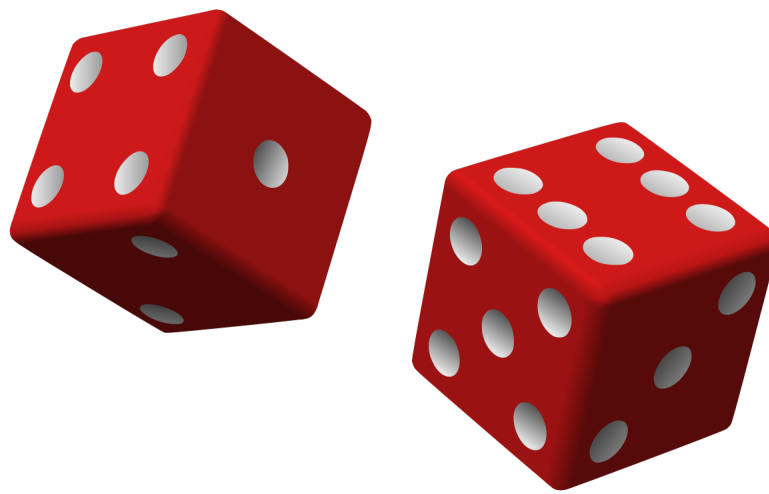


# Compte Rendu Projet C++ Génération de nombres aléatoires

Adrien Deleplace & Paul Dudnic

19 Mai 2020



# TP 1

## Question 1

On exécute un test simple (voir *examples/DemoQ1.cpp* dans le TP1) pour illustrer les différence entre les deux écritures :

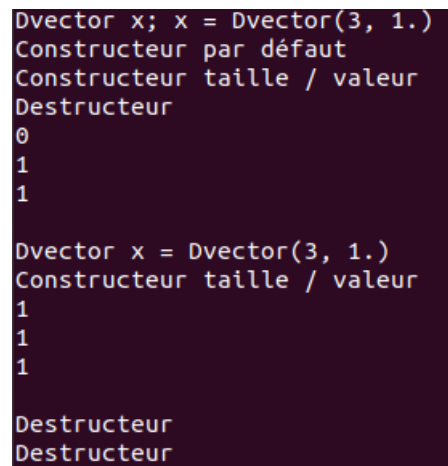
1. 

```
{  
    Dvector x;  
    x = Dvector(3, 1.);  
}
```
2. 

```
{  
    Dvector x = Dvector(3, 1.);  
}
```

## Résultats

On obtient les résultats de la figure 1



```
Dvector x; x = Dvector(3, 1.)  
Constructeur par défaut  
Constructeur taille / valeur  
Destructeur  
0  
1  
1  
  
Dvector x = Dvector(3, 1.)  
Constructeur taille / valeur  
1  
1  
1  
  
Destructeur  
Destructeur
```

FIGURE 1 – Résultat de l'exécution de *examples/DemoQ1.cpp*

## Explications

Étapes de l'affectation (1) (Problématique) :

1. Construction de *x* via le constructeur par défaut
2. Construction d'un objet temporaire (notons le *temp*) via le constructeur taille / valeur
3. Copie de *temp* dans *x* via l'opérateur `=` (non redéfini ici)
4. Destruction de *temp*

Problème :

L'opérateur `=` n'ayant pas été redéfini ici (il le sera dans le TP2), c'est l'opérateur par défaut qui est utilisé. Cet opérateur donne alors lieu à deux affectations :

- `x.m_size`  $\leftarrow$  `temp.m_size` : copie de la taille `temp.m_size`
- `x.m_coords`  $\leftarrow$  `temp.m_coords` : copie de l'adresse mémoire `temp.m_coords`

Lors de la destruction de *temp*, l'espace mémoire alloué à `temp.m_coords` est libéré. Mais cet espace mémoire est toujours pointé par `x.m_coords` : c'est une source de problèmes. Sur la figure 1 on observe que les coordonnées de `x` sont faussées après la 1ère affectation : on tente en effet d'accéder à un espace mémoire libéré  $\implies$  Bug inévitable.

Étapes de l'affectation (2) (Correcte) :

1. Construction de `x` via le constructeur taille / valeur

L'affectation se fait tel que désiré et on peut manipuler `x` sans problèmes par la suite. La figure 1 montre que les coordonnées de `x` sont bien construites.

A la fin de l'exécution du programme, les deux objets créés sont détruits d'où les deux appels au Destructeur.

## Question 2

Voir tests dans le répertoire `/test` du TP1.

### Note sur les tests

Pour ces tests et pour les tests des TPs suivants, nous utilisons le framework de tests unitaires proposé par la bibliothèque BOOST ainsi que la fonctionnalité CTest du logiciel CMake, ce qui permet d'automatiser les tests.

La commande `make test` donne alors un résultat similaire à la figure 2. Les tests réussis sont indiqués comme *Passed*, ceux échoués sont indiqués comme *Failed*. Plus de détails sur les tests, par exemple pour savoir pourquoi un test a échoué, peuvent être obtenus en lançant l'exécutable des tests. Dans cet exemple : `test/TestCopyConstructor`, ce qui produit le résultat de la figure 3.

## Question 3

Voir sources *Dvector.h* et *Dvector.cpp* dans le répertoire `/src` du TP1.

## Question 4

On lance `valgrind` sur un exemple complet manipulant plusieurs Dvectors, on obtient les résultats de la figure 4.

```

Running tests...
Test project /home/paul/Documents/ensimag/c++/tp_cpp/TP1_dudnicp_deleplaa/build
  Start 1: TestCopyConstructor
1/5 Test #1: TestCopyConstructor .....***Failed    0.01 sec
  Start 2: TestDefaultConstructor
2/5 Test #2: TestDefaultConstructor .....    Passed    0.01 sec
  Start 3: TestDisplay
3/5 Test #3: TestDisplay .....    Passed    0.01 sec
  Start 4: TestFileConstructor
4/5 Test #4: TestFileConstructor .....    Passed    0.01 sec
  Start 5: TestSizeValueConstructor
5/5 Test #5: TestSizeValueConstructor .....    Passed    0.00 sec

80% tests passed, 1 tests failed out of 5

Total Test time (real) =  0.05 sec

The following tests FAILED:
      1 - TestCopyConstructor (Failed)
Errors while running CTest
Makefile:103: recipe for target 'test' failed
make: *** [test] Error 8

```

FIGURE 2 – Résultat de la commande `make test`

```

Running 2 test cases...
Constructeur par défaut
Constructeur par copie
Destructeur
Destructeur
Constructeur taille / valeur
Constructeur par copie
/home/paul/Documents/ensimag/c++/tp_cpp/TP1_dudnicp_deleplaa/test/TestCopyConstructor.cpp(26): error: in "vect_copy": check v.size() == 4 has failed [3 != 4]
Destructeur
Destructeur

*** 1 failure is detected in the test module "TestCopyConstructor"

```

FIGURE 3 – Résultat de la commande `test/TestCopyConstructor`

```

==29925==
==29925== HEAP SUMMARY:
==29925==    in use at exit: 0 bytes in 0 blocks
==29925==   total heap usage: 6 allocs, 6 frees, 74,072 bytes allocated
==29925==
==29925== All heap blocks were freed -- no leaks are possible
==29925==
==29925== For counts of detected and suppressed errors, rerun with: -v
==29925== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

```

FIGURE 4 – Exécution de `valgrind` sur `examples/DemoComplete` du TP1

## TP2

### Question 1

Voir tests dans le répertoire */test* du TP2.

### Question 2

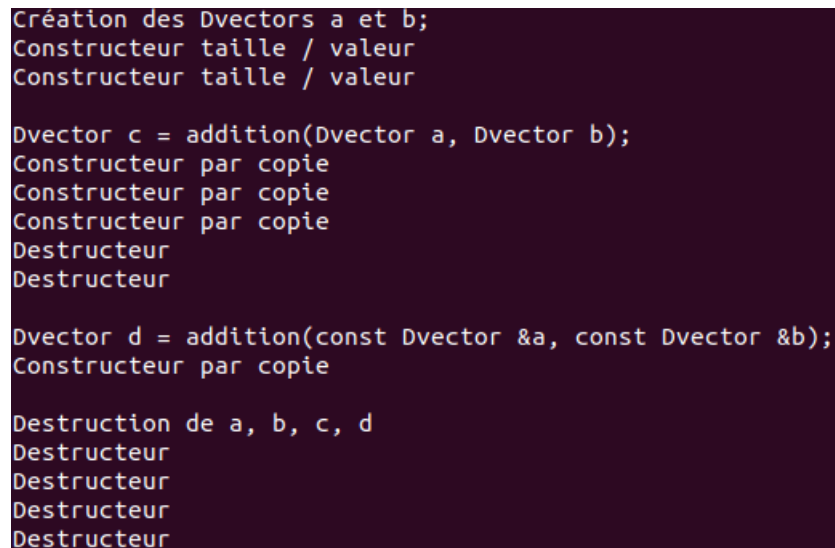
On exécute un test simple (voir *examples/DemoQ2.cpp* dans le TP2) pour illustrer les différences entre les deux écritures :

1. `Dvector addition1(Dvector a, Dvector b);`
2. `Dvector addition2(const Dvector &a, const Dvector &b);`

On crée ici deux fonctions *addition1* et *addition2* qui reproduisent la méthode `operator+` de la classe `Dvector`. Ces deux fonctions sont en tout point identiques, à l'exception de leur signature. Cela permet d'illustrer les différences entre les deux écritures en un seul programme, sans devoir modifier le code de la classe `Dvector`.

### Resultats

On obtient les résultats de la figure 5.



```
Création des Dvectors a et b;
Constructeur taille / valeur
Constructeur taille / valeur

Dvector c = addition(Dvector a, Dvector b);
Constructeur par copie
Constructeur par copie
Constructeur par copie
Destructeur
Destructeur

Dvector d = addition(const Dvector &a, const Dvector &b);
Constructeur par copie

Destruction de a, b, c, d
Destructeur
Destructeur
Destructeur
Destructeur
```

FIGURE 5 – Résultats de l'exécution de *examples/DemoQ2.cpp*

### Explications

Étapes de l'addition (1) (passage d'arguments par copie) :

1. Construction de deux temporaires  $temp_a$  et  $temp_b$  identiques respectivement à  $a$  et  $b$  via le constructeur par copie
2. Construction du Dvector à retourner  $ret$  identique à  $temp_a$  via le constructeur par copie
3. Addition  $ret \leftarrow ret + temp_b$
4. Retour de  $ret$
5. Destruction de  $temp_a$  et  $temp_b$

Lors de l'appel de la fonction, les deux arguments  $a$  et  $b$  sont recopiés dans deux temporaires qui sont détruits à la sortie de la fonction. Cela peut poser des problèmes de performance si les objets sont volumineux car l'ordinateur doit alors créer deux nouveaux objets, ce qui prend du temps et de la mémoire.

Étapes de l'addition (2) (passage d'arguments par référence) :

1. Construction du Dvector à retourner  $ret$  identique à  $temp_a$
2. Addition  $ret \leftarrow ret + temp_b$
3. Retour de  $ret$

Les objets passés en argument de la fonction ne sont pas recopiés, on les manipule via une référence. Cela permet de sauvegarder du temps et de la mémoire.

### Question 3

Voir sources *Dvector.h* et *Dvector.cpp* dans le répertoire */src* du TP2.

### Question 4

Pour factoriser le code on peut utiliser par exemple les opérateurs  $+=$ ,  $-=$ ,  $*=$  et  $/=$  dans l'implémentation respective des opérateurs  $+$ ,  $-$ ,  $*$  et  $/$ .

Voir sources *Dvector.h* et *Dvector.cpp* dans le répertoire */src* du TP2.

### Question 5

On exécute deux fois le même test, une fois en utilisant la fonction `memcpy`, l'autre fois non. Dans ce test, on crée un Dvector vide puis on lui affecte via l'opérateur  $=$  un Dvector de taille 100,000,000 (voir *examples/DemoQ4.cpp* dans le TP2). On obtient les résultats suivants.

```
real    0m0,922s
user    0m0,547s
sys     0m0,375s
```

FIGURE 6 – Temps d'exécution sans `memcpy`

```
real    0m0,756s
user    0m0,428s
sys     0m0,328s
```

FIGURE 7 – Temps d'exécution avec `memcpy`

L'utilisation de `memcpy` améliore les performances de la fonction.

### Question 6

On lance `valgrind` sur un exemple complet manipulant plusieurs Dvectors, on obtient les résultats de la figure 8.

```

==4080==
==4080== HEAP SUMMARY:
==4080==    in use at exit: 0 bytes in 0 blocks
==4080==   total heap usage: 17 allocs, 17 frees, 74,187 bytes allocated
==4080==
==4080== All heap blocks were freed -- no leaks are possible
==4080==
==4080== For counts of detected and suppressed errors, rerun with: -v
==4080== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

```

FIGURE 8 – Exécution de `valgrind` sur *examples/DemoComplete* du TP2

## TP3

### Question 1

Voir fichiers *TestGénérateurParkMiller*, *TestGénérateurXorShift*, *TestDistributionNormale* du répertoire *test/* du TP3. On vérifie tout d’abord que les fonctions déterministes renvoient les résultats attendus. En particulier pour les tests portant sur la classe *DistributionNormale*, on utilise la classe `normal_distribution` de la bibliothèque BOOST comme référence. Dans nos tests, nous assurons une erreur relative inférieure à 0.01% entre nos fonctions `pdf`, `cdf`, `inv_cdf`, et celles de la bibliothèque BOOST.

### Questions 2-3

#### Tests statistiques

Pour évaluer les performances des générateurs Park-Miller et XorShift, nous mesurons tout d’abord que le comportement de ces générateurs est proche de celui d’une variable réellement aléatoire. Pour cela, nous faisons passer 3 types de tests aux deux générateurs : des tests de fréquence, d’auto-corrélation et des *run tests*. Nous nous contenterons ici d’expliquer le principe de chacun de ces tests, sans rentrer dans les détails mathématiques. Plus d’informations sur chacun des tests seront fournies dans les références.

#### Tests de Fréquence [1]

Il s’agit de vérifier que les fréquences des nombres obtenus correspondent bien aux fréquences théoriques d’une distribution donnée. Ici nous voulons vérifier que les générateurs génèrent des nombres entre 0 et *max* (qui varie selon le générateur) de manière uniforme, donc si on note *X* un nombre généré aléatoirement, on veut avoir :

$$\frac{X}{\text{max}} \sim U([0, 1])$$

Pour cela nous effectuons deux tests statistiques, un test de Kolmogorov-Smirnov et un test du  $\chi^2$ , tous deux avec un seuil de confiance  $\alpha = 0.05$ . Les figures 9 et 10 montrent les résultats obtenus pour un échantillon de 1,000,000 de valeurs.

On observe ainsi que le générateur XorShift passe les tests de fréquence mais pas le générateur Park-Miller. Le générateur Park-Miller génère donc des nombres de manière pas tout à fait uniforme.

```

Running 2 test cases...
--- Test de Kolmogorov-Smirnov ---
D obtenu : 0.00145138
D toléré : 0.00136
/home/paul/Documents/ensimag/c++/tp_cpp/TP3_dudnicp_deleplaa/test/TestParkMiller
Frequency.cpp(31): error: in "Kolmogorov_Smirnov_test": check D < expectedD has
failed [0.0014513837921980022 >= 0.0013600000000000001]
--- Test du X2 ---
X2 obtenu : 534.58
X2 toléré : 123.225
/home/paul/Documents/ensimag/c++/tp_cpp/TP3_dudnicp_deleplaa/test/TestParkMiller
Frequency.cpp(50): error: in "chi2_test": check X2 < expectedX2 has failed [534.
579600000000014 >= 123.2252214533618]

*** 2 failures are detected in the test module "TestParkMillerFrequency"

```

FIGURE 9 – Tests de fréquence pour le générateur Park-Miller

```

Running 2 test cases...
--- Test de Kolmogorov-Smirnov ---
D obtenu : 0.00110305
D toléré : 0.00136
--- Test du X2 ---
X2 obtenu : 95.9662
X2 toléré : 123.225

*** No errors detected

```

FIGURE 10 – Tests de fréquence pour le générateur XorShift

### Test d'auto-corrélation [2]

Vérifier que le générateur génère des nombres selon une distribution uniforme ne suffit pas pour déterminer si le générateur a un comportement aléatoire. Prenons un exemple.

40 93 16 48 59 12 99 58 18 42

Les nombres de cette séquence sont distribués de manière uniforme sur  $[0, 100]$  mais on reconnaît un motif : chaque nombre tous les 3 nombres semble avoir une petite valeur, c'est un comportement prévisible donc non aléatoire. En plus de vérifier que les nombres sont générés de manière uniforme, il faut donc aussi vérifier que les nombres générés ne sont pas corrélés entre eux. C'est le but du test d'auto-corrélation. Il s'agit ici de vérifier qu'une certaine variable calculée à partir des valeurs générées par le générateur suit une loi normale centrée réduite avec un seuil de confiance  $\alpha = 0.05$ . Les figures 11 et 12 montrent les résultats obtenus pour un échantillon de 1,000,000 de valeurs :

```

Running 1 test case...
Z obtenu : 0.937813
Z toléré : 1.95996

*** No errors detected

```

FIGURE 11 – Test d'auto-corrélation pour le générateur Park-Miller

```

Running 1 test case...
Z obtenu : 1.86513
Z toléré : 1.95996

*** No errors detected

```

FIGURE 12 – Test d'auto-corrélation pour le générateur XorShift



Les deux générateurs passent le test d'auto-corrélation, mais les valeurs générées par le générateur XorShift semblent plus corrélées que celles générées par le générateur Park-Miller.

### Run tests [3]

La fréquence et l'auto-corrélation ne suffisent pas pour déterminer si un générateur se comporte de manière aléatoire. Regardons les deux séquences suivantes :

```
3 8 14 15 20 38 57 71 80 93
57 96 54 73 61 20 3 19 43 39
```

Ces deux séquences sont susceptibles de passer les deux tests précédents : les valeurs sont distribuées uniformément entre 0 et 100, et il n'y a pas de corrélations entre elles. Pourtant il est peu probable qu'elles aient été générées de manière aléatoire : la première séquence est strictement croissante, tandis que la deuxième a ses 5 premiers termes au dessus de la moyenne et le 5 derniers en dessous. En plus de tester la fréquence et l'auto-corrélation il faut tester les sous-suites - *runs* - de la séquence pour déterminer si elle a été générée de manière aléatoire.

On distingue deux types de sous-suites : les sous suites croissantes / décroissantes, et les sous-suites au dessus / en dessous de la moyenne. Pour chaque type, on vérifie que le nombre et la taille des sous-suites suivent des distributions précises (toujours au seuil de confiance  $\alpha = 0.05$ ). Les figures 13 et 14 montrent les résultats obtenus.

```
Running 4 test cases...
--- Runs "up & down" number test ---
Z obtenu : 2.0958
Z toléré : 1.95996
/home/paul/Documents/ensimag/c++/tp_cpp/TP3_dudnicp_deleplaa/test/TestParkMillerRuns.cpp(34): error: in "runs_up_down_number": check Z < Z_alpha has failed [2.0958014185973979 >= 1.9599639845400547]
--- Runs "above & below mean" number test ---
Z obtenu : 0.277076
Z toléré : 1.95996
--- Runs "up & down" length test ---
X2 obtenu : 33.4591
X2 toléré : 122.108
--- Runs "above & below mean" length test ---
X2 obtenu : 41.2301
X2 toléré : 123.225

*** 1 failure is detected in the test module "TestParkMillerAutocorrelation"
```

FIGURE 13 – Run tests pour le générateur Park-Miller

Ici aussi, le générateur XorShift est plus performant que le générateur Park-Miller qui échoue un des run tests : le nombre de runs croissantes / décroissantes ne semble pas suivre la distribution attendue.

### Conclusion sur les tests statistiques

Pour pousser les tests statistiques plus loin, il aurait été possible d'envisager d'utiliser la batterie de tests *DieHard* [4] proposée par George Marsaglia, qui a aussi imaginé le générateur XorShift [5],

```

Running 4 test cases...
--- Runs "up & down" number test ---
Z obtenu : 0.81745
Z toléré : 1.95996
--- Runs "above & below mean" number test ---
Z obtenu : 0.342882
Z toléré : 1.95996
--- Runs "up & down" length test ---
X2 obtenu : 56.0579
X2 toléré : 122.108
--- Runs "above & below mean" length test ---
X2 obtenu : 61.1053
X2 toléré : 123.225

*** No errors detected

```

FIGURE 14 – Run tests pour le générateur XorShift

réputés pour être difficiles à passer. Néanmoins au vu des résultats statistiques obtenus, le générateur XorShift semble plus efficace pour générer des nombres aléatoires. Cependant d'autres critères peuvent également être pris en compte.

### Périodicité

Il peut être intéressant de regarder la période de chacun des générateurs pour comparer leurs efficacités : un générateur avec une plus grande période sera plus efficace qu'un générateur avec une plus petite période. Une grande période est nécessaire par exemple lorsqu'on veut générer un très grand nombre de données aléatoires, pour ne pas avoir une répétition dans les données. Ici pas besoin de tests statistiques (ce serait d'ailleurs trop long à effectuer), une approche mathématique peut nous apporter le résultat. Pour le générateur Park-Miller, la période dépend de la graine mais est toujours inférieure ou égale au paramètre  $m$  (dans notre cas  $m = 2147483647 = 2^{31} - 1$  du générateur, avec égalité sous certaines conditions précises [6]. Pour le générateur XorShift, une bonne sélection des paramètres  $a_1$ ,  $a_2$  et  $a_3$  assure une période égale à  $2^{64}$  quelle que soit la graine [5].

Ici aussi donc, le générateur XorShift est plus efficace que le générateur Park-Miller.

### Rapidité

On mesure à quelle vitesse le générateur est capable de générer des nombres. Les figures 15 et 16 montrent les résultats obtenus pour la génération de 10,000,000 nombres aléatoires.

```

real    0m0,296s
user    0m0,280s
sys     0m0,016s

```

FIGURE 15 – Temps mis par le générateur Park-Miller

```

real    0m0,124s
user    0m0,112s
sys     0m0,012s

```

FIGURE 16 – Temps mis par le générateur XorShift

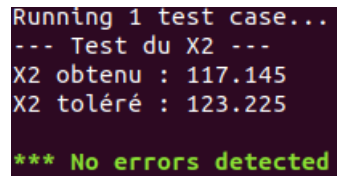
Ici aussi le générateur XorShift est plus efficace.

## Conclusion de la comparaison des deux générateurs

Les résultats obtenus semblent indiquer que le générateur XorShift a un comportement plus proche d'une variable réellement aléatoire que le générateur Park-Miller. Il semble aussi beaucoup plus rapide et a une plus grande période. De manière générale, tout porte à croire que le générateur XorShift est plus efficace que le générateur Park-Miller.

## Notes sur la distribution normale

La classe `DistributionNormale` transforme une distribution uniforme générée par un générateur aléatoire en une distribution normale grâce à la méthode de la transformée de Box-Muller [7]. Pour tester l'efficacité de cette méthode nous nous contenterons ici d'un test du  $\chi^2$  pour vérifier que les fréquences obtenues correspondent aux fréquences attendues d'une distribution normale (les autres caractéristiques étant assurées par les tests sur le générateur aléatoire). Nous ferons les tests sur une distribution normale centrée réduite, puisqu'il est très facile de passer d'une loi normale centrée réduite à une loi normale quelconque (simple addition / multiplication). Nous utiliserons le générateur XorShift pour générer les nombres aléatoires puisque il semble meilleur que le générateur Park-Miller. La figure montre les résultats du tests du  $\chi^2$  effectué.



```
Running 1 test case...
--- Test du X2 ---
X2 obtenu : 117.145
X2 toléré : 123.225
*** No errors detected
```

FIGURE 17 – Résultats du tests du  $\chi^2$  d'adéquation à la loi normale

## Références

- [1] <https://www.eg.bucknell.edu/~xmeng/Course/CS6337/Note/master/node43.html>
- [2] <https://www.eg.bucknell.edu/~xmeng/Course/CS6337/Note/master/node45.html>
- [3] <https://www.eg.bucknell.edu/~xmeng/Course/CS6337/Note/master/node44.html>
- [4] [https://en.wikipedia.org/wiki/Diehard\\_tests](https://en.wikipedia.org/wiki/Diehard_tests)
- [5] <https://www.jstatsoft.org/article/view/v008i14>
- [6] [https://en.wikipedia.org/wiki/Linear\\_congruential\\_generator](https://en.wikipedia.org/wiki/Linear_congruential_generator)
- [7] [https://en.wikipedia.org/wiki/Box\\_Muller\\_transform](https://en.wikipedia.org/wiki/Box_Muller_transform)