# RjpWiki アーカイブス

## 【確率分布，乱数関数一覧 (06.06.28) 】

R は確率分布に関する豊富な関数群を持ち，従来統計学の利用で不可欠であった各種確率分布に関する数表は，よほど特殊なものを除き，もはや必要が無い．R では例えば正規分布は norm という名前を持ち， dnorm はその密度関数， pnorm はその分布関数， qnorm はそのクォンタイル関数，更に rnorm は正規 (疑似) 乱数の生成関数を表す，等の一貫した記法を使う．

# 1 疑似乱数

## 1.1 疑似乱数発生関連関数

.Random.seed は整数ベクトルで，R の乱数生成機構に対する乱数生成器 (RNG) の状態を含んでいる．これは保存し，元に戻すことができるが，ユーザが変更すべきではない．RNGkind() は使用中の RNG の種類を確認したり，設定したりするためのより分かりやすいインタフェイスである．RNGversion() は以前のバージョンの R に於けるように乱数発生器を設定するために使うことができる (再現性のために設けられている)．set.seed() は乱数種を指定するための推薦できる方法である．

```
.Random.seed <- c(rng.kind, n1, n2, ...)
save.seed <- .Random.seed
RNGkind(kind = NULL, normal.kind = NULL)
RNGversion(vstr)
set.seed(seed, kind = NULL)
```

## 1.2 ユーザ定義の疑似乱数発生器

関数 RNGkind() はユーザがコードした一様疑似乱数と正規疑似乱数生成器の使用を許す．以下にその詳細を述べる．

ユーザー指定の一様 RNG は動的にロードされたコンパイル済みコードのエントリ点から呼び出される．ユーザはエントリ点 user unif rand を提供する必要があり，これは引数はなく，倍精度実数へのポインタを返す．以下の例は一般的なパターンを示す．

オプションとして，ユーザは RNGkind (もしくは set.seed) が呼び出されたとき引数 unsigned int で呼び出されるエントリ点 user unif init を提供することができ，ユーザの RNG コードを初期化するために使うことが想定されている．2 個の引数は「乱数種」を設定するのに使われることを想定している．それは set.seed への seed 引数であるか，もし RNGkind が呼び出されるならば本質的にランダムな乱数種である．

もしこれらの関数だけが提供されると，生成器に対する如何なる情報も .Random.seed に記録されない．オプションとして，引数無しに呼び出されると，乱数種の数と乱数種の整数配列へのポインタを返す関数 user unif nseed と user unif seedloc を与えることができる．GetRNGstate と PutRNGstate への呼び出しは，この配列を.Random.seed へ，そして .Random.seed からコピーする．ユーザ定義の正規 RNG は単一のエントリ点 user norm rand で指定され，これは引数を持たず，倍精度実数へのポインタを返す．

# 2　連続分布

## 2.1　一様分布

これらの関数は区間 min から max 上の一様分布に関する情報を与える．dunif() は密度関数，punif() は分布関数，qunif() はクォンタイル関数，そして runif() は乱数を与える．

```
dunif(x, min=0, max=1, log = FALSE)
punif(q, min=0, max=1, lower.tail = TRUE, log.p = FALSE)
qunif(p, min=0, max=1, lower.tail = TRUE, log.p = FALSE)
runif(n, min=0, max=1)
```

## 2.2　正規分布

dnorm() は平均 mean，標準偏差 sd の正規分布の密度関数，pnorm() は分布関数，qnorm() はクォンタイル関数，そして rnorm() は乱数を与える．

```
dnorm(x, mean=0, sd=1, log = FALSE)
pnorm(q, mean=0, sd=1, lower.tail = TRUE, log.p = FALSE)
qnorm(p, mean=0, sd=1, lower.tail = TRUE, log.p = FALSE)
rnorm(n, mean=0, sd=1)
```

## 2.3　対数正規分布

dlnorm は対数正規分布の密度関数，plnorm は分布関数，qlnorm はクォンタイル関数，そして rlnorm は乱数を与える．対数値の平均は meanlog，標準偏差は sdlog となる．

```
dlnorm(x, meanlog = 0, sdlog = 1, log = FALSE)
plnorm(q, meanlog = 0, sdlog = 1, lower.tail = TRUE, log.p = FALSE)
qlnorm(p, meanlog = 0, sdlog = 1, lower.tail = TRUE, log.p = FALSE)
rlnorm(n, meanlog = 0, sdlog = 1)
```

## 2.4　ガンマ分布

dgamma() はパラメータ shape と scale のガンマ分布の密度関数，pgamma() は分布関数，qgamma() はクォンタイル関数，そして rgamma() は乱数を与える．

```
dgamma(x, shape, rate = 1, scale = 1/rate, log = FALSE)
pgamma(q, shape, rate = 1, scale = 1/rate,
       lower.tail = TRUE, log.p = FALSE)
qgamma(p, shape, rate = 1, scale = 1/rate,
       lower.tail = TRUE, log.p = FALSE)
rgamma(n, shape, rate = 1, scale = 1/rate)
```

## 2.5　ベータ分布

dbeta はパラメータ shape1, shape2 (そしてオプションの非心度パラメータ ncp) を持つベータ分布の密度関数，pbeta は分布関数，qbeta はクォンタイル関数，そして rbeta は乱数を与える．

```
dbeta(x, shape1, shape2, ncp=0, log = FALSE)
pbeta(q, shape1, shape2, ncp=0, lower.tail = TRUE, log.p = FALSE)
qbeta(p, shape1, shape2,        lower.tail = TRUE, log.p = FALSE)
rbeta(n, shape1, shape2)
```

## 2.6　カイ自乗分布

dchisq() は自由度 df，そしてオプションの非心パラメータ ncf を持つカイ自乗分布の密度関数，pchisq() は分布関数，qchisq() はクォンタイル関数，そして rchisq() は乱数を与える．

```
dchisq(x, df, ncp=0, log = FALSE)
pchisq(q, df, ncp=0, lower.tail = TRUE, log.p = FALSE)
qchisq(p, df, ncp=0, lower.tail = TRUE, log.p = FALSE)
rchisq(n, df, ncp=0)
```

## 2.7　t 分布

dt() は自由度 df (そして非心度パラメータ ncp の) t 分布の密度関数，pt() は分布関数，qt() はクォンタイル関数，そして rt() は乱数を与える．

```
dt(x, df, ncp=0, log = FALSE)
pt(q, df, ncp=0, lower.tail = TRUE, log.p = FALSE)
qt(p, df,        lower.tail = TRUE, log.p = FALSE)
rt(n, df)
```

## 2.8 F 分布

df() は自由度 df1, df2 (そしてオプションの非心度パラメータ ncp) を持つ F 分布の密度関数，pf() は分布関数，qf() はクォンタイル関数，そして rf() は乱数を与える．

```
df(x, df1, df2, log = FALSE)
pf(q, df1, df2, ncp=0, lower.tail = TRUE, log.p = FALSE)
qf(p, df1, df2,        lower.tail = TRUE, log.p = FALSE)
rf(n, df1, df2)
```

## 2.9 コーシー分布

dcauchy() は位置パラメータ location，スケールパラメータ scale のコーシー分布の密度関数，pcauchy() は分布関数，qcauchy() はクォンタイル関数，そして rcauchy() は乱数を与える．

```
dcauchy(x, location = 0, scale = 1, log = FALSE)
pcauchy(q, location = 0, scale = 1, lower.tail = TRUE, log.p = FALSE)
qcauchy(p, location = 0, scale = 1, lower.tail = TRUE, log.p = FALSE)
rcauchy(n, location = 0, scale = 1)
```

## 2.10 指数分布

dexp() は割合 rate (つまり平均が 1/rate) の指数分布の密度関数，pexp() は分布関数，qexp() はクォンタイル関数，そして rexp() は乱数を与える．

```
dexp(x, rate = 1, log = FALSE)
pexp(q, rate = 1, lower.tail = TRUE, log.p = FALSE)
qexp(p, rate = 1, lower.tail = TRUE, log.p = FALSE)
rexp(n, rate = 1)
```

## 2.11 ロジスティック分布

dlogis() は位置パラメータ location とスケールパラメータ scale のロジスティック分布の密度関数，plogis() は分布関数，qlogis() はクォンタイル関数，そして rlogis() は乱数を与える．

```
dlogis(x, location = 0, scale = 1, log = FALSE)
plogis(q, location = 0, scale = 1, lower.tail = TRUE, log.p = FALSE)
qlogis(p, location = 0, scale = 1, lower.tail = TRUE, log.p = FALSE)
rlogis(n, location = 0, scale = 1)
```

### 2.12 ワイブル分布

dweibull() はパラメータ shape と scale のワイブル分布の密度関数，pweibull() は分布関数，qweibull() はクォンタイル関数，そして rweibull() は乱数を与える．

```
dweibull(x, shape, scale = 1, log = FALSE)
pweibull(q, shape, scale = 1, lower.tail = TRUE, log.p = FALSE)
qweibull(p, shape, scale = 1, lower.tail = TRUE, log.p = FALSE)
rweibull(n, shape, scale = 1)
```

### 2.13 スチューデント化範囲 (Studentized range) 分布

スチューデント化範囲 (Studentized range) 分布とは，$R$ を $n$ 個の標準正規標本の範囲，$s^2$ をそれと独立な自由度 $df$ のカイ自乗分布に従う確率変数とすると，$R/s$ の確率分布である．

```
ptukey(q, nmeans, df, nranges = 1, lower.tail = TRUE, log.p = FALSE)
qtukey(p, nmeans, df, nranges = 1, lower.tail = TRUE, log.p = FALSE)
```

## 3 離散分布

### 3.1 無作為抽出とランダムな置換

sample() は x の要素から指定したサイズの標本を復元，もしくは非復元抽出する．

```
sample(x, size, replace = FALSE, prob = NULL)
```

### 3.2 二項分布

dbinom はパラメータ size と prob の二項分布の確率関数，pbinom() は分布関数，qbinom() はクォンタイル関数，そして rbinom() は乱数を与える．

```
dbinom(x, size, prob, log = FALSE)
pbinom(q, size, prob, lower.tail = TRUE, log.p = FALSE)
qbinom(p, size, prob, lower.tail = TRUE, log.p = FALSE)
rbinom(n, size, prob)
```

## 3.3 負の二項分布

dnbinom() はパラメータ size と prob の負の二項分布の密度関数，pnbinom() は分布関数，qnbinom() はクォンタイル関数，そして rnbinom() は乱数を与える．

```
dnbinom(x, size, prob, mu, log = FALSE)
pnbinom(q, size, prob, mu, lower.tail = TRUE, log.p = FALSE)
qnbinom(p, size, prob, mu, lower.tail = TRUE, log.p = FALSE)
rnbinom(n, size, prob, mu)
```

## 3.4 ポアッソン分布

dpois89 はパラメータ lambda のポアッソン分布の (対数) 確率関数，ppois() は (対数) 分布関数，qpois() はクォンタイル関数，そして rpois() は乱数を計算する．

```
dpois(x, lambda, log = FALSE)
ppois(q, lambda, lower.tail = TRUE, log.p = FALSE)
qpois(p, lambda, lower.tail = TRUE, log.p = FALSE)
rpois(n, lambda)
```

## 3.5 幾何分布

dgeom() はパラメータ prob の幾何分布の密度関数，pgeom() は分布関数，qgeom() はクォンタイル関数，そして rgeom() は乱数を与える．

```
dgeom(x, prob, log = FALSE)
pgeom(q, prob, lower.tail = TRUE, log.p = FALSE)
qgeom(p, prob, lower.tail = TRUE, log.p = FALSE)
rgeom(n, prob)
```

## 3.6 超幾何分布

dhyper() は超幾何分布の密度関数，phyper() は分布関数，qhyper() はクォンタイル関数，そして rhyper() は乱数を与える．

```
dhyper(x, m, n, k, log = FALSE)
phyper(q, m, n, k, lower.tail = TRUE, log.p = FALSE)
qhyper(p, m, n, k, lower.tail = TRUE, log.p = FALSE)
rhyper(nn, m, n, k)
```

### 3.7 多項分布

多項分布に従う乱数ベクトルを発生し，確率関数を計算する．

```
rmultinom(n, size, prob)
dmultinom(x, size = NULL, prob, log = FALSE)
```

### 3.8 Wilcoxon のランク和統計量分布

dwilcox() はサイズがそれぞれ m, n の標本から得られた Wilcoxon のランク和統計量の密度関数，pwilcox() は分布関数，qwilcox() はクォンタイル関数，そして rwilcox() は乱数を与える．

```
dwilcox(x, m, n, log = FALSE)
pwilcox(q, m, n, lower.tail = TRUE, log.p = FALSE)
qwilcox(p, m, n, lower.tail = TRUE, log.p = FALSE)
rwilcox(nn, m, n)
```

### 3.9 Wilcoxon 符号付きランク統計量分布

サイズ n の標本に対する Wilcoxon 符号付きランク統計量の分布関数に関する情報を得る．dsignrank() は確率関数，psignrank() は分布関数，qsignrank() はクォンタイル関数，そして rsignrank() は乱数を与える．

```
dsignrank(x, n, log = FALSE)
psignrank(q, n, lower.tail = TRUE, log.p = FALSE)
qsignrank(p, n, lower.tail = TRUE, log.p = FALSE)
rsignrank(nn, n)
```

# 4 その他

## 4.1 ランダムな 2 次元配列

Patefield のアルゴリズムを用い周辺和を与えたランダムな 2 次元配列（2 次元分割表）を生成する．

```
r2dtable(n, r, c)
```

## 4.2 一致確率

一般化された「誕生日のパラドックス」問題への近似解を与える．pbirthday() は一致確率，qbirthday() は指定された一致確率に必要な観測数を計算する．

```
qbirthday(prob = 0.5, classes = 365, coincident = 2)
pbirthday(n, classes = 365, coincident = 2)
```

## 4.3 randu

VAX の FORTRAN 関数 RANDU (VMS 1.5) からとられた引き続く乱数の 400 個の三つ組

# 5 各論

## 5.1 R で使える疑似乱数発生法 (R 1.7.0) 以降

.Random.seed は乱数のシードを含む整数ベクトル．保管したり，元に戻したり出来るが，ユーザーが変更すべきではない．RNGkind は RNG の種類を問い合わせたり，変更するためのより扱いやすいインタフェイスである．RNGversion 以前の R のバージョンの乱数発生法を設定するために使うことが出来る (一貫性のために)．set.seed は乱数種を指定するためのお勧めの方法である．

```
用法:
.Random.seed <- c(rng.kind, n1, n2, ...)
save.seed <- .Random.seed

RNGkind(kind = NULL, normal.kind = NULL)
RNGversion(vstr)
set.seed(seed, kind = NULL)
```

| | |
|---|---|
| 引数: | |
| `kind:` | 文字，もしくは `NULL`．もし `kind` が文字列ならば，`R` の `RNG` を指定した方法に設定する．もし `NULL` ならば現在の手法を返す．`"default"` にすると現在の既定手法に戻す． |
| `normal.kind:` | 文字，もしくは `NULL`．もし `kind` が文字列ならば，`R` の正規乱数発生法を指定した方法に設定する．もし `NULL` ならば現在の手法を返す．`"default"` にすると現在の既定手法に戻す． |
| `seed:` | 単一の値．一つの整数と解釈される． |
| `vstr:` | バージョン番号を含む文字列，例えば `"1.6.2"` |
| `rng.kind:` | 上の `kind` に対する `0:k` 中のコード |
| `n1, n2, ...:` | 整数．`rng.kind` に依存する．詳細は各手法を参照 |

- ”Mersenne-Twister” R 1.7.0 以降の既定の疑似乱数発生法．Matsumoto and Nishimura (1998) の twisted GFSR 法．周期 $2^{19937}-1(=4.315\times10^{6001})$．すべての周期に関し 623 次元空間に均一分布．乱数種は 32 ビット整数の 624 次元ベクトル＋その中の現在位置を示す整数

- ”Wichmann-Hill”’ 乱数種, ‘.Random.seed[-1] == r[1:3]’ は長さ 3 の整数ベクトルであり，各 ‘r[i]’ は ‘1:(p[i] - 1)’ 中にある．ここで ‘p’ は長さ 3 の素数のベクトル ‘p =(30269, 30307, 30323)’ である．Wichmann-Hill 生成規則は長さ 6.9536e12 を持つ (= ‘prod(p-1)/4’, 原論文を修正した Applied Statistics (1984) 33, 123 を見よ)．

- ”Marsaglia-Multicarry”’ キャリーつきの乗算 RNG を使い，George Marsaglia が彼のメイリングリスト ‘sci.stat.math’ への投稿記事で紹介した．$2^{60}$ 以上の周期を持ち，すべての検査をパスする (Marsaglia による)．乱数種は二つの整数である (あらゆる値が許される)．

- ”Super-Duper”’ Marsaglia の有名な７０年代の Super-Duper 法．これは MTUPLE test of the Diehard battery をパスしないオリジナル版である．多くの初期値に対し周期 $4.6\times10^{18}$ を持つ．乱数種は二つの整数である (最初の種は任意，二番目は奇数でなければならない)．二つの種はそれぞれ Tausworthe と congruence 倍長整数である．S の ‘.Random.seed[1:12]’ への一対一対応が可能であるが，好評はしないし，この生成法は S-PLUS の最近のバージョンとは全く同じではない．

- ”Knuth-TAOCP” Knuth (1997) による．減算を伴う遅延つきフィボナッチ数列を用いる GFSR 法．つまり，使用される再帰式は X[j] = (X[j-100] - X[j-37]) mod $2^{30}$ であり，種は最近の 100 個の数の集合である (実際は 101 個の数であり，最後のものはバッファのサイクリックなシフトである)．周期は約 $2^{129}$．

- ”Knuth-TAOCP-2002” 以前のバージョンとは上位互換ではない 2002 年のバージョンであり，種からの GFSR の初期化が変更された．R は連続する種の選択を認めないという報告済みの弱点を持っており，種をすでに攪拌していた．

- ”user-supplied” ユーザー提供の発生法を使用．詳細は ‘Random.user’ を見よ．

### 5.2 正規疑似乱数発生法 ‘normal.kind’

- ”Inversion”’ (現在の既定手法)
- ”Kinderman-Ramage” (R 1.7.0 以前の既定手法，近似誤差を含み使うべきではない)
- ”Buggy Kinderman-Ramage”
- ”Ahrens-Dieter”
- ”Box-Muller”
- ”user-supplied”

‘set.seed’ はその単独の整数引数を用い，要求されただけの種を設定する．これは最小の整数引数を指定することにより全く異なった種を得，同時により複雑な手法（特に ‘”Mersenne-Twister”’ と ‘”Knuth-TAOCP”’）に適正な乱数種集合を得る，簡単な方法を目指している．

値:
‘.Random.seed’ は整数ベクトルであり，その最初の要素は RNG と正規乱数発生法の種類をコーディングしている．最後の二つの十進桁数は，k を利用可能な RNG の数として 0:(k-1) 中にある．この 0-99 の値は正規乱数の型を表す.

背後にある C コードでは ‘.Random.seed[-1]’ は ‘unsigned’ であり，従って R では ‘.Random.seed[-1]’ は負であっても良い.

‘RNGkind’ は，呼び出しの前に使用中の RNG と正規乱数発生法の二文字の文字ベクトルを返す．もしどちらの引数も ’NULL’ でなければコンソールには表示されない.
‘RNGversion’ は同じ情報を返す.

‘set.seed’ は ‘NULL’ を返すが，コンソールには現れない.

注意:
最初はシードは無い．必要な時現在の時間からつくり出される．したがって，既定では異なったセッションは異なったシミュレーション結果を与える.
.Random.seed’ は，少なくともシステムの発生法に対する，一様疑似乱数に対するシードのセットを保存する．他の発生法の状態を保存せず，特に，Box-Muller 正規疑似乱数の状態を保存しない．もし，後で状態を再生したければ .Rnadom.seed を設定するのではなく .set.seed を呼び出すべきである.

例

```
runif(1); .Random.seed; runif(1); .Random.seed
## もしシードがなければ新しいものが"ランダム"につくり出される.
rm(.Random.seed); runif(1); .Random.seed

RNGkind("Wich") # (kind に対する部分文字列マッチング)

## 以下は runif(.) が  Wichmann-Hill 法に対しどのように動作するかを
## R 関数のみで示す
p.WH <- c(30269, 30307, 30323)
a.WH <- c(  171,   172,   170)
next.WHseed <- function(i.seed = .Random.seed[-1])
  { (a.WH * i.seed) %% p.WH }
my.runif1 <- function(i.seed = .Random.seed)
  { ns <- next.WHseed(i.seed[-1]); sum(ns / p.WH) %% 1 }
rs <- .Random.seed
(WHs <- next.WHseed(rs[-1]))
u <- runif(1)
stopifnot(
 next.WHseed(rs[-1]) == .Random.seed[-1],
 all.equal(u, my.runif1(rs))
)

## ----
.Random.seed
ok <- RNGkind()
RNGkind("Super")  # "Super-Duper" にマッチ
RNGkind()
.Random.seed # Super-Duper に対する新しいシード

## リセット:
RNGkind(ok[1])
```

### 5.3 疑似乱数発生法による違いを示す例

2 種類の疑似乱数発生法の比較．引き続く 3 組みの一様疑似乱数を座標とする 3 次元単位立方体中の 1 千万個の点の断面 (実際は薄片) を座標面に射影する．MM 法等の乗算合同法の癖を示す例 (学芸大学の森さんの ”発見” に基づく例).

Mersenne-Twister 法使用 (R 1.7.0 以降の既定手法)

```
random1 <- function () {
  old.par <- par(no.readonly = TRUE); on.exit(par(old.par))
  png("MTrand.png")
  # まず枠だけ描く
  plot(c(0,0,1,1), c(0,1,0,1), main="", xlab="", ylab="", type="n")
  RNGkind(kind = "Mersenne-Twister")
  for (i in 1:10^7) {
    x <- runif(3)      # 一千万個の点の座標を一様疑似乱数で発生
    # x 座標が 1/1000 以下のものだけ y,z 座標に射影
    if (x[1] < 0.001) points(x[2], x[3], pch = ".")
  }
  dev.off()
}
```

Marsaglia-Multicarry 法使用 (R 1.7.0 以前の既定手法). すべての点が平行な面の上に乗っていることがわかる！

```
random2 <- function () {
  old.par <- par(no.readonly = TRUE); on.exit(par(old.par))
  png("MMrand.png")
  plot(c(0,0,1,1), c(0,1,0,1), main="", xlab="", ylab="", type="n")
  RNGkind(kind = "Marsaglia-Multicarry")
  for (i in 1:10^7) {     # 一千万個の点の座標を一様疑似乱数で発生
    x <- runif(3)
    # x 座標が 1/1000 以下のものだけ y,z 座標に射影
    if (x[1] < 0.001) points(x[2], x[3], pch = ".")
  }
  dev.off()
}
```

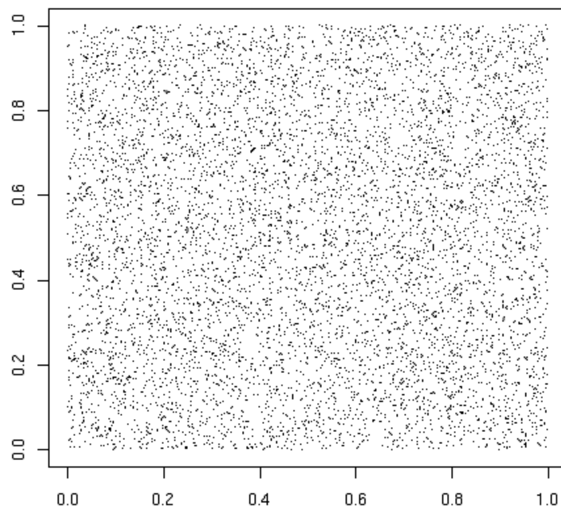


図 1: Mersenne-Twister 法

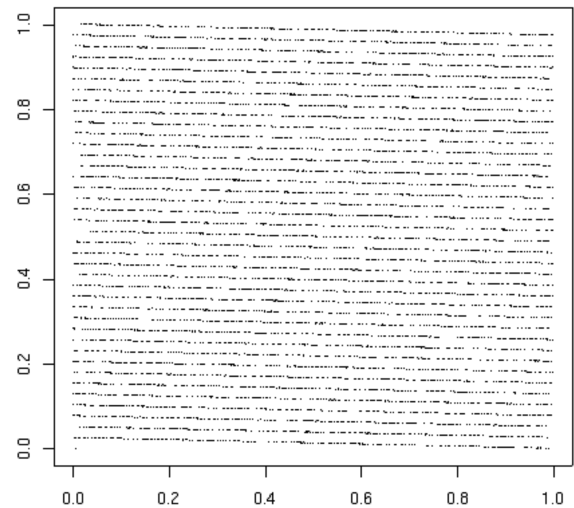


図 2: Marsaglia-Multicarry 法

### 5.4　ランダム抽出と並べ変え

書式

```
sample(x, size, replace = FALSE, prob = NULL)
```

引数

| | |
|---|---|
| `x` | ベクトル (数値，複素数，文字列，論理値) |
| `size` | 選び出される個数を表す正整数 |
| `replace` | 論理値．復元抽出を行なうか？ |
| `prob` | ベクトルから抽出する際の重み確率 |

- x が長さ 1 なら 1:x から抽出
- size の既定値は length(x)．sample(x) はランダムな並べ換えを意味する
- prob 比率を意味し，総和が 1 でなくても良いが，非負で，すべてがゼロであってはいけない．もし replace = FALSE ならば，これらの確率は sequential に適用される (つまり，次の項目の選択確率は残った項目に対する比率に比例して選ばれる)

```
> sample(1:10)  # ランダムな並べ換え
 [1]  4  1  8  3 10  9  5  2  6  7
> sample(1:10)
 [1]  9  3  6  4  7  8 10  1  2  5
> sample(1:10, 5)  # 5個を非復元抽出
[1] 10  9  7  2  5
> sample(1:10, replace=TRUE)  # 復元抽出
 [1] 10  2  6  7  4  3  1  9  1  7
> sample(1:10, 5, replace=TRUE)
[1]  7  4 10 10  7
# 成功確率 0.8 の長さ百のベルヌイ試行データを生成
> sample(c(0,1), 100,  replace = TRUE, prob = c(0.2, 0.8))
[1]  1 1 1 0 1 1 0 1 1 1 0 1 1 1 1 0 1 0 0 1 1 0 1 1 1 0 1 1 1 0 1 0 0 1 0 1 0
[38] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 1 1 0 1 1 1 1 1 1 0 1 0 1 0 0 0 0
[75] 1 1 1 1 1 0 1 1 1 0 1 1 1 1 1 0 1 1 1 0 1 1 0 1 0 1
# 単純無作為抽出法のパラドックス
# 10 万個の母集団の平均値が 100 個の単純無作為標本の平均で十分推定可能なら
# 100 万個の母集団の平均値も 100 個の単純無作為標本の平均で十分推定可能!
> x <- rnorm(1000000)
> y <- numeric(10000)
> for (i in 1:10000) y[i] <- mean(sample(x,100))
> sd(y)   # 百万個から百個を単純無作為抽出した時の推定平均値の標準偏差
[1] 0.1003183
> x <- rnorm(100000)
> for (i in 1:10000) y[i] <- mean(sample(x,100))
> sd(y)  # 10 万個から百個を単純無作為抽出した時の推定平均値の標準偏差
[1] 0.1000383   # 殆んど一緒
```

## 5.5 与えた周辺和を持つランダムな二次元分割表

書式

```
r2dtable(n, r, c)
```

引数

n　　: 生成する分割表の数

r　　: 行和を与えるベクトル

c　　: 列和を与えるベクトル (c, r の総和は一致する必要)

返り値 : 行・列和がそれぞれ r,c であるランダムな二次元分割表 n 個のリスト

```
> x = r2dtable(2, c(10,10), c(15,5))
> x
[[1]]
     [,1] [,2]
[1,]    8    2
[2,]    7    3

[[2]]
     [,1] [,2]
[1,]    6    4
[2,]    9    1
> r2dtable(1,c(3,6,4), c(4,5,4))
[[1]]
     [,1] [,2] [,3]
[1,]    0    2    1
[2,]    3    1    2
[3,]    1    2    1
```

n 行 m 列の二次元分割表を対象にする検定で，計算量が大きい場合にモンテカルロ法により近似的な解を求めるときに使える．

## 5.6 乱数の再現 set.seed() 関数 (2003.12.27)

(疑似) 乱数は R 起動時に適当に初期化 (システム時間を利用？) され，それ以降毎回異なった乱数が発生される．もし数値計算の再現性が問題になる時は，乱数種を set.seed(整数値) 関数で指定する．与えられた整数値は内部的に各乱数生成法に適切な真の乱数種 (それを知る必要はほとんど無いはず) に変換される．

```
> runif(5)
[1] 0.8797957 0.7068747 0.7319726 0.9316344 0.4551206
> runif(5) # 当然毎回違った乱数が得られる
[1] 0.59031973 0.82043609 0.22411848 0.41166683 0.03861056
> set.seed(101); runif(5)  # 乱数種を指定
[1] 0.37219838 0.04382482 0.70968402 0.65769040 0.24985572
> set.seed(101); runif(5)  # 乱数種を同じにすれば同じ結果を
                           # 再現 (疑似="いんちき"たるゆえん)
[1] 0.37219838 0.04382482 0.70968402 0.65769040 0.24985572
```

## 5.7 逆関数法による乱数の作り方 (2004.01.26)

上に紹介している分布に無い分布に従う乱数を生成したくなる場合がある．まず，$U$ を一様分布に従う確率変数とし，$X = F^{-1}(U)$ は

$$P(X \leq x) = P(F^{-1}(U) \leq x) = P(U \leq F(x)) = F(x)$$

となることより $X$ は分布関数 $F$ に従う．よって，分布関数の逆関数の引数に一様乱数を入れてやれば，欲しい分布の乱数が得られる．

指数分布の分布関数 $F(x) = 1 - \exp(-x)$ の逆関数は $-\log(1-x)$ なので，$X = -\log(1-U)$ とすれば $X$ は平均 1 の指数分布に従う．さらに，$1-U$ と $U$ は同分布なので，$X = -\log(U)$ の $U$ に一様乱数を入れてやることで平均 1 の指数乱数が得られる （ただし $x > 0$）．ラプラス分布（両側指数分布）の密度関数は $f(x) = 1/2\exp(-|x|)$ だが，この場合は一様乱数 $U_1, U_2$ を生成し，$U_1$ が 1/2 以上ならば $-\log(U_2)$ を，$U_1$ が 1/2 未満ならば $\log(U_2)$ を採択すればラプラス分布に従う乱数が得られる．

## 5.8 両側指数分布の乱数を発生させる (2005.7.23)

```
myrand <- function(n) {
  u <- ifelse(runif(n)>1/2, 1, -1)
  v <- log(runif(n))
  return(u*v)
}
mydata <- myrand(100000)                          # 乱数を生成
plot(density(mydata), xlim=c(-7, 7),
     ylim=c(0, 0.5), col="red", ann=F)            # 乱数のグラフ
laplace <- function(x) 1/2*exp(-abs(x))           # 密度関数
par(new=T)
curve(laplace, xlim=c(-7, 7), ylim=c(0, 0.5))  # 本当のグラフ
```

「平均と分散をいろいろ変えてみたい！」とおっしゃる場合は，逆関数法に従って上の関数定義を修正して下さい．

## 5.9 棄却法による乱数の作り方 (2004.01.15)

逆関数法は理論的には正しい方法なのだが，コンピュータは有限な値しか生成できないので，無限のサポートを持っている分布の裾の方の乱数の精度が悪くなる場合がある（例:棄却法で生成した指数乱数），そこで以下に棄却法による乱数生成の方法を紹介する．用いる関数を先に説明する．用いる関数・定数を先に説明する．

```
f(x)    :  生成したい乱数が従う分布の密度関数
g(x)    :  乱数生成が容易な分布の密度関数
c(>=1)  :  f(x) <= c*g(x) が成り立つ定数 (小さい方が良い)
h(x)    =  f(x)/{c*g(x)}
```

生成するアルゴリズムはこちら．

```
u <- runif(1)
v <- g(x) に従う乱数
u <= h(v) ならば v を乱数として採択する
```

## 5.10 $f(x) = 1/2\sin(x)(0 < x < \pi)$ という分布に従う乱数を生成する (2005.7.23)

$$g(x) = 1/\pi \qquad (0 < x < \pi)$$
$$c = \pi/2$$

とおくと $f(x) \leq c * g(x)$ が成り立ち，この場合は $h(x) = \sin(x)$ となる．よって以下の関数 myrand(x)（x：生成する乱数の数）で生成することが出来る．

```
myrand <- function(x) {
  y <- c()
  i <- 1
  while (i <= x) {
    u <- runif(1)
    v <- pi*runif(1)
    w <- sin(v)
    if (u < w){
      y <- append(y, v)
      i <- i+1
    }
  }
  return(y)
}


> yy <- myrand(1000)   # 乱数を 1000 個生成
```

```
> plot(density(yy))   # し，このデータについて密度推定
```

## 5.11 アドオンパッケージ gld (generalized (Tukey) lambda distribution) 2004.09.15

アドオンパッケージ gld (Tukey の一般化ラムダ分布) に関する関数がある．この分布は少数のパラメータで様々な形状の確率分布を表現できる．以下はそのインデックスである．

```
rgl                    random numbers from the generalised lambda distribution
qgl                    quantiles of the generalised lambda distribution
pgl                    probabilities of the generalised lambda distribution
dgl                    densities of the generalised lambda distribution
qdgld                  quantile densities of the generalised lambda distribution
gl.check.lambda        Function to check the validity of parameters
                       of the generalized lambda distribution
plotgl                 Plots of density and distribution function for
                       the generalised lambda distribution
plotgld                Plot probability density functions of the generalised lambda distribution
plotglc                Plot distribution functions of the generalised lambda  distribution
qqgl                   Quantile-Quantile plot against the generalised lambda  distribution
starship               Carry out the "starship" estimation method for
                       the generalised lambda distribution
starship.adaptivegrid  Carry out the "starship" estimation method for
                       the generalised lambda distribution using a   grid-based search
starship.obj           Objective function that is minimised in starship  estimation method
```

## 5.12 3 変量正規分布の乱数

は以下のように生成することが出来る．以下で用いている行列の平方根についてはこちら．

```
r3norm <- function(mu, A, n) {
  U <- svd(A)$u
  V <- svd(A)$v
  D <- diag(sqrt(svd(A)$d))
  B <- U %*% D %*% t(V) # 行列 A の平方根
  w <- c()
  for (i in 1:n)
    w <- append(w, list(mu + B%*%cbind(rnorm(3))))
  return(w)
}

mu <- cbind(c(1,1,1)) # 平均ベクトル (縦ベクトル)
A <- array(c(2,1,1,1,2,1,1,1,2), dim=c(3,3))
w <- r3norm(mu, A, 2000)
```

## 5.13 3 変量 $t$ 分布の乱数

は以下のように生成することが出来る．まず，アルゴリズムを書く代わりに命題みたいなものを示す．

```
R                  :  自由度 m のカイ二乗分布に従う確率変数
Z                  :  p 変量正規分布 N( 0 , I p ) に従う確率ベクトル
                      (独立な標準正規乱数がズラッと縦に並んでいる)
V                  :  正定値対称行列（ちらばりを表す行列で分散共分散行列
                      とは少し異なるもの)
V = C %*% t(C)     :  コレスキー分解
X = sqrt(m/R)*Z    :  p 変量楕円 t 分布 met( p , m , 0 , I p ) に従う
Y = μ + C %*% X    :  p 変量楕円 t 分布 met( p , m , μ , V )に従う
```

以下に 3 変量楕円 $t$ 乱数に従う乱数を生成する関数を定義する．

```
met3 <- function(m, mu, V, n) {
  # m : 自由度
  # mu : 平均ベクトル
  # V : 散らばり行列
  # n : 乱数の個数
  U  <- svd(V)$u
  V1 <- svd(V)$v
  D  <- diag(sqrt(svd(V)$d))
  B  <- U %*% D %% t(V1)
  w <- c()
  for (i in 1:n) {
    R <- 0
    for (j in 1:m) R <- R + rnorm(1)^2
    w <- append(w, list(mu + B %*% (cbind(rnorm(3))*sqrt(m/R))))
  }
  return(w)
}
```