici de

1. modifier votre code afin de calculer et conserver dans des tableaux de dimension appropriées les vitesse moyenne et densité à chaque pas de temps;

2. Insérez des commandes PLPLOT appropriées afin de reproduire les Figures 11.2 (sans l'encadré) et 11.4 (sans nécessairement les axes verticaux multiples)

3. La simulation des Figs. 11.2, 11.3 et 11.4 est-elle représentative du comportement du modèle?

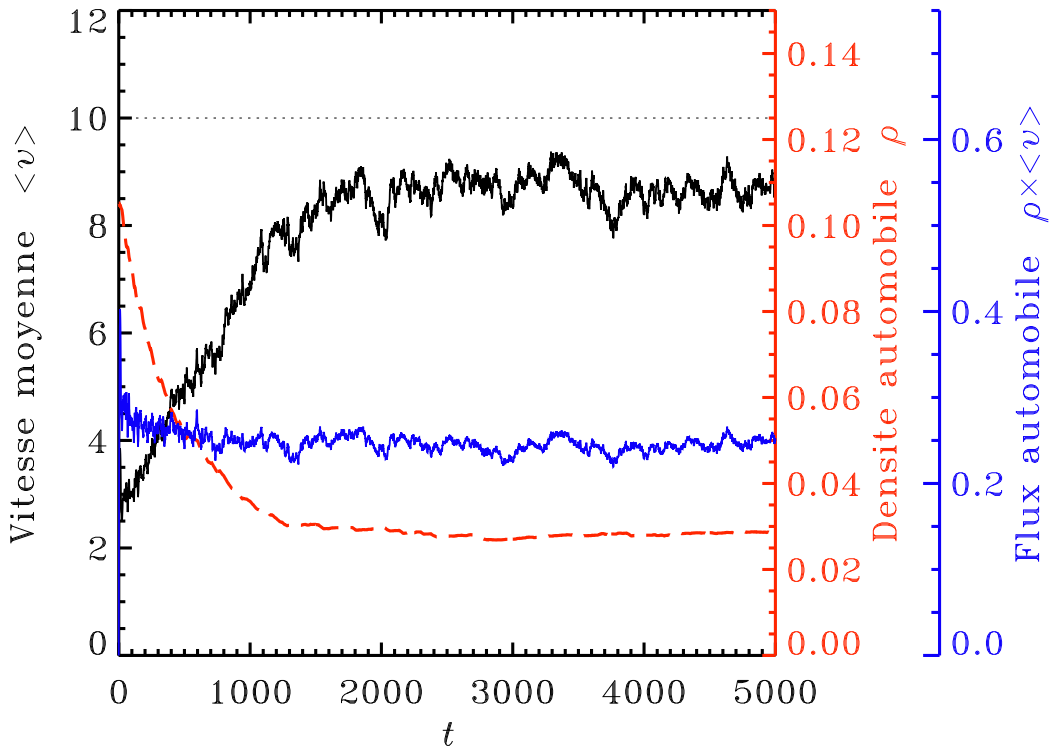

Figure 11.4: Variation temporelle de la vitesse moyenne $\langle v \rangle$, de la densité de voitures ρ, et du flux Φ, pour la simulation des Figures 11.2 et 11.3, maintenant poussée jusqu'à 5000 pas de temps.

11.3 Comprendre le comportement du modèle

Avant de vous lancer dans l'ingénierie du trafic sur l'autoroute 15 (voir plus loin...) il vous sera impératif de développer une bonne compréhension du comportement du modèle.

11.3.1 Dépendance sur les conditions initiales

La série d'embouteillages monstres se développant en début de simulation suggère que la condition initiale choisie ici n'est peut-être pas la meilleure, du point de vue de la fluidité du trafic. Jusqu'à quel point cette condition initiale influence-t-elle l'évolution de la simulation? Est-elle "oubliée" rapidement ou lentement? Examinons ces questions.

1. Exécutez la simulation avec quelques germes différents pour le générateur de nombre aléatoire.

2. Répétez la simulation cette fois avec une condition initiale où les voitures sont initialement réparties uniformément en x, avec une distance de 10 unités entre chaque paire de voitures.

3. Répétez ces deux séries d'expériences avec $N = 200$ et $N = 500$ voitures, toujours avec un espacement moyen de 10 unités, alétoire ou constant.

Maintenant, examinez bien vos résultats et répondez aux questions suivantes:

1. Les vitesse moyenne et densité finales dans l'état statistiquement stationnaire dépendent-elles de la condition initiale? du nombre de voitures?

2. Le temps requis pour atteindre l'état statistiquement stationnaire dépend-t-il de la condition initiale? du nombre de voitures?

3. Jusqu'à quel point la complexité de l'écoulement du trafic reflète-t-elle celle de la condition initiale?

Réfléchissez à la meilleure manière de présenter vos résultats de manière claire et concise (graphique avec plusieurs courbes, tableaux, etc.)

11.3.2 La dynamique du trafic à basse densité

Une approche évidente à l'élimination des embouteillages est d'espacer les voitures suffisamment l'une de l'autre de manière à ce que si l'une d'elles freine aléatoirement, ceci ne causera pas un freinage de la voiture la suivant avant que la voiture ayant freiné n'ait pu accélérer de nouveau à sa vitesse de croisière. Examinez cette possibilité en effectuant une série de simulations où les voitures sont uniformément espacées initialement, avec des distances inter-voitures allant de 5 à 100, par sauts de 5 jusqu'à 30 en 10 par la suite. Assurez-vous bien de pousser vos simulations suffisamment loin dans le temps pour atteindre un état statistiquement stationnaire. Ensuite, attaquez-vous aux questions suivantes:

1. Existe-t-il une "densité seuil" au dessous de laquelle les embouteillages n'apparaissent jamais? Si oui, quelle est sa valeur?

2. Auriez-vous pu prédire un tel seuil à partir des règles d'ajustement de vitesse caractérisant vos simulations?

3. Une fois l'état statistiquement stationnaire atteint, comment varie le flux de voitures en fonction de la densité initiale?

11.3.3 Les embouteillages sont-ils périodiques?

Réfléchissez un peu; il existe plusieurs temps caractéristiques cachés dans votre système; par exemple, comme "l'information" relative à un embouteillage ne voyage (vers l'arrière) qu'une voiture à la fois par pas de temps, selon une séquence de freinage successifs, on pourrait imaginer qu'il puisse exister dans le système une périodicité de l'ordre de $T \sim N$. Un examen visuel de la Figure 11.2 suggère également que certaines périodicité spatiales puissent également être présentes, qui devraient se traduire en périodicités temporelles également via l'algorithme de déplacement des voitures. Ces périodicités, si elles existent vraiment, seraient extrêmement intéressantes à identifier dans le but de prédire les embouteillages; donc vous devez regarder ça de plus près.

Vous allez travailler à partir de la séquence temporelle de la vitesse moyenne (cf. trait noir sur la Figure 11.4), pour la simulation de référence présentée aux Figs. 11.2 à 11.4 ($N = 300$, répartition initiale aléatoire avec distance inter-voitures moyenne de 10). Faites bien attention à ne pas utiliser la partie de la séquence correspondant au transient initial, et de vous en tenir à la partie subséquente où les valeurs des densité et vitesse moyenne se sont vraiment stabilisées ($t \gtrsim 2000$ sur la Fig. 11.4). Il donc serait judicieux de pousser votre simulation au moins jusqu'à $t = 5000$ ou même 10000, de manière à avoir une séquence temporelle de $\langle v \rangle$ suffisamment longue.

Par transformée de Fourier (comme au Labo 8), calculez le spectre de la séquence temporelle de $\langle v \rangle$. N'oubliez pas de soustraire la valeur moyenne! Ensuite:

1. Pouvez-vous distinguer des pics bien définis dans le spectre?

2. Comment pourriez vous caractériser mathématiquement la forme du spectre?

3. Qu'en concluez vous quant aux périodicités possibles des embouteillages?

11.3.4 Les embouteillages ont-ils une échelle caractéristique?

Vous aurez déjà saisi que la formation d'un embouteillage représente une forme d'avalanche de freinages successifs. Si l'analogie tient, on pourrait s'attendre à ce que ces "avalanches" soient caractérisées par une invariance d'échelle. La *taille* d'un embouteillage devrait normalement être définie comme le nombre de voitures bloquée ($v_k^n \leq 1$) sommé sur la durée d'arrêt de chaque voiture impliquée. Autrement dit, le nombre de points bleus dans chaque "amas" distinct sur la Figure 11.3.

Une alternativer plus facile consiste à choisir quelques voitures-test (genre, une dizaine) suffisamment séparées l'une de l'autre, et de comptabiliser les *durées* des phases où chaque voiture est arrêtée ou presque (dans le sens $v_k^n \leq 1$). On sait déjà (voir Figures 11.2 et 11.3) que ces phases sont plus courtes que la durée de l'embouteillage dans son ensemble, mais elles devraient tout de même montrer les même distributions statistiques. Donc, travaillant encore une fois avec la simulation de référence comme à la section précédent, extrayez les séquences temporelles des vitesses de vos voitures test, et pour chacune batissez une liste de durées des phases où $v_k^n \leq 1$. Combinez ensuite les listes associées à toutes vos voitures-test et calculez un histogramme de la distribution des durées de ces phases d'arrêt. Quelle fonction mathématique décrit le mieux cette distribution? Vos embouteillages montrent-ils une invariance d'échelle?[1].

Notez bien que vous devez exclure du calcul les embouteillages se produisant durant le transient associée à la relaxation de votre condition initiale, et ne conserver que les embouteillages se développant durant le régime statistiquement stationnaire de la simulation (genre $t \gtrsim 2000$). Assurez vous de rouler votre simulation suffisamment longtemps pour avoir un nombre assez élevé d'embouteillages

11.3.5 Est-ce un état SOC?

On a vu à la §9.3 des notes de cours les cinq éléments essentiels aux systèmes en autorégulation critique. Ces cinq éléments sont-ils tous présents dans votre simulation du trafic? Discutez ces équivalences (si elles existent), et spéculez (intelligemment et brièvement) sur la possible nature SOC de la circulation automobile.

11.4 Le défi du Ministère des Transport

Vous avez pu constater qu'indépendamment de la condition initiale, votre trafic converge toujours vers une "solution" caractérisée par des vitesse et densité moyennes bien définies (cf. Fig. 11.4). La question est, pourrait-on faire mieux en répartissant de manière plus "intelligente" les voitures, et/ou en contrôlant activement la vitesse moyenne du trafic?

Notre Bon Gouvernement vous donne carte blanche pour faire un test grandeur nature sur l'autoroute 15, de Rosemère à l'intersection avec la 40, un beau lundi matin. Une escouade de la SQ stoppe tout le trafic à l'échangeur de la 640, et vous aide à le répartir dans la voie du centre (celle de gauche étant réservée aux autobus, celle de droite aux bixis). Une voiture-patrouille flambant neuve de la SQ, tous gyrophares allumés, est en tête de convoi, et assure un démarrage graduel ainsi que le contrôle de la situation (en principe). Vous voyez le portrait...

L'idée est évidemment de maximiser le flux du trafic. Vous pouvez contrôler les aspects suivants de la simulation:

[1] Les plus enthousiastes pourront répéter l'exercice en mesurant la taille totale des embouteillages. Voici une suggestion pour vous lancer: commencez par définir un tableau bidimensionnel de type `int`, genre `mat[N][NT]`, la première dimension correspondant à la numérotation des voitures et la seconde à l'itération temporelle. Il s'agit simplement d'assigner un code numérique, "1" pour une voiture bloquée ($v_k^n \leq 1$) et "0" sinon. Ceci peut se faire à mesure que la simulation avance dans le temps. Comme vos voitures ne se dépassent pas, ce tableau préserve la structure spatiotemporelle des embouteillages, dans les sens que chaque groupe de "1" contigus dans `mat`, séparé d'autres groupes semblables par au moins un "0" dans les directions spatiales (première dimension) et temporelle (seconde dimension), corresponds à un embouteillage distinct. Une fois l'itération temporelle terminée, il s'agira ensuite décrire un bout de code C qui balaye `mat`, et calcule la taille comme étant égale au nombre de "1" dans chaque groupe. [FACULTATIF!!]

1. La répartition initiale des voitures (distance constante ou aléatoire, densité, etc.)
2. La vitesse maximale de la voiture de tête, sujet évidemment à $v_{\max} \leq 10$.

Vous n'avez aucun contrôle, cependant, sur le "comportement" des autres chauffeurs (fréquence et amplitude du freinage aléatoire, taux de freinage ou d'accélération, etc). N'oubliez pas: ce qui doit être maximisé, c'est le *flux* de voiture, pas seulement leur vitesse moyenne, car le but est de faire rentrer toutes les voitures à Montréal le plus rapidement possible.

Sur la base de ces essais, et de votre compréhension générale du comportement du modèle, que recommandereriez-vous au Ministère?

11.5 Votre rapport de synthèse au Ministère

Si Notre Bon Gouvernement vous donne pour $5000 en beignes et café comme salaire pour un stage d'été, il exigera certainement un RAPPORT. Ce rapport devrait inclure au moins les items suivants:

1. Un résumé "exécutif", compréhensible par un(e) politicien(ne) qui n'a pas suivi PHY-1234. Évitez le jargon technique, les mots de plus de quatre syllabes, etc. Une demi-page maximum;
2. Un résumé "technique", pour vos collègues ayant suivi PHY-1234. Une demi-page maximum;
3. Une brève description du modèle utilisé et du protocole de simulation; une demi-page maximum;
4. Quelques résultats représentatifs illustrant le comportement du modèle, sous forme de tableaux, Figures, avec légendes et un peu de texte explicatif indiquant au lecteur les choses importantes à noter sur ces tableaux et Figures; notez bien que les questions spécifiques posées aux sections 11.3 et 11.4 ne sont ici que des points de départ à votre discussion.
5. Un tableau compilant et résumant vos résultats de simulations de la §11.4 sur l'ingénierie du trafic, accompagné d'une discussion du protocole de simulation utilisé pour obtenir ces résultats. Il sera particulièrement important de présenter de façon claire et judicieuse vos résultats au niveau du flux de voitures.
6. Des estimés d'erreur et une discussion du niveau d'incertitude associé à ces résultats. Une brève discussion de la manière dont ces estimés d'erreurs ont été obtenus;
7. Une synthèse et discussion des caractéristiques des embouteillages résiduels caractérisant votre solution optimale au niveau du flux de voitures;
8. Une annexe incluant un listing de votre code C, bien indenté et bien documenté (commentaires), en particulier au niveau des diverses modifications que vous avez apportées au code original de la Figure 11.1. Ce code devrait pouvoir être lisible et compréhensible par un joyeux stagiaire héritant de la phase II de l'étude l'an prochain;
9. En guise de conclusion, une courte liste de recommandations-clef relatives à l'ingénierie du trafic par l'État, telles que justifiées par vos résultats de simulation.

Le tout ne devrait pas dépasser $\sim$ 25 pages, Figures et tableaux inclus. Une synthèse intelligente, accompagnée de quelques Figures et/ou tableaux présentant des résultats représentatifs, est souvent plus utile qu'un catalogage de résultats de simulations couvrant systématiquement l'espace des paramètres. Le Ministère vous demande ici un **rapport de synthèse** supportant quelques recommandations clefs. Jouez le jeu! et Joyeux Noël!

Laboratoire 12

Les avalanches

12.1 Objectifs

1. Découvrir le comportement et les propriétés des systèmes en autorégulation critique.

2. Approfondir vos techniques d'analyse de séquences temporelles

12.2 Rapport de Lab

À remettre dans une semaine. Le rapport doit inclure les réponses aux questions posées aux sections 10.4 à 10.6 inclusivement, codes tableaux et graphiques à l'appui, ainsi qu'une discussion critique (mais brève) de vos résultats.

12.3 Le modèle Tas-de-Sable

Vous travaillerez ici avec le modèle Tas-de-Sable tel qu'il est décrit à la §9.2 des notes de coures, qui ne seront pas reproduites en détail ici. La Figure 12.1 liste le code C qui servira de point de départ à ce laboratoire. Ce code effectue une simulation de la croissance d'un tas de sable, à mesure que des petites quantités de sable sont déposées au hasard sur le tas. La quantité de sable à l'itération temporelle n à la position j sur le réseau est emmegasinée dans un tableau `sable[N]`, où `N` est le nombre de noeuds dans le réseau unidimensionnel. Si la pente locale associée à deux noeuds contigus dépasse un seuil critique Z_c, du sable est redistribué entre ces deux noeuds afin de ramener la pente sous sa valeur critique. Voir la section 9.2 des Notes de cours pour plus de détail sur tout ça.

Le code est de la Figure 12.1 est configuré pour débuter avec une condition initiale `sable=0` partout. Chaque itération temporelle implique deux étapes distinctes: (1) balayer le réseau tout en vérifiant la stablité de chaque paire de noeuds contigus et, le cas échéant, accumuler dans une variable temporaire (le tableau `move`) la quantité de sable à échanger entre chaque paire de noeuds instables afin de stabiliser le système; (2), mettre à jour simultanément tous les noeuds du réseau avant de passer à l'itération temporelle suivante. L'ajout de sable n'est effectué que si le système est stable partout. Un noeud est choisi aléatoirement, et une petite quantité de sable y est ajoutée, de grandeur également choisie aléatoirement.

La simulation que vous exécuterez dans ce qui suit diffère de celle décrite dans les Notes de cours en un seul point: la condition limite `sable[N-1]=0` n'est imposée qu'au dernier noeud du réseau, mais pas au premier. Ceci correspond à un tas de sable s'élevant contre un mur. La Figure 12.2 montre la croissance du tas pour une simulation typique. Comparez ceci attentivement avec la Figure 9.2 des Notes, et comprenez bien pourquoi et comment le changement de conditions limites cause la différence observée.

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#define N 101        /* taille du reseau */
#define PENTECRIT 5. /* pente critique */
#define E       0.15 /* amplitude de forcage */
int main(void)
{
/* Declarations ========================================================= */
  float sable[N], move[N] ;
  float masse, dmasse, av, pente ;
  int j, jj, iter, niter=100000 ;
/* Executable =========================================================== */
  for ( j=0 ; j<N ; j++) { sable[j]=0. ; }       /* Condition initiale */

  for (iter=0 ; iter<niter ; iter++) {         /* iteration temporelle */

     for ( j=0 ; j<N ; j++) { move[j]=0. ; }
     dmasse=0 ;

     for (j=0 ; j<N-1 ; j++ ) {
        pente=fabs(sable[j+1]-sable[j]) ;    /* pente associee a j,j+1 */
        if ( pente >= PENTECRIT ) {     /* la paire j,j+1 est instable */
            av=0.5*(sable[j+1]+sable[j]) ;
            move[j]  =move[j]  +0.5*(av-sable[j]) ;
            move[j+1]=move[j+1]+0.5*(av-sable[j+1]) ;
            dmasse+=pente/2. ;           /* cumul de la masse deplacee */
        }
     }

     if ( dmasse > 0. ) {  /* Il y a eu avalanche; on ajuste le reseau */
        for (j=0 ; j<N ; j++ ) { sable[j]=sable[j]+move[j] ; }
     }
     else {                      /* Il n'y a pas eu avalanche; on force */
        jj=floor(1.*N*rand()/RAND_MAX) ; /* choix aleatoire d'un noeud */
        sable[jj]=sable[jj]+E*rand()/RAND_MAX ;    /* ajout de sable */
     }

     sable[N-1]=0. ;              /* Imposition des conditions limites */

     masse=0. ;                         /* calcul de la masse du reseau */
     for (j=0 ; j<N ; j++ ) { masse+=sable[j] ; }

     printf ("masses totale et deplacee: %f, %f\n",masse,dmasse) ;
   }
}
```

Figure 12.1: Code C pour le modèle TdS sur un réseau en une dimension, avec condition limite "mur vertical" au premier noeud, et condition limite ouverte au dernier noeud.

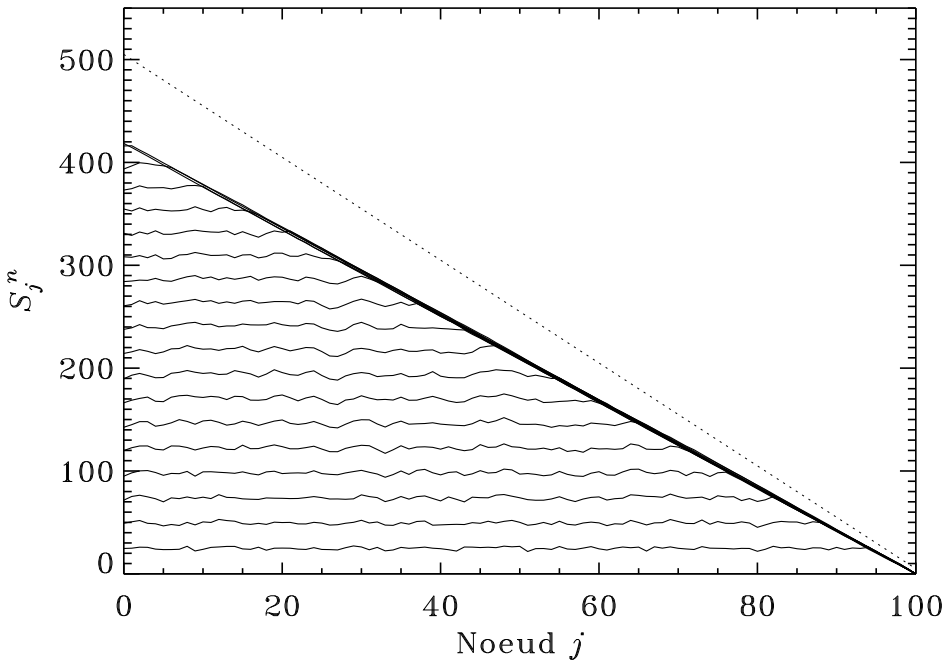

Figure 12.2: Croissance du tas de sable produit par le code C de la Figure 12.1, ici avec $J = 101$, $Z_c = 5$ et $E = 0.1$. Ici, et contrairement à la situation considérée à la §9.2 des notes de cours, la croissance est contrainte par un mur vertical à gauche (noeud $j = 0$). Le trait pointillé indique la forme attendue d'un tas dont la pente est égale à $-Z_c$. Chaque courbe est séparée de la précédente par 50000 itérations.

12.4 Le tas en tant qu'attracteur

Votre première tâche sera de rouler code ci-dessus, et de déterminer combien d'itérations sont requises pour atteindre un état statistiquement stationnaire. Faites ceci pour trois ou quatre conditions initiale différentes, par exemple:

1. `sable`= 0 à tous les noeuds;
2. `sable`= 500 à tous les noeuds;
3. `sable`= 500 aux premiers `N`/2 noeuds;
4. `sable`= 500 aux noeuds situés entre `N`/4 et 3`N`/4.

Vérifiez que dans tous les cas, une fois l'état stationnaire atteint la masse moyenne du tas (moyenne sur 10^4 itérations, disons) est la même pour toutes les conditions initiales. Par minimisation des moindres carrés (§5.6 des notes de cours), déterminez la pente du tas dans son état stationnaire et vérifiez que la valeur de cette pente est également indépendante de la condition initiale.

12.5 Caractéristiques des avalanche

Une fois l'état statistiquement stationnaire atteint, il s'agira de mesurer diverses propriétés des avalanches se déclenchant dans le système, plus spécifiquement:

1. La masse de l'avalanche (ΔM), soit la quantité totale de sable déplacé du début à la fin de l'avalanche;
2. Le pic de l'avalanche (P), soit la masse maximale de sable déplacé en une itération temporelle entre le début et la fin de chaque avalanche;
3. la durée de l'avalanche (T), soit le nombre d'itération écoulées entre le début et la fin de chaque avalanche.

Ceci peut se faire en même temps que la simulation, en suivant ce qui se passe avec le tableau `tsav` dans le code ci-dessus. Ce tableau compile la quantité totale de sable déplacé à chaque itération temporelle. Si une avalanche débute à l'itération n_1, alors on aura:

$$\texttt{tsav}[\texttt{n}_1] > 0 \ , \qquad \texttt{tsav}[\texttt{n}_1 - 1] = 0 \ , \qquad \text{avalanche commence}$$

tandis que si l'itération n_2 ($> n_1$) marque la fin d'une avalanche, on aura plutôt:

$$\texttt{tsav}[\texttt{n}_2] = 0 \ , \qquad \texttt{tsav}[\texttt{n}_2 - 1] > 0 \ , \qquad \text{avalanche se termine}$$

Donc on aurait, pour cette avalanche,

$$\Delta M = \sum_{n=n_1}^{n_2} \texttt{tsav}[\texttt{n}] \ ,$$
$$P = _{\max} (\texttt{tsav}[\texttt{n}]) \ , \qquad n = n_1, ..., n_2 \ ,$$
$$T = n_2 - n_1 + 1 \ .$$

Codez cette logique à la toute fin de l'itération temporelle dans le code de la Figure 12.1. Assurez vous de vous définir trois tableaux pour accumuler les valeurs des variables ΔM, P et T; les éléments `[0]` de ces tableaux contiendront les mesures associées à la première avalanche, les éléments `[1]` celles associées à la seconde, et ainsi de suite. Vous devrez donc vous définir une variable de type `int` (`iav`, disons) qui numérotera chaque avalanche de manière unique. Le plus facile est d'initialiser cette variable compteur à -1 avant le début de la boucle temporelle, et de l'incrémenter de 1 chaque fois qu'un début d'avalanche est détecté. Ça pourrrait avoir l'air de ceci:

```
if ( tsav[n] > 0 && tsav[n-1] == 0 ) {    /* debut d'avalanche */
   iav+=1   ;                             /* compteur d'avalanche */
   ndebut=n ;
}
if ( tsav[n] == 0 && tsav[n-1] > 0 ) {    /* fin d'avalanche */
   avmax=0. ;
   avsom=0. ;
   for ( k=ndebut ; k<n ; k++ ) {         /* somme et recherche du max */
      if (tsav[k] > avmax ) { avmax=tsav[k] ; }
      avsom+=tsav[k] ;
   }
   dmasse[iav]= avsom    ;               /* masse totale deplacee */
   pic[iav]   = avmax    ;               /* masse maximale deplacee */
   duree[iav] = n-ndebut ;               /* duree de l'avalanche */
}
```

Si tout se passe bien, à la sortie de votre boucle temporelle principale, vous aurez trois tableaux contenant les masses, pics et durées d'un nombre `iav` d'avalanches s'étant produites sur votre tas. Nous sommes prêt à débuter l'analyse statistique:

1. Portez en graphique les valeurs de P versus ΔM, et T versus ΔM. Ces variables sont-elles fortement correlées? Pourquoi?

2. Calculez les histogrammes $f(\Delta M)$, $f(P)$, $f(T)$ des variables ΔM, P et T;

3. Portez en graphique le logarithme de l'histogramme versus le logarithme de la variable même, i.e., $\log(f(\Delta M))$ versus $\log(\Delta M)$, etc.

4. Identifiez les intervalles sur ces histogrammes qui sont bien approximés par des droites, et mesurez les pentes correspondantes.

12.6 Dépendance sur les paramètres du modèle

Le code ci-dessus ajoute, à chaque itération où le tas est stable, une certaine quantité de sable à un noeud choisi aléatoirement, cette quantité étant également choisis aléatoirement dans un intervalle prédéterminé:

$$S_j^n \to S_j^n + r \times E \ , \qquad r \in [0,1] \ , \tag{12.1}$$

où r est un nombre aléatoire extrait d'une distribution uniforme dans l'intervalle $[0,1]$, et E est une amplitude prédéfinie ($E = 0.1$ dans le code ci-dessus).

On dit habituellement de systèmes en autorégulation critique qu'ils doivent être forcés "lentement". Dans le contexte du tas de sable, ceci veut dire que la quantité de sable déposée doit être petite par rapport à la quantité qui produirait une avalanche à tout coup; en d'autres mots, puisque l'intervalle "Δx" sur le réseau est toujours égal à un, celà revient à exiger que:

$$E \ll Z_c \ .$$

Il s'agit ici de vous faire examiner un peu cette question. Effectuez deux simulations avec $E = 10.0$ et $E = 1.0$, vous assurant de bien les pousser jusqu'à l'état statistiquement stationnaire. Poursuivez ensuite chaque simulation suffisamment longtemps pour produire quelques milliers d'avalanches, afin de pouvoir produire des statistiques raisonnables. Ensuite,

1. Mesurez de nouveau la pente du tas, et déterminez si elle dépend de E;

2. Recalculez les histogrammes des masses déplacées ΔM, et vérifiez si elles prennent toujours la forme de loi de puissance, et si oui, si la pente logarithmique varie avec E.

12.6.1 Bonus!

Encore une SUPER QUESTION BONUS: Généralisez le modèle à un réseau en deux dimensions spatiales, et vérifiez si la dimensionalité du réseau affecte la forme de l'histogramme des masses déplacées.

Ouf. C'est fini pour les labos réguliers...

Lectures supplémentaires:

La section 9.2 des Notes de cours est à lire impérativement avant ce Labo. Si vous êtes curieux par rapport à la dynamique des *vrais* tas de sable, je vous encourage à consulter l'excellent ouvrage:

Duran, J., *Sands, powders and grains*, Springer, 2000.

Appendice A

Le Rapport de Laboratoire

Ce petit document explicite les attentes par rapport à ce que vous devez nous produire comme rapport de laboratoire.

A.1 Contenu du rapport

A.1.1 Objectifs, théorie, etc.

Il ne s'agit vraiment pas de me retranscrire les notes de cours ou de laboratoire dans tous leurs détails. Une synthèse d'une demie page à une page résumant la problématique, la mathématique et la physique du problème est suffisante.

A.1.2 Algorithmes

Sans nous refaire toute la théorie contenue dans le notes de cours, vous devez expliciter quels algorithmes sont utilisés, et quelle forme ils prennent dans le contexte du problème spécifique qui est traité (e.g., Algorithme d'Euler explicite; appliqué au mouvement planétaire, $x_{k+1} = x_k + ...$, etc. Devrait habituellement tenir en une page.

A.1.3 Validation

Vous devez décrire comment vous avez validé votre code C, e.g., en calculant une orbite circulaire dont vous connaissez déjà les caractéristiques; en vérifiant le niveau auquel l'énergie est conservée, etc. Vous devez également documenter le niveau de précision numérique auquel vous avez pu arriver en utilisant l'algorithme sur ce problème-test. Devrait habituellement tenir en une page au max.

A.1.4 Résultats

Vos résultats doivent être présentés de manière claire et concise. L'utilisation de graphiques ou tableaux comparatifs est souvent optimale. Les notes de la laboratoire contiendront habituellement des instructions spécifiques par rapport aux questions auxquelles vous devez fournir des réponses basées sur vos calculs. N'hésitez pas à faire des calculs supplémentaires pour pousser plus loin vos analyses. C'est ici où vous pouvez utiliser votre créativité.

A.1.5 Discussion des erreurs et incertitudes

Cette partie du rapport est au moins aussi importante que la précédente. Le résultat d'une modélisation, qu'elle soit numérique ou mathématique, n'est jamais particulièrement convaincant si vous n'êtes pas en mesure de lui assujettir un estimé d'erreur. Vous devez donc discuter

les erreurs de nature numérique (troncation, choix d'un algorithme plutôt qu'un autre, etc.), ainsi que les erreurs conceptuelles associées à la formulation même du modèle utilisé. C'est ici où votre esprit critique devrait s'exprimer!

A.2 Présentation du rapport

Même si ca fait un peu Néanderthalien, les rapports doivent être remis sous forme papier, la remise électronique étant très difficile à gérer en pratique; de plus, vos correcteurs écriront des commentaires et explications dans les marges de vos rapports, forme de rétroaction qui, espérons-le, vous sera très utile dans la préparation des rapports subséquents.

Vous pouvez écrire votre rapport à la main, ou utiliser un programme d'édition de texte (e.g., Microsoft Word, TextEdit, etc.). Dans un cas comme dans l'autres, brochez ou reliez les pages de vos rapports!

Vos correcteurs prennent à coeur de vous servir une correction juste et équitable. Mais vous pourrez facilement imaginer que rendu à 1 heure 15 du matin, si le soixante-seizième de la pile de soixante-dix-huit s'avère être un torchon gribouillé au crayon 4H en pattes de mouches minuscules, celà causera des grincements de dents qui pourraient fort bien réverbérer jusqu'à la note assignée en bout de ligne au rapport.

A.2.1 Longueur

Il n'y a pas de longueur spéficique exigée pour les rapports; d'une semaine à l'autre certains seront plus longs que d'autres, contiendront plus de graphiques ou de tableaux, etc., le tout dépendant évidemment de la manière dont vous formattez vos Figures tableaux, et/ou de la taille de votre écriture. Ceci dit, si vous vous retrouvez en bas de 5 pages ou au dessus de 20, il y a fort probablement dérapage...

Vous devez inclure une page titre avec votre nom, code permanent, et titre du labo. Les correcteurs utiliseront parfois cette page titre pour y écrire des commentaires généraux, d'où l'idée de laisser du blanc même si ça a l'air d'un gaspillage de papier. L'impression recto-verso est encouragée mais pas obligatoire.

A.2.2 Qualité du français

PHY-1234 n'est pas un cours de français, mais vous devez tout de même nous remettre un rapport lisible avec un minimum de fautes d'orthographe, des phrases complètes (minimalement: sujet-verbe-complément!), etc. La plupart des programmes d'édition de texte incluent un programme de vérification de l'orthographe et, jusqu'à un certain point, de la grammaire. Utilisez-les! Si vous faites votre rapport à la main, n'hésitez pas à dépoussiérer le vieux dictionnaire Larousse qui traine dans vos tablettes, ou même, Aargh, le Bescherelle...

Je suis très au fait de ma propre fréquence moyenne de fautes de français par page de notes, et ne vous exigerai certainement pas un français écrit parfait. Vous ne serez pas pénalisés pour une erreur occasionnelle, mais quand les erreurs grammaticales ou orthographiques deviennent trop denses, la lecture de votre rapport s'en trouve affectée négativement et ceci sera reflété dans votre note.

A.2.3 Graphiques et Tableaux

Parfois les notes de laboratoire demanderont explicitement de présenter les résultats sous forme de graphique ou de tableau, parfois ce choix sera laissé à votre jugement. Vous pouvez aussi choisir d'inclure un tableau ou graphique qui n'est pas explicitement demandé, si vous jugez que la présentation de vos résultat s'en trouve clarifiée.

Les graphiques et tableaux doivent êre numérotés, et cette numérotation utilisée dans le texte pour y faire référence. Chaque tableau ou graphique doit aussi avoir un titre, et, si le moindrement complexe, une légende explicative. Les axes des graphiques doivent aussi avoir un

titre, incluant les unités utilisées, le cas échéant. Même chose pour la première ligne ou colonne de votre tableau, listant les quantités tabulées. Voir les Notes de cours pour des exemples spécifiques.

A.2.4 Références

Dans la préparation de votre rapport vous pouvez certainement aller chercher des informations ailleurs que dans les Notes de cours ou de laboratoire: livres de référence, pages Web, etc. Dans un tel cas vous devez obligatoirement indiquer la référence complète (bibliographique, URL, etc.) permettant de retracer de manière non-ambigue ces informations. Et attention, pas de copier-coller de pages Web; vous devez faire la **synthèse** de l'information trouvée en vos propres mots. (Les correcteurs aussi savent aller sur Google!). Toute instance de copier-coller détectée sera considérée comme un cas de plagiat (voir §A.4 ci-dessous).

A.3 Remise des rapports

Les rapports doivent obligatoirement être remis pas plus tard qu'**en entrant** au laboratoire de la semaine suivant le labo remis (et à l'heure nominale du début du labo!). Vous pouvez évidemment remettre votre rapport avant, dans le casier à devoir à coté du secrétariat du département de physique, dans la bonne fente SVP! Les rapports remis en retard sans excuse jugée valable seront sujets à une déduction de 2.5 points sur 10 par tranche de 24 heures.

A.4 Plagiat

Vous remarquerez rapidement qu'au cours de séances de laboratoire nous vous encourageons à vous entraider, débattre entre vous de la meilleure manière de coder quelquechose, etc, et cette collaboration peut fort bien continuer une fois sortis du labo, au niveau de l'interprétation de vos résultats, etc. Donc, pas besoin de plonger sous la table si je m'avère à entrer à la Planck pour m'acheter un brownie. Cependant, quand vient le temps d'écrire votre rapport vous devez faire ça **seul(e)**.

La politique UdeM par rapport au plagiat sera appliquée impitoyablement: zéro pour tous les travaux identiques, peu importe qui a copié sur qui. La troisième infraction entraine l'échec automatique au cours.

A.5 Pondération

Les correcteurs utiliseront la pondération suivante pour noter vos rapports:

1. Introduction (objectifs, théorie, algorithmes, etc.): 1 point sur 10.
2. Validation (voir §A.1.3): 1 point sur 10.
3. Présentation des résultats et réponses aux questions spécifiques: 3 points sur 10
4. Discussion critique et estimés d'erreurs et incertitudes: 3 points sur 10.
5. Codes C en annexe (clarté du code, indentation, commentaires explicatifs, etc.) 1 point sur 10.
6. Qualité de la présentation (français, voir §A.2.2; graphiques/tableaux, voir §A.2.3; etc): 1 point sur 10.
7. Questions bonus (quand il y en a): 1 point sur 10 supplémentaire si réponse correcte