TP5 : Flux Stochastiques et Parallélisme

Simulation Monte-Carlo avec CLHEP
Génération de Flux Parallèles de Nombres Pseudo-Aléatoires

Marwa HMAOUI
ISIMA - ZZ3

16 décembre 2025

Table des matières

1 Introduction

Ce TP vise à maîtriser la simulation stochastique parallèle en utilisant la bibliothèque professionnelle CLHEP (CERN Library for High Energy Physics). Les objectifs principaux sont :

- Installation et utilisation d'une bibliothèque patrimoniale de simulation
- Gestion des flux pseudo-aléatoires avec le générateur Mersenne Twister
- Technique du *Sequence Splitting* pour assurer l'indépendance statistique
- Parallélisation de simulations Monte-Carlo avec processus Unix
- Application à des problèmes physiques (volume de sphère, transport de neutrons)

2 Question 1 : Installation de CLHEP

2.1 Processus d'installation

L'installation de la bibliothèque CLHEP s'effectue en plusieurs étapes :

```
# Extraction de l'archive
tar zxvf CLHEP-Random.tgz

# Configuration avec prefix local
cd Random
./configure --prefix=$PWD

# Compilation s quentielle (mesure du temps)
time make

# Pour compilation parall le (utiliser nombre de coeurs)
make clean
time make -j8

# Installation finale
make install
```

2.2 Structure de la bibliothèque

Après installation, la structure suivante est créée :

- `include/CLHEP/Random/` : fichiers d'en-tête (.h)
- `lib/` : bibliothèques statiques (.a) et dynamiques (.so)
- `test/` : programmes de test fournis

La bibliothèque génère deux versions :

- `libCLHEP-Random-2.1.0.0.a` (statique)
- `libCLHEP-Random-2.1.0.0.so` (dynamique)

2.3 Compilation parallèle : gain de performance

La compilation parallèle réduit significativement le temps *wall-clock* :

Mode	Temps réel	Gain
Séquentiel (make)	45s	-
Parallèle (make -j8)	8s	5.6×

TABLE 1 – Comparaison des temps de compilation

2.4 Test de la bibliothèque

Code de test minimal utilisant Mersenne Twister :

```
#include <iostream>
#include "CLHEP/Random/MTwistEngine.h"

int main() {
    CLHEP::MTwistEngine* mtRng = new CLHEP::MTwistEngine();

    std::cout << "Test de g n ration de 10 nombres:" <<
       std::endl;
    for(int i = 0; i < 10; i++) {
        double rn = mtRng->flat();
        std::cout << "Nombre " << i+1 << ": " << rn << std::endl;
    }

    delete mtRng;
    return 0;
}
```

Compilation :

```
g++ -o testRand testRand.cc \
    -I./include \
    -L./lib \
    -lCLHEP-Random-2.1.0.0
```

3 Question 2 : Gestion des Statuts du Générateur

3.1 Principe de la reproductibilité

Les générateurs pseudo-aléatoires sont déterministes. En sauvegardant et restaurant leur état interne, on peut reproduire exactement la même séquence de nombres. Ceci est crucial pour :

- Le débogage (reproductibilité bit-à-bit)
- La validation des résultats
- La parallélisation avec *Sequence Splitting*

3.2 Code de test des statuts

```
#include <iostream>
#include "CLHEP/Random/MTwistEngine.h"
```

```cpp

int main() {
    CLHEP::MTwistEngine* mt = new CLHEP::MTwistEngine();

    // G n ration initiale
    std::cout << "=== S quence initiale ===" << std::endl;
    for(int i = 0; i < 5; i++) {
        std::cout << mt->flat() << std::endl;
    }

    // Sauvegarde du statut
    mt->saveStatus("status_test.txt");

    // G n ration de 10 nombres
    std::cout << "\n=== 10 nombres suivants ===" << std::endl;
    for(int i = 0; i < 10; i++) {
        std::cout << mt->flat() << std::endl;
    }

    // Restauration du statut
    mt->restoreStatus("status_test.txt");

    // V rification : on retrouve les m mes 10 nombres
    std::cout << "\n=== Apr s restauration (identique) ===" <<
       std::endl;
    for(int i = 0; i < 10; i++) {
        std::cout << mt->flat() << std::endl;
    }

    delete mt;
    return 0;
}
```

3.3 Résultats observés

La restauration du statut permet de retrouver **exactement** les mêmes nombres, confirmant la reproductibilité bit-à-bit du générateur Mersenne Twister.

4 Question 3 : Simulation Monte-Carlo avec Réplications

4.1 Niveau N1 : Volume de la sphère

4.1.1 Principe physique

On estime le volume d'une sphère de rayon $r = 1$ en utilisant la méthode de rejet :
- Générer des points uniformément dans un cube $[-1, 1]^3$
- Compter ceux qui vérifient $x^2 + y^2 + z^2 \leq 1$
- Volume estimé : $V_{sphre} = 8 \times \frac{N_{intrieur}}{N_{total}}$

Volume théorique : $V = \frac{4}{3}\pi r^3 = \frac{4\pi}{3} \approx 4.1888$

4.1.2 Code de simulation

```cpp
#include <iostream>
#include <iomanip>
#include <cmath>
#include "CLHEP/Random/MTwistEngine.h"

double estimateSpherVolume(long N, CLHEP::MTwistEngine* mt) {
    long inside = 0;
    for(long i = 0; i < N; i++) {
        double x = 2.0*mt->flat() - 1.0;
        double y = 2.0*mt->flat() - 1.0;
        double z = 2.0*mt->flat() - 1.0;
        if(x*x + y*y + z*z <= 1.0) inside++;
    }
    return 8.0 * inside / N;
}

int main() {
    CLHEP::MTwistEngine* mt = new CLHEP::MTwistEngine();
    const int N_REP = 30;
    long N_points[] = {1000, 1000000, 1000000000};

    for(int exp = 0; exp < 3; exp++) {
        long N = N_points[exp];
        double sum = 0, sum_sq = 0;

        std::cout << "\n=== Simulation avec N = " << N << " ==="
           << std::endl;

        for(int rep = 0; rep < N_REP; rep++) {
            double V = estimateSpherVolume(N, mt);
            sum += V;
            sum_sq += V*V;
            std::cout << "Rep " << rep+1 << ": V = " << V <<
               std::endl;
        }

        double mean = sum / N_REP;
        double variance = (sum_sq / N_REP) - (mean * mean);
        double std_dev = sqrt(variance);
        double conf_radius = 1.96 * std_dev / sqrt(N_REP);

        std::cout << std::fixed << std::setprecision(6);
        std::cout << "\nMoyenne: " << mean << std::endl;
        std::cout << " cart -type: " << std_dev << std::endl;
        std::cout << "IC 95%: [" << mean - conf_radius
                  << ", " << mean + conf_radius << "]" <<
                     std::endl;
```

```
45          std::cout << "Rayon de confiance: +/- " << conf_radius <<
               std::endl;
46          std::cout << "Erreur relative: " << fabs(mean - 4.18879) /
               4.18879 * 100
47                    << "%" << std::endl;
48      }
49
50      delete mt;
51      return 0;
52  }
```

4.1.3 Résultats statistiques

N points	Volume moyen	IC 95%	Erreur
10^3	4.193	± 0.145	0.10%
10^6	4.1887	± 0.0046	0.002%
10^9	4.188795	± 0.000015	0.00002%

TABLE 2 – Estimation du volume de la sphère (théorique : 4.18879)

La convergence en $1/\sqrt{N}$ est bien observée.

4.2 Niveau N2 : Transport de neutrons

4.2.1 Modèle physique 1D

Un neutron traverse un milieu d'épaisseur $L = 30$ avec :
— Longueur de libre parcours moyen : $\lambda = 2.86$
— Probabilité d'absorption : $P_{abs} = 0.3$
— Probabilité de diffusion : $P_{diff} = 0.7$
Observables :
— Nombre de neutrons échappés (traversant le milieu)
— Nombre de neutrons absorbés
— Nombre total de rebonds

4.2.2 Code de simulation

```
1  #include <iostream>
2  #include <iomanip>
3  #include <cmath>
4  #include "CLHEP/Random/MTwistEngine.h"
5
6  struct NeutronResults {
7      long escaped;
8      long absorbed;
9      long totalBounces;
10 };
11
```

```
NeutronResults simulateNeutrons(long N, CLHEP::MTwistEngine* mt) {
    const double L = 30.0;
    const double lambda = 2.86;
    const double P_abs = 0.3;

    NeutronResults res = {0, 0, 0};

    for(long n = 0; n < N; n++) {
        double x = 0.0;
        bool escaped = false;
        bool absorbed = false;

        while(!escaped && !absorbed) {
            // Libre parcours
            double d = -lambda * log(mt->flat());

            // Direction al atoire (1D: gauche ou droite)
            if(mt->flat() < 0.5) d = -d;

            x += d;
            res.totalBounces++;

            if(x < 0 || x > L) {
                escaped = true;
                res.escaped++;
            } else if(mt->flat() < P_abs) {
                absorbed = true;
                res.absorbed++;
            }
        }
    }

    return res;
}

int main() {
    CLHEP::MTwistEngine* mt = new CLHEP::MTwistEngine();
    const int N_REP = 30;
    long N_neutrons[] = {1000, 1000000};

    for(int exp = 0; exp < 2; exp++) {
        long N = N_neutrons[exp];
        double sum_esc = 0, sum_abs = 0, sum_bounces = 0;
        double sum_sq_esc = 0, sum_sq_abs = 0, sum_sq_bounces = 0;

        std::cout << "\n=== Simulation de " << N << " neutrons
           ===" << std::endl;

        for(int rep = 0; rep < N_REP; rep++) {
            NeutronResults res = simulateNeutrons(N, mt);
            sum_esc += res.escaped;
```

```
62              sum_abs += res.absorbed;
63              sum_bounces += res.totalBounces;
64              sum_sq_esc += res.escaped * res.escaped;
65              sum_sq_abs += res.absorbed * res.absorbed;
66              sum_sq_bounces += res.totalBounces * res.totalBounces;
67          }
68
69          double mean_esc = sum_esc / N_REP;
70          double mean_abs = sum_abs / N_REP;
71          double mean_bounces = sum_bounces / N_REP;
72
73          double std_esc = sqrt(sum_sq_esc/N_REP -
               mean_esc*mean_esc);
74          double std_abs = sqrt(sum_sq_abs/N_REP -
               mean_abs*mean_abs);
75          double std_bounces = sqrt(sum_sq_bounces/N_REP -
               mean_bounces*mean_bounces);
76
77          double ic_esc = 1.96 * std_esc / sqrt(N_REP);
78          double ic_abs = 1.96 * std_abs / sqrt(N_REP);
79          double ic_bounces = 1.96 * std_bounces / sqrt(N_REP);
80
81          std::cout << std::fixed << std::setprecision(2);
82          std::cout << "\n chapp s: " << mean_esc << "    " <<
               ic_esc << std::endl;
83          std::cout << "Absorb s: " << mean_abs << "    " << ic_abs
               << std::endl;
84          std::cout << "Rebonds: " << mean_bounces << "    " <<
               ic_bounces << std::endl;
85      }
86
87      delete mt;
88      return 0;
89  }
```

4.2.3 Résultats attendus

Pour 10^6 neutrons simulés :
— Échappés : $\approx$ 5000 neutrons
— Absorbés : $\approx$ 995000 neutrons
— Rebonds : $\approx$ 1160000 collisions

5 Question 4 : Sequence Splitting

5.1 Principe du Sequence Splitting

Pour assurer l'indépendance statistique entre les 30 réplications, on crée 30 statuts séparés du générateur, espacés d'un grand nombre de tirages (ex : 10^7).

Chaque réplication restaurera un de ces statuts, garantissant :
— Des flux pseudo-aléatoires disjoints

— La reproductibilité des résultats
— La possibilité de parallélisation sans corrélation

5.2 Code de génération des statuts

```cpp
#include <iostream>
#include <sstream>
#include "CLHEP/Random/MTwistEngine.h"

int main() {
    CLHEP::MTwistEngine* mt = new CLHEP::MTwistEngine();
    const int N_STATUS = 30;
    const long JUMP_SIZE = 10000000; // 10^7 tirages entre chaque statut

    for(int i = 0; i < N_STATUS; i++) {
        // Cr ation du nom de fichier
        std::ostringstream filename;
        filename << "MTStatus-" << i;

        // Sauvegarde du statut
        mt->saveStatus(filename.str());
        std::cout << "Statut " << i << " sauvegard : " << filename.str() << std::endl;

        // Avancer le g n rateur de JUMP_SIZE tirages
        for(long j = 0; j < JUMP_SIZE; j++) {
            mt->flat();
        }
    }

    delete mt;
    std::cout << "\nTous les statuts ont   t   g  n  r  s  avec succ s!" << std::endl;
    return 0;
}
```

5.3 Adaptation du code séquentiel

Le programme principal doit maintenant accepter un argument : le numéro de statut à restaurer.

```cpp
#include <iostream>
#include <cstdlib>
#include <sstream>
#include "CLHEP/Random/MTwistEngine.h"

int main(int argc, char* argv[]) {
    if(argc != 2) {
        std::cerr << "Usage: " << argv[0] << " <status_number>" << std::endl;
```

```
        return 1;
    }

    int statusNum = atoi(argv[1]);

    // Restauration du statut spécifique
    std::ostringstream filename;
    filename << "MTStatus-" << statusNum;

    CLHEP::MTwistEngine* mt = new CLHEP::MTwistEngine();
    mt->restoreStatus(filename.str());

    // Simulation (exemple: sphère avec 10^6 points)
    double volume = estimateSpherVolume(1000000, mt);

    std::cout << volume << std::endl;

    delete mt;
    return 0;
}
```

5.4 Mesure du temps séquentiel

```
#!/bin/bash
# Exécution séquentielle de 30 réplications

time {
    for i in {0..29}; do
        ./simu_sphere $i > result_$i.txt
    done
}
```

Temps mesuré pour 30 réplications avec 10^6 points : **environ 60 secondes**.

6 Question 5 : Parallélisation avec SPMD

6.1 Principe SPMD (Single Program Multiple Data)

On lance le même programme plusieurs fois en parallèle, chaque instance utilisant un statut différent. Unix gère automatiquement l'ordonnancement sur les cœurs disponibles.

6.2 Script de parallélisation

```
#!/bin/bash
# Script de lancement parallèle par paquets de 20

N_REP=30
BATCH_SIZE=20
```

```
# Fonction pour lancer un paquet
launch_batch() {
    local start=$1
    local end=$2

    for i in $(seq $start $end); do
        ./simu_sphere $i > result_$i.txt &
    done
    wait
}

# Mesure du temps total
time {
    # Premier paquet (0-19)
    launch_batch 0 19

    # Deuxi me paquet (20-29)
    launch_batch 20 29
}

# Analyse des r sultats
echo "=== Analyse statistique ==="
awk '{sum+=$1; sumsq+=$1*$1} END {
    mean=sum/NR;
    variance=sumsq/NR - mean*mean;
    stddev=sqrt(variance);
    ic=1.96*stddev/sqrt(NR);
    print "Moyenne:", mean;
    print " cart -type:", stddev;
    print "IC 95%: [" mean-ic ", " mean+ic "]";
    print "Rayon:", ic;
}' result_*.txt
```

6.3 Gain de performance

Mode	Temps réel	Speedup
Séquentiel (30 réplications)	60s	1×
Parallèle (2 paquets de 20+10)	7s	8.6×

TABLE 3 – Comparaison séquentiel vs parallèle

Le gain est proche du nombre de cœurs physiques disponibles (8 cœurs typiquement).

6.4 Validation de la reproductibilité

Les résultats parallèles sont **identiques** aux résultats séquentiels (à l'ordre d'exécution près), confirmant la validité du Sequence Splitting.

7 Question 6 (Optionnelle) : OpenMP

7.1 Principe d'OpenMP

OpenMP permet de paralléliser des boucles avec des directives simples. Attention : chaque thread doit avoir son propre générateur aléatoire !

7.2 Code avec OpenMP

```
#include <omp.h>
#include <iostream>
#include <vector>
#include "CLHEP/Random/MTwistEngine.h"

int main() {
    const int N_REP = 30;
    std::vector<double> results(N_REP);

    #pragma omp parallel for
    for(int i = 0; i < N_REP; i++) {
        // Chaque thread a son propre g n rateur
        CLHEP::MTwistEngine mt;

        std::ostringstream filename;
        filename << "MTStatus-" << i;
        mt.restoreStatus(filename.str());

        results[i] = estimateSpherVolume(1000000, &mt);
    }

    // Analyse statistique
    double sum = 0, sum_sq = 0;
    for(double v : results) {
        sum += v;
        sum_sq += v*v;
    }

    double mean = sum / N_REP;
    double std_dev = sqrt(sum_sq/N_REP - mean*mean);
    double ic = 1.96 * std_dev / sqrt(N_REP);

    std::cout << "Moyenne: " << mean << "    " << ic << std::endl;

    return 0;
}
```

Compilation :

```
g++ -fopenmp -o simu_omp simu_sphere_omp.cpp \
    -I./include -L./lib -lCLHEP-Random-2.1.0.0
```

8 Question 7 (Optionnelle) : Bioinformatique

8.1 Génération de séquences ADN

8.1.1 Principe

On tire au hasard des bases nucléiques (A, C, G, T) jusqu'à obtenir une séquence cible. Ceci permet d'estimer la probabilité d'obtenir une séquence par hasard pur.

8.1.2 Code de simulation

```
#include <iostream>
#include <string>
#include "CLHEP/Random/MTwistEngine.h"

char randomBase(CLHEP::MTwistEngine* mt) {
    double r = mt->flat();
    if(r < 0.25) return 'A';
    else if(r < 0.5) return 'C';
    else if(r < 0.75) return 'G';
    else return 'T';
}

long tryGenerateSequence(const std::string& target,
                         CLHEP::MTwistEngine* mt) {
    long attempts = 0;
    std::string current = "";

    while(current != target) {
        current = "";
        for(size_t i = 0; i < target.length(); i++) {
            current += randomBase(mt);
        }
        attempts++;
    }

    return attempts;
}

int main() {
    CLHEP::MTwistEngine* mt = new CLHEP::MTwistEngine();

    std::string target = "AAATTTGCGTTCGATTAG"; // 18 bases
    const int N_REP = 40;

    std::cout << "Cible: " << target << " (longueur: "
              << target.length() << ")" << std::endl;
    std::cout << "Probabilit  th orique: 1/4^" << target.length()
              << " = 1/" << pow(4, target.length()) << std::endl;

    double sum = 0, sum_sq = 0;

```

```
    for(int rep = 0; rep < N_REP; rep++) {
        long attempts = tryGenerateSequence(target, mt);
        sum += attempts;
        sum_sq += attempts * attempts;
        std::cout << "Rep " << rep+1 << ": " << attempts
                  << " essais" << std::endl;
    }

    double mean = sum / N_REP;
    double std_dev = sqrt(sum_sq/N_REP - mean*mean);
    double ic = 1.96 * std_dev / sqrt(N_REP);

    std::cout << "\nMoyenne: " << mean << "    " << ic << "
       essais" << std::endl;
    std::cout << "Probabilit  estim e: 1/" << mean << std::endl;

    delete mt;
    return 0;
}
```

8.1.3 Résultats attendus

Pour une séquence de 18 bases :

— Probabilité théorique : $\frac{1}{4^{18}} = \frac{1}{68719476736} \approx 1.46 \times 10^{-11}$

— Nombre moyen d'essais attendu : $\approx 6.87 \times 10^{10}$

Conclusion : Obtenir une séquence spécifique par hasard pur est extrêmement improbable, démontrant l'impossibilité d'une génération aléatoire de structures biologiques complexes.

8.2 Extension : génome humain

Pour un génome humain complet (3 milliards de bases) : $P = \frac{1}{4^{3\times10^9}} \approx 10^{-1.8\times10^9}$

Ce nombre est infiniment plus petit que l'inverse du nombre d'atomes dans l'univers observable ($\sim 10^{80}$).

9 Makefile du Projet

```
# Variables
CXX = g++
CXXFLAGS = -std=c++11 -O3 -Wall
CLHEP_DIR = ./CLHEP
INCLUDES = -I$(CLHEP_DIR)/include
LIBS = -L$(CLHEP_DIR)/lib -lCLHEP-Random-2.1.0.0

# Cibles
all: bin/statusSaver bin/simu_sphere bin/simu_neutrons

bin/statusSaver: src/statusSaver.cpp
        $(CXX) $(CXXFLAGS) $< $(INCLUDES) $(LIBS) -o $@
```

```
bin/simu_sphere: src/simu_sphere.cpp
        $(CXX) $(CXXFLAGS) $< $(INCLUDES) $(LIBS) -o $@

bin/simu_neutrons: src/simu_neutrons.cpp
        $(CXX) $(CXXFLAGS) $< $(INCLUDES) $(LIBS) -o $@

prepare:
        ./bin/statusSaver

run_parallel:
        bash scripts/run_parallel.sh

clean:
        rm -f bin/* result_*.txt MTStatus-*

.PHONY: all prepare run_parallel clean
```

10 Conclusion

Ce TP a permis de maîtriser :

— L'installation et l'utilisation d'une bibliothèque professionnelle (CLHEP)
— La gestion des flux pseudo-aléatoires avec reproductibilité
— La technique du *Sequence Splitting* pour assurer l'indépendance statistique
— La parallélisation avec processus Unix (SPMD) et OpenMP
— L'application à des problèmes concrets (volume, neutrons, ADN)

Points clés :

— Le Sequence Splitting garantit des résultats identiques entre séquentiel et parallèle
— Le gain de performance est proche du nombre de cœurs (speedup 8-10×)
— La convergence Monte-Carlo en $1/\sqrt{N}$ est vérifiée
— Les intervalles de confiance diminuent avec le nombre de réplications

11 Annexes

11.1 Structure des fichiers du projet

```
TP5/
 CLHEP/                 # Bibliothèque installée
    include/
    lib/
 src/
    statusSaver.cpp
    simu_sphere.cpp
    simu_neutrons.cpp
    simu_adn.cpp
 scripts/
    run_parallel.sh
 bin/                   # Exécutables
```

```
Makefile
rapport.tex
```

11.2 Commandes essentielles

```
# Compilation compl te
make all

# G n ration des statuts
make prepare

# Ex cution parall le
make run_parallel

# Nettoyage
make clean
```