TP : Flux stochastiques – modèles du hasard - Nombres quasi-aléatoires - Parallélisme
Génération de flux parallèles de nombres pseudo-aléatoires
Confrontation avec une bibliothèque professionnelle et scientifique – CLHEP
PJ : bibliothèque CLHEP, code de calcul intégral simple en 2D (cf. Simu ZZ2)

Les buts principaux du TP sont les suivants : se confronter à une bibliothèque « patrimoniale » de taille professionnelle pour la simulation (Physique des Hautes Energies : CLHEP) et de réaliser un calcul stochastique parallèle en utilisant les processus Unix. ATTENTION : bien **lire le sujet et les TIPS** avant d'attaquer les questions. Travailler à partir de la version proposée pour CLHEP, cette ancienne version est portable avec un compilateur C++ usuel (g++ notamment) avec les modes d'installation ancestraux sur Unix. **Il est fortement conseiller d'utiliser directement vos machines virtuelles sous Linux.**
Si vous travaillez sur une machine Windows personnelle, installer le sous-sytème Linux sous Windows (Ubuntu ou une autre distribution de Linux) - https://korben.info/tuto-pour-installer-linux-sous-windows-10-avec-wsl.html et les packages de développement. Cela fonctionne avec Windows.

1) Installer la bibliothèque (observer la hiérarchie des répertoires, où sont les sources, les include, les codes de test (les redondances !) et repérer où est l'emplacement de la librairie une fois compilée (vérifier la date – les anciennes versions devraient être datées de 2005. Si le processus se passe bien vous aurez un .a et un .so daté du jour à la fin de l'édition des liens). Il faut compiler, lier puis regarder les nouveaux répertoires qui sont créés, la création des bibliothèques de leurs chemins d'accès… Tester cette bibliothèque avec le code C++ donné en fin de TP et le code de test fourni (`testRand.cc`)

2) Archiver quelques fichiers de statuts pour le générateur Mersenne Twister (MT) avec la méthode saveStatus. Séparer ces statuts d'un petit nombre de tirages (10 nombres par exemple). Tester la restauration à un statut donné avec la méthode restoreStatus. On retrouve les mêmes nombres pseudo-aléatoires pour reproduire une séquence, la déboguer etc… La répétabilité bit à bit est nécessaire pour déboguer en informatique (si on ne peut plus déboguer l'informatique déterministe est morte…). Pour mémoire, les générateurs des nombres pseudo-aléatoires sont des algorithmes déterministes, ce ne sont que des 'simulations' du hasard de plus ou moins bonne qualité, mais qui peuvent (doivent) reproduire les mêmes séquences en fonction de leurs status initiaux.

3) Faire un code C++ utilisant la méthode de Monte Carlo et qui enchaînera 30 réplications d'un calcul. Chaque simulation produira un nombre limité d'événements. (N1) Si vous êtes débutants, vous garderez comme calcul la simulation du volume d'une sphère (avec 10^3, 10^6 et 10^9 points). (N2) Vous Développer un code qui simule l'envoi de neutrons à travers de l'eau. Vous étudierez un code fourni en 1 dimension, puis vous testerez le principe en 2D et 3D en fournissant des codes C++. Les résultats attendus pour 1 000 et 1000 000 de neutrons simulés sont : combien de neutrons se sont échappés (ont traversé l'eau – autour de 5000 pour un million de neutrons). Combien ont été absorbés et combien de rebonds sont observés ? Calculer les rayons de confiance autour des moyennes de ces 3 variables (pour le N1 : faire l'intervalle autour de la surface estimée).

4) Faire un code qui sauve 30 statuts séparés par « assez d'espace » pour les différentes expériences les plus consommatrices en tirages de nombres pseudo aléatoires. L'idée permettra de mettre en œuvre du « Sequence Splitting », à tester en séquentiel (puis en parallèle à la question 5) pour évaluer les réplications. Le gain en performance est validé si les simulations parallèles donnent le même résultat !

   Pour les étudiants les plus compétents, mettre en place la technique du pré-calcul des nombres pseudo-aléatoires étudiée dans le TP2.

   Chaque réplication doit utiliser un flux indépendant de nombres pseudo-aléatoires, le même à chaque exécution de cette réplication. Pour connaître l'état du générateur pour chaque réplication on peut le

restaurer dans un des états pré-calculés (utiliser la méthode restoreStatus au début de chaque réplication).

Mesurer le temps de calcul séquentiel pour 30 réplications.

5) Le temps calcul en parallèle est fonction des performances de votre machine et du nombre de cœurs disponibles. Paralléliser l'exécution en utilisant la technique du « sequence splitting », pour calculer simultanément par paquet de 20 réplications.

   L'approche la plus simple est de garder votre programme séquentiel et d'utiliser le système multitâche d'Unix pour faire du « SPMD » (Simple Program Multiple Data). Dans notre cas, les données qui changent sont uniquement le flux aléatoire.

   Réfléchir à la façon de faire puis proposer une façon de lancer **en parallèle** 30 réplications d'une simulation sur un serveur multi-cœur (0 à X cœurs logiques) avec des flux stochastiques indépendants.

   Les résultats de la sortie standard peuvent être redirigés dans un fichier). Chaque ligne de commande peut être lancée en tache de fond (& cf. exemple de ligne de commande ci-dessous pour la 7ème simulation – utilisant en entrée le statut7 de MT et écrivant dans le fichier de sortie n°7. Les commandes pour par paquet de 20 simulations peuvent être regroupées dans un script qui lancera toutes les réplications en parallèle. Exemple pour une ligne :

```
$ simuSV3D MTStatus-7 > EstimationSV3D-7 &
```

Question optionnelles pour les étudiants intéressés par le calcul parallèle.

6) Découvrez par vous-même une technique ou bibliothèque de parallélisation – partir de l'exemple de codes développé et reprendre la question 5. Dans cette 6ème partie, au lieu d'avoir des programmes séquentiels indépendants qui utilisent chacun un flux et qui sont lancés en parallèle sous Unix, vous pourrez utiliser une des principales bibliothèques de parallélisme : OpenMP pour lancer en parallèle simulation de la question 5.

7) Application à la bioinformatique : Utiliser MT pour générer des mots et des portions de phrases avec des caractères tirés au hasard. Par exemple écrire un programme simple qui essayer de générer au hasard l'oligopeptide : « gattaca » - https://fr.wikipedia.org/wiki/Bienvenue_%C3%A0_Gattaca
   Quelques rappels de biologie moléculaires peuvent être trouvés ci-dessous :
   https://www.supagro.fr/ress-tice/ue1-ue2_auto/Bases_Biologie_Moleculaire_v2/co/0_module_bases_biologie_moleculaire_V2_1.html

   Chaque tirage donne une base nucléique – symbolisé par une lettre avec 4 possibilités ('A', 'C', 'G', 'T'). Il est facile de dénombrer le nombre de combinaisons possibles. Simuler ce processus et compter le nombre d'essais pour obtenir une 1ère réussite qui trouve la séquence : « AAATTTGCGTTCGATTAG » (sur une seule réplication). Puis faire 40 réplications indépendantes pour donner un nombre moyen d'essais et un rayon de confiance. Dans un 1er temps ne pas réinitialiser le générateur et travailler en séquentiel. Puis utiliser la technique du « sequence splitting » en restaurant des statuts de MT espacés de façon adaptée pour un travail en parallèle. Quand vous aurez fait plusieurs simulations, vous pourrez calculer une moyenne de probabilité et un écart type, dans l'idéal donnez l'intervalle de confiance. Comparer avec la probabilité exacte. Essayer de générer au hasard une phrase intelligible : « Le hasard n'écrit pas de messages ». Le hasard écrit encore moins facilement des programmes (ou des gènes). En se basant sur les anciens codes ASCII, estimer à l'avance le temps de calcul et la probabilité d'obtenir par hasard cette chaine. Puis enfin, estimer la probabilité d'avoir 'par hasard' la chaîne exacte d'un être

humain avec tous ses gènes fonctionnels – on peut considérer qu'elle comporte 3 milliards de bases nucléiques. Au lieu de réfléchir au temps à la probabilité de générer une très très longue phrase intelligible, on peut chercher la probabilité de trouver « au hasard » le texte cohérent d'un livre avec des chapitres, paragraphes, phrases…

TIPS to install a patromonial CLHEP library :

Install the CLHEP library (given in .tgz). Check the hierarchy of folders, where are the sources, include… **before and after installation** – list files & folders with dates. **Look at ALL the tips below** before installing. The Unix command tar below (Tape ARchive) will unzip the .tgz with a verbose formatted manner).

```
$ tar zxvf CLHEP-Random.tgz
```

In your installation process, you will first make a sequential installation. First, find the configure script in the *Random* directory that has been created, launch it before making the compilation, check and note the total compilation time.

```
./configure --prefix=$PWD // where PWD is the installation directory
time make                 // look at how much time is needed for sequential compilation
```

Then, remove the created directories (`rm –fr ./Random`). Now you will compile in parallel to exploit the full potential of the SMP machine you are using for the labs. The etud server proposes many logical cores. Check and note how much time you save with a parallel compilation (though you are all sharing the same machine). The user time will approximatively be the same, the sys time also, but the real time (wall clock) – **your real waiting time** – will be reduced thanks to this parallelism.

```
./configure --prefix=$PWD // where PWD is the installation directory
time make –jXXX           // specifies a parallel compilation with XXX logical cores
make install              // this rule of the Makefile finishes the installation
```

Test the code proposed at the end of this file and also the main test proposed in the package (Full testing – the code is present in the test Directory of the installed library). In order to compile the code, you need to look where the '*include'* and '*lib'* libraries have been installed. If the installation went correctly, you will find two versions of the CLHEP library. The date/time for both files will correspond to your compilation. One will be the static version with a '.a' extension (like libm.a for the static version of the standard math library). You can see the object files inside a static library with the regular *ar* Unix command. Ex: `ar –t libm.a` lists all the object files of the standard C math library.

```
libCLHEP-Random-2.1.0.0.a
```

The other one will be a dynamic version '.so' – for shared objects – this corresponds to DLLs (Dynamic Link Libraries) under Windows). In this version, only one version of each compiled object file will be loaded in memory for all executables. Objects will be shared and re-entered at runtime by executing processes.

```
libCLHEP-Random-2.1.0.0.so
```

To compile your test file (given below), you can first start with a separate compilation ('*-c' option*) to first avoid the linking stage. The *'-I'* option gives the path where we find the include directory – in this case we suppose that we have the testRand.cpp file place just above the *include* directory. You can use the absolute path (it will be longer to strike).

```
g++ -c testRand.cc –I./include
```

If this phase is ok, you can go further to add the linking stage after compilation. This will be done if we remove the *'-c'* option and also precise the path of the *lib* directory with the *'-L'* option. Then we ask for an executable result with the output *'-o'* option

```
g++ testRand.cc –I./include –L./lib –o myExe
```

This will not be enough, since we have not mentioned that we use the CLHEP library at this linking stage. (like if we had forgotten to precise *'-lm'* for a C program that uses a function of the math library). Depending on the server installation context (changing every year with updates) you have to find the right compilation line that will work. Here are some lines below; the first is in a dynamic link context. Then we go more and more static.

```
(1)   g++ -o myExe testRand.cc –I./include –L./lib -lCLHEP-Random-2.1.0.0
(2)   g++ -o myExe testRand.cc –I./include –L./lib -lCLHEP-Random-2.1.0.0 -static
(3)   g++ -o myExe testRand.cc –I./include –L./lib ./lib/libCLHEP-Random-2.1.0.a
(4)   g++ -o myExe testRand.cc –I./include ./lib/libCLHEP-Random-2.1.0.a
```

When everything works fine, you can go in the `test` directory and compile the `testRandom.cc` file which makes a much wider usage of all objects & methods proposed in the library. On modern C++ compilers, you may have to include `stdlib.h` for the `exit` function. This is achieved by a `#include <cstdlib>` (note that a 'c' is added to the standard name and that the '.h' extension is not mentioned in modern C++ when we include old C libraries. If you are familiar with Unix, you can use the `ldd` Unix command to see the dependencies of an executable – you may see that the LD_LIBRARY_PATH global variable has to be updated to specify the correct PATH and to enable a dynamic linking with the shared object library.

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$PWD/CLHEP/lib
```

When compiled the executable will give you an idea of the utility of this library for Monte Carlo simulations which simulates the major probability.

Compilation idéale des programmes CLHEP:

Attention à la version du g++. Regarder la structure et l'emplacement des fichiers installés et ajuster le chemin des includes – I et celui de la bibliothèque – L. Voici un exemple :

```
CLHEP_DIR=chemin vers le répertoire d'installation

g++   -o testRandom testRandom.cc
      -I$CLHEP_DIR/Random/include
      -L$CLHEP_DIR/Random/lib
      -lCLHEP-Random-2.1.0.0
```

Au besoin ajouter `-static` pour inclure la librairie dans l'exécutable ou avant le lancement du programme compilé ajouter le chemin vers les librairies dans la variable d'environnement LD_LIBRARY_PATH

Super classe

Pour tirer un nombre utiliser la méthode : `flat();`
Pour sauver un statut : `saveStatus(nomDeFichier);`
Pour restaurer le générateur à un statut : `restoreStatus(nomDeFichier);`

Sous classes

La bibliothèque CLHEP propose de nombreux générateurs, selon les connaissances actuelles, 2 générateurs seulement sont corrects : Ranlux64Engine (très lent avec un 'luxury' level de 64, et Mersenne Twister de Matsumoto – non cryptosecure). Voici la liste des générateurs de cette bibliothèque.

```
JamesRandom             RanecuEngine
TripleRand              RanshiEngine
DRand48Engine           Ranlux64Engine
DualRand                RanluxEngine
Hurd160Engine           Hurd288Engine
MTwistEngine            RandEngine
NonRandomEngine
```

Exemple de code CLHEP :

Génération de nombres dans un fichier binaire (pour test avec le logiciel DIE HARD de G. Marsaglia)

```
// Générer des nombres dans un fichier binaire (pour test DIE HARD de G. Marsaglia)

#include <sys/types.h>
#include <sys/stat.h>

#include <fcntl.h>
#include <limits.h>
#include <unistd.h>

#include "CLHEP/Random/MTwistEngine.h"    // Entête pour MT dans la bibliothèque CLHEP

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

   int          fId;
   double       fRn;
   unsigned int iRn;

   // Fichier binaire pour tests Die Hard
   fId = open("./qrngb",O_CREAT|O_TRUNC|O_WRONLY,S_IRUSR|S_IWUSR);

   // Le test de Die Hard est sur 3 millions de nombres
   for(int i = 1; i < 3000000; i++)
   {
      fRn = mtRng->flat(); // Génération uniforme avec MT
      iRn = (unsigned int) (fRn * UINT_MAX);

      write(fId, &iRn, sizeof(unsigned int));
   }

   close(fId);

   delete mtRng;

   return 0;
}
```

Pour information si vous souhaitiez ajouter un générateur : lorsque le résultat multiplicatif intermédiaire dépasse 32 bits, il faudra utiliser des `long long`.

Ceci peut poser un problème à la compilation (ANSI ISO C99), il faut alors au besoin modifier les fichiers `configure.in - Makefile.am` et prendre en compte ces modifications par un bootstrap (`./bootstrap`)

Pour toute prise en compte des modifications, penser à supprimer les bibliothèques avec un `rm -fr lib*` ou plus sagement la commande make adaptée et exécutée dans le bon répertoire. Ensuite vous pouvez relancer le `make -jXX` et `make install`.

Juste pour information SFMT : Saito chercheur et ancien doctorant de Matsumoto a proposé avec lui un générateur avec de meilleures performances notamment une période de 2^{216091}-1, et une vitesse de génération deux fois plus rapide sur la plupart des architectures (2006). Un générateur pour GP-GPU a été proposé en 2009 et plus récemment un Tiny MT avec une période bien plus courte (plus faible empreinte mémoire).

URL : http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/index.html

Appendix – volume of a Sphere – Simple Monte Carlo code

```
13  /** ------------------------------------------------------ *
14    * computeSphereVolume                                    *
15    *                                                        *
16    * @brief  This function computes the volume of a sphere *
17    *         using Monte Carlo simulation (Radius = 1)      *
18    *         Analytically V = 4/3πR³                        *
19    *                                                        *
20    * @param  The number of Points                           *
21    *                                                        *
22    * @return An approximation of the sphere surface         *
23    * ------------------------------------------------------ **/
24
25  double computeSphereVolume(int numOfPoints)
26  {
27      int insideSphere = 0;
28
29      // Check if the points are inside the French
30      for(int i = 0; i < numOfPoints; i++)
31      {
32          double x = random_double();
33          double y = random_double();
34          double z = random_double();
35
36          double distance = sqrt(x * x + y * y + z * z);
37
38          if (distance <= 1.0) insideSphere++;
39      }
40
41      // Volume of the cube = 8 (with side length 2)
42      return (8. * (double) insideSphere / (double) numOfIPoints);
43  }
```