Simulating discrete random variables

Florence Perronnin

Univ. Grenoble Alpes, LIG, Inria

October 9, 2018

Outline

Pseudo-random number generator

PRNG generate sequences of deterministic but looking random, hopefully uniformly distributed numbers.

- Several types of PRNG exist : LCG, Mersenne-twister, L'Ecuyer, Marsaglia, . . .
- Their output usually pass some randomness tests, but not always
- You need to know:
  - which one you are using (too often the default value!) `RNGkind()`
  - its upsides and downsides
  - why you chose that particular PRNG (and whether suits the application)

class
std::default_random_engine

Default random engine

This is a random number engine class that generates pseudo-random numbers.

It is the library implemention's selection of a generator that provides at least acce
casual, inexpert, and/or lightweight use.

Member types

The following alias is a member type of `default_random_engine`:

member type	definition	notes
`result_type`	An unsigned integer type	The type of the numbers generated.

The currently available RNG kinds are given below. `kind` is partially matched to this list. The default is `"M`

`"Wichmann-Hill"`

The seed, `.Random.seed[-1] == r[1:3]` is an integer vector of length 3, where each `r[i]` is in `1:(`
primes, `p = (30269, 30307, 30323)`. The Wichmann–Hill generator has a cycle length of *6.9536e*
(1984) **33**, 123 which corrects the original article).

`"Marsaglia-Multicarry"`:

A *multiply-with-carry* RNG is used, as recommended by George Marsaglia in his post to the mailing
than *2^60* and has passed all tests (according to Marsaglia). The seed is two integers (all values allov

`"Super-Duper"`:

Marsaglia's famous Super-Duper from the 70's. This is the original version which does *not* pass the M
period of *about 4.6*10^18* for most initial seeds. The seed is two integers (all values allowed for the

We use the implementation by Reeds *et al* (1982–84).

The two seeds are the Tausworthe and congruence long integers, respectively. A one-to-one mapping
will not publish one, not least as this generator is **not** exactly the same as that in recent versions of S

`"Mersenne-Twister"`:

From Matsumoto and Nishimura (1998). A twisted GFSR with period *2^19937 - 1* and equidistributi
whole period). The 'seed' is a 624-dimensional set of 32-bit integers plus a current position in that se

`"Knuth-TAOCP-2002"`:

Shortcomings of a PRNG

- Limited period (max. cycle length), possibly depending on chosen seed
- lack of uniformity
- correlation of successive values
- `http://www.pcg-random.org/statistical-tests.html`
- The object `.Random.seed` is only looked for in the user's workspace.

  Do not rely on randomness of low-order bits from RNGs. Most of the supplied uniform generators return 32-bit integer values that are converted to doubles, so they take at most *2^32* distinct values and long runs will return duplicated values (Wichmann-Hill is the exception, and all give at least 30 varyin bits.)

  Author(s)

  of RNGkind: Martin Maechler. Current implementation, B. D. Ripley

Controlling randomness

State of a PRNG

`set.seed()` enables to control the seed of a PRNG :

- set a deterministic seed for reproducibility, bug fixing...
- pick a “random” seed for production

Sometimes one needs to record the state of a PRNG.
`.Random.seed` in `R`, `random.getstate()` in Python,. . .

Outline

Simulation of random variables

Problem

Suppose we have a good pseudo-random number generator. Then its output X_n is a sequence of (let's say discrete) uniform random variates in some interval $[a, b]$.
Now, how are we supposed to simulate (=generate) a non-uniform random variable Y with some (known) distribution p_Y?
Or a uniform r.v. Z over a complicated state space?

random variable (r.v.)

= probabilistic object.

random variate

= simulated output.

Examples

- Design a one-armed bandit
- Simulate product defects (failures)
- Simulate consumer demand
- Generate "typical" states of an automaton
- Generate textures and 3D models in video games
- Simulate the path of a hurricane
- and so on...

Typically these output are random but non-uniform.

Examples

- **R** `rbinom,rgeom, rnorm ...`
- **Python** `random.normalvariate(mu, sigma)`, `random.expovariate(lambd)...`
- **Matlab** `randn`
- **C++** `std::geometric_distribution,...`

Transforming random variates

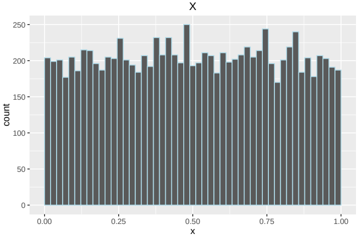
transformation algorithm A

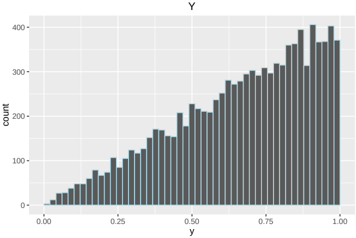
PRNG output $\{X_k\}_{k\geq 0} \stackrel{A}{\longmapsto} \{Y_j\}_{j\geq 0}$ where:

1. $X \sim \mathcal{U}([0,1])$ output of the PRNG
2. $Y \sim F_Y$ simulated variate

Example:

```
N=10000
x=runif(N)
y=sqrt(x)
```





Validation of the simulated r.v.

- Validity of the transformation: Prove that if the pseudo-random numbers $\{X_k\}$ are (really) uniform, then $Y_j \sim F_Y$
- Discrete input robustness : PRNG not uniform over $[0, 1]$ but over a large set of rationals such as $\left\{\frac{0}{K}, \frac{1}{K}, \dots, \frac{K}{K}\right\}$
- Validity of the transform when taking the specific PRNG properties into account (e.g., low-order bits)
- Empirical validation : statistical tests validating that the outputs $Y_1, \dots, Y_n$ are i.i.d. with law F_Y.

Complexity

In addition to the validity of the transform, the quality of a simulation algorithm depends on:

Time complexity

The number of basic operations needed to generate 1 random variate.

Space complexity

The memory space needed for generating an r.v.

Outline

From $[0,1]$ to integer

Suppose you have a random generator `random()` providing uniform samples U_k in $[0,1]$. But you need a uniform integer in $\{1,2,\ldots,N\}$.

$\lfloor \texttt{random()} \times N + 1 \rfloor$

Proof. Let V be the output of the algorithm and U the result of `random()`. Then for any k in $\{1,2,\ldots,N\}$:

$$\begin{aligned}
\mathbb{P}[V=k] &= \mathbb{P}[\lfloor \texttt{random()} \times N + 1 \rfloor = k] \\
&= \mathbb{P}[\lfloor U \times N + 1 \rfloor = k] \\
&= \mathbb{P}[k \le U \times N + 1 \le k+1] \\
&= \mathbb{P}\left[\frac{k}{N} \le U \le \frac{k+1}{N}\right] \\
&= \frac{k+1}{N} - \frac{k}{N} \qquad \text{(uniformity on } [0,1]) \\
&= \frac{1}{N} \qquad \text{(uniform on } \{1,2,\ldots,N\}).
\end{aligned}$$

Première approche du rejet

Dé-8 $\mapsto$ Dé-6

Jojo possède un dé à 8 faces, mais pour jouer avec Dédé, lui faudrait un dé à 6 faces. Peuvent-ils jouer quand même?

Dé-6()

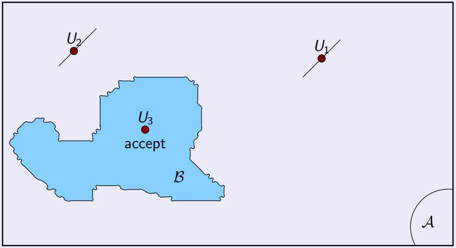
Données: Une fonction **Dé-8()** générateur aléatoire de $\{1, \cdots, 8\}$
Résultat: Une séquence i.i.d. de loi uniforme sur $\{1, \cdots, 6\}$

repeat
| $X =$**Dé-8()**
until $X \leq 6$
return X

Méthode du Rejet

Principe

1. Générer uniformément sur $\mathcal{A}$
2. Accepter si le point est dans $\mathcal{B}$.



Algorithme

Génère-unif($\mathcal{B}$)

Données:
Générateur uniforme sur $\mathcal{A}$
Résultat:
Générateur uniforme sur $\mathcal{B}$

repeat
| X =**Génère-unif($\mathcal{A}$)**
until $X \in \mathcal{B}$
return X

Preuve

Génère-unif($\mathcal{B}$)

Données:
Générateur uniforme sur $\mathcal{A}$
Résultat:
Générateur uniforme sur $\mathcal{B}$

$N = 0$
repeat
  $X =$**Génère-unif($\mathcal{A}$)**
  $N = N + 1$
until $X \in \mathcal{B}$
return X, N

Tirages **Génère-unif($\mathcal{A}$)**:
$X_1, X_2, \cdots, X_n, \cdots$

Soit $\mathcal{C} \subset \mathcal{B}$ une partie de $\mathcal{B}$. Montrons que $\mathbb{P}[X \in C] = \frac{|\mathcal{C}|}{|\mathcal{B}|}$ (loi uniforme sur $\mathcal{B}$).

$$\begin{aligned}
& \mathbb{P}(X \in \mathcal{C}, N = k) \\
= \ & \mathbb{P}(X_1 \notin \mathcal{B}, \cdots, X_{k-1} \notin \mathcal{B}, X_k \in \mathcal{C}) \\
= \ & \mathbb{P}(X_1 \notin \mathcal{B}) \cdots \mathbb{P}(X_{k-1} \notin \mathcal{B}) \mathbb{P}(X_k \in \mathcal{C}) \\
= \ & \left(1 - \frac{|\mathcal{B}|}{|\mathcal{A}|}\right)^{k-1} \frac{|\mathcal{C}|}{|\mathcal{A}|}
\end{aligned}$$

$$\begin{aligned}
\mathbb{P}(X \in \mathcal{C}) &= \sum_{k=1}^{+\infty} \mathbb{P}(X \in \mathcal{C}, N = k) \\
&= \sum_{k=1}^{+\infty} \left(1 - \frac{|\mathcal{B}|}{|\mathcal{A}|}\right)^{k-1} \frac{|\mathcal{C}|}{|\mathcal{A}|} = \frac{|\mathcal{C}|}{|\mathcal{B}|}
\end{aligned}$$

Donc la loi est **uniforme** sur $\mathcal{B}$

Complexité (coût)

Génère-unif($\mathcal{B}$)

Données:
Générateur uniforme sur $\mathcal{A}$
Résultat:
Générateur uniforme sur $\mathcal{B}$

```
N = 0
repeat
    X = Génère-unif(A)
    N = N + 1
until X ∈ B
return X, N
```

N Nombre d'itérations (variable aléatoire)

$$\begin{aligned} \mathbb{P}(N=k) &= \mathbb{P}(X \in \mathcal{B}, N=k) \\ &= \left(1 - \frac{|\mathcal{B}|}{|\mathcal{A}|}\right)^{k-1} \frac{|\mathcal{B}|}{|\mathcal{A}|} \end{aligned}$$

Loi géométrique de paramètre $p_a = \frac{|\mathcal{B}|}{|\mathcal{A}|}$.
Nombre moyen d'itérations :

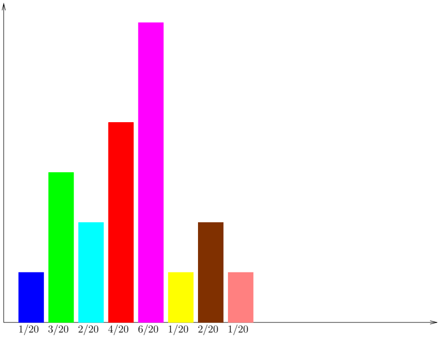
$$\begin{aligned} \mathbb{E}[N] &= \sum_{k=1}^{+\infty} k(1-p_a)^{k-1} p_a \\ &= \frac{1}{(1-(1-p_a))^2} p_a = \frac{1}{p_a}. \end{aligned}$$

p_a est la probabilité d'acceptation

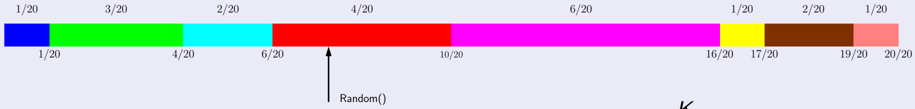
Outline

Loi sur un ensemble fini

- K valeurs
- Générer une couleur aléatoire selon la distribution ci-contre

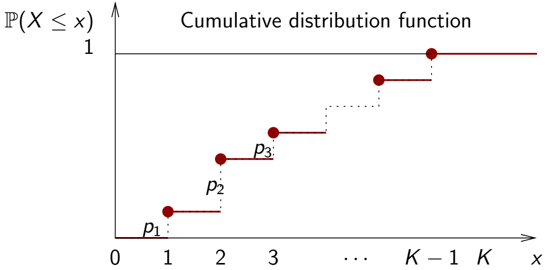


Histogramme : représentation "à plat"



Coût (nombre moyen de comparaisons) : $\hat{C}(P) = \sum_{k=1}^{K} k.p_k = 4.35$

Inverse de la fonction de répartition



Principe

1. Diviser $[0, 1[$ en intervalles de longueur p_k
2. Générer un nombre U uniforme sur $[0, 1]$: `U=runif(1)`
3. Trouver l'intervalle contenant U
4. Retourner l'index de l'intervalle

Coût de calcul : $\mathcal{O}(\mathbb{E}[X])$ itérations
Coût mémoire : $\mathcal{O}(K)$

Inverse de la fonction de répartition: algorithme

Algorithme

Inverse(P[ ])

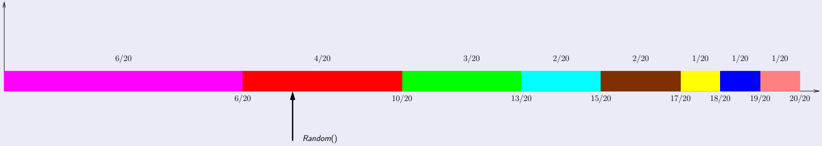
Données: Un tableau de probabilités $P[] = \{p_1, \cdots, p_K\}$
Résultat: Un entier k généré avec la probabilité p_k

$u =$ **Random()**
$k = 0$
$S = 0$
while $u > S$
  $k = k + 1$
  $S = S + P[k]$
return k

Optimisation

Méthodes d'optimisation

- pré-calcul de la fonction de répartition dans une table
- ranger les objets par probabilité décroissante $\Rightarrow$ Coût=3.1

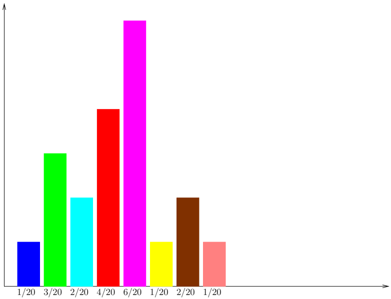


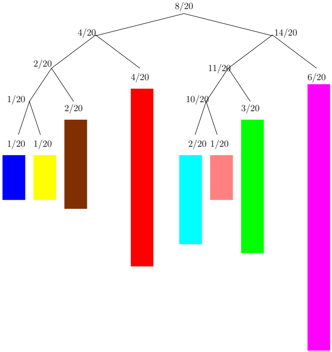
- utiliser une recherche dichotomique
- utiliser un arbre binaire de recherche (optimalité = Huffmann coding tree)

Commentaires

- Dépend de l'usage du générateur (répétitions)
- pré-calcul en $\mathcal{O}(K)$ (peut être grand)

Optimalité



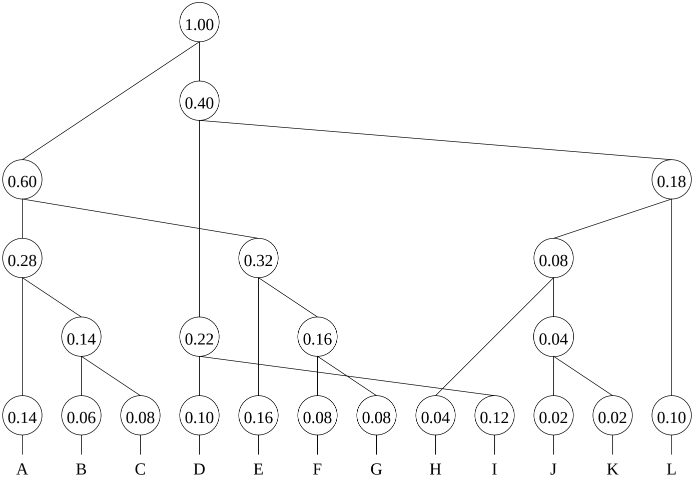


Nombre de comparaisons

Structure d'arbre binaire de recherche

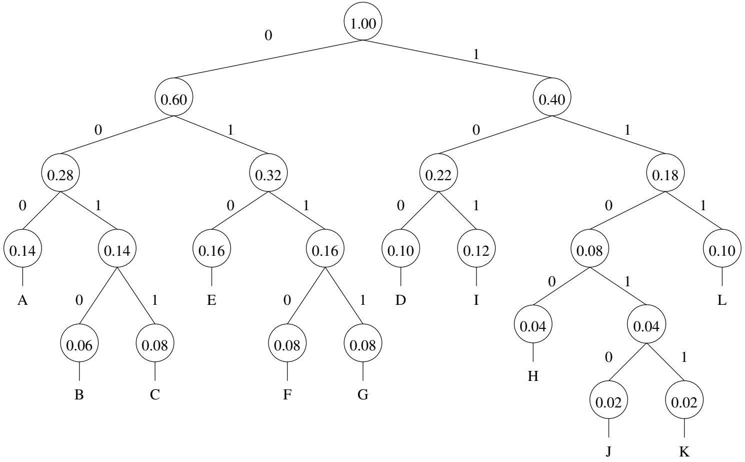
$$\mathbb{E}[N] = \sum_{k=1}^{K} p_k . l_k = 2.75, \qquad \text{Entropie} = \sum_{k=1}^{K} p_k(-\log_2 p_k) = 2.70$$

Algorithme de Huffman (1951)



A	0.1
B	0.0
C	0.0
D	0.1
E	0.1
F	0.0
G	0.0
H	0.0
I	0.1
J	0.0
K	0.0
L	0.1

Algorithme de Huffman (1951)



A	0.14	000
B	0.06	001
C	0.08	001
D	0.10	100
E	0.16	010
F	0.08	011
G	0.08	011
H	0.04	110
I	0.12	101
J	0.02	110
K	0.02	110
L	0.10	111

Codage optimal : L-moy = 3.42, Entropie = 3.38

Profondeur = $-\log_2$(probabilité)

Algorithme de Huffman (1951): Implantation

Arbre-Huffman(P[])

Données: Un tableau de probabilité P=$\{p_1, \cdots, p_K\}$
Résultat: Un arbre binaire de Huffman transformé en arbre binaire de recherche

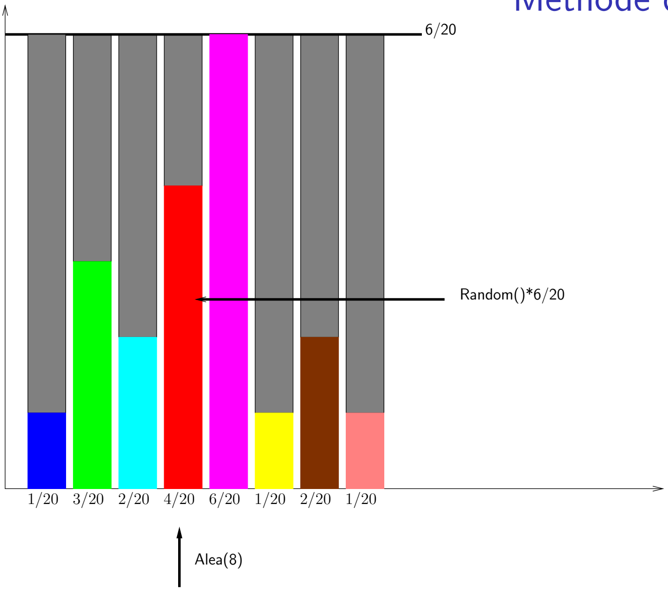
```
F: file à priorité
for k = 1 to K
    z=nouveau_noeud()
    z.gauche=Nil z.droit=Nil
    z.poids=P[k] Insérer(F,z)
while Taille(F)≠1
    z=nouveau_noeud()
    z.gauche=Extraire(F) z.droit=Extraire(F)
    z.poids=z.gauche.poids+z.droit.poids Insérer(F,z)
z=Extraire(F)
Mettre_à_jour_étiquettes(z)                    // parcours infixé
```

Infinite support

Question

How can we use the inverse CDF technique for random variables with an infinite number of possible values?

Méthode du rejet



Méthode du rejet (suite)

Génère(P[])

Données: Un tableau de probabilités $P[] = \{p_1, \cdots, p_K\}$
Résultat: Un entier k généré avec la probabilité p_k

$N = 0$
repeat
  $k =$ Partie entière$(Random() * K + 1)$
  $u = Random() * p_{max}$
  $N = N + 1$
until $u \leq P[k]$
return k, N

Preuve

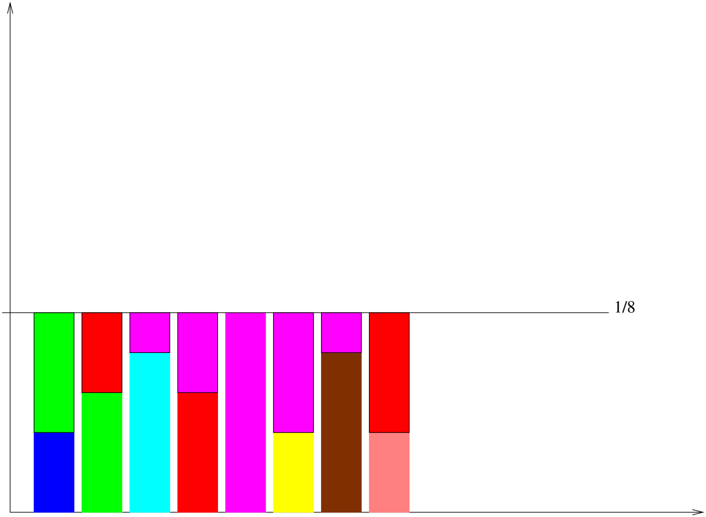
Même preuve que pour la loi uniforme

Coût moyen (nb d'itérations) :

$$p_a = \frac{1}{K.p_{max}} \text{ et } \mathbb{E}[N] = Kp_{max}$$

Outline

Méthode d'aliasing [Walker]



Méthode d'aliasing : construction des tables

Table_Alias(P[])

Données: Un tableau de probabilité $P = [p_1, \cdots, p_K]$
Résultat: Un tableau de seuils $S = [s_1, \cdots, s_K]$
et d'alias $A = [a_1, \cdots, a_K]$

$L = \emptyset \; U = \emptyset$
for $k = 1$ **to** K
    switch $P[k]$ **do**
        case $< \frac{1}{K}$ $L = L \cup \{k\}$
        case $> \frac{1}{K}$ $U = U \cup \{k\}$

while $L \neq \emptyset$
    $i = Extract(L) \; k = Extract(U)$
    $S[i] = P[i] \; A[i] = k$
    $P[k] = P[k] - (\frac{1}{K} - P[i])$
    switch $P[k]$ **do**
        case $< \frac{1}{K}$ $L = L \cup \{k\}$
        case $> \frac{1}{K}$ $U = U \cup \{k\}$

Méthode d'aliasing : génération

Génère(S[],A[])

Données: Un tableau de seuils $S = [s_1, \cdots, s_K]$
et d'alias $A = [a_1, \cdots, a_K]$

Résultat: Un indice k généré selon la probabilité $P = [p_1, \cdots, p_K]$

k = **Alea**(K) `// générateur uniforme d'entiers de 1 à` K

if R$andom()\frac{1}{K} < S[k]$
  return k

else
  return $A[k]$

Méthode d'aliasing : Complexité

Temps de calcul :

- $\mathcal{O}(K)$ pour le pré-calcul des tables d'alias
- $\mathcal{O}(1)$ pour la génération

Coût Mémoire:

- seuils $\mathcal{O}(K)$ (même coût que le vecteur P)
- alias $\mathcal{O}(K)$ (tableau d'index)

Outline

Binomial distribution $X \sim \mathcal{B}(n, p)$

Distribution

$$\mathbb{P}[X = k] = \binom{n}{k} p^n (1 - p)^{n-k}$$

For small values of n

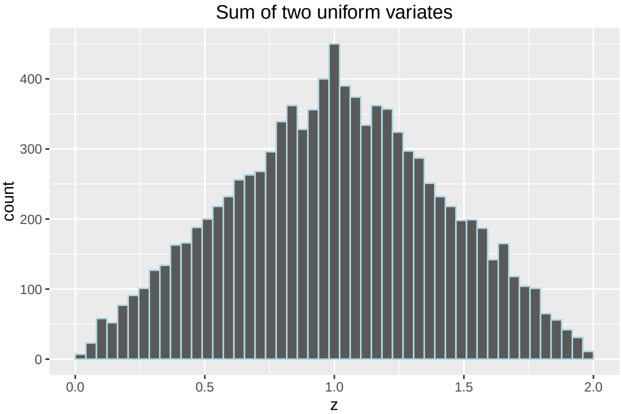
X models the sum of n independent Bernoulli trials.

For larger n

Recursive methods [Berard]

Triangular distribution

Let $X \sim \mathcal{U}([0,1])$ and $Y \sim \mathcal{U}([0,1])$.
Then $Z = X + Y$ has triangular distribution:



Outline

Methods for generating discrete random variables

Generic methods

- Inverse of CDF : can be pre-computed for finite r.v. at the extra-cost of a table
- Rejection method : complexity depends on rejection probability
- Walker (alias) method : faster but requires pre-processing and alias table.

Specialized methods

- exploit intrinsic structure of probability laws

Caveats

- Validity of the transformation
- Time complexity (number of operations)
- Memory overhead

Sources

- Jean-Marc Vincent, Random generation of discrete random variables, Notes de cours, 2010.
- Jean-Marc Vincent, Générateurs de loi uniforme et de lois discrètes, Notes de cours, 2016.
- Jean Bérard, Génération de variables pseudo-aléatoires, Notes de cours, Université Claude Bernard - Lyon 1.
- Luc Devroye, Non-uniform Random Variate Generation, http://www.eirene.de/Devroye.pdf
- Pierre L'Ecuyer, Random number generation, Handbook of Computational Statistics. Springer, Berlin, Heidelberg, pp. 35-71, 2012.
- Alastair Walker, An efficient method for generating discrete random variables with general distributions, ACM Transactions on Mathematical Software (TOMS) 3: 253-256, 1977.