Rで統計解析入門

(17) 関数とシミュレーション〔例数設計の準備〕

本日のメニュー

1. ベクトル
2. 関数の作成方法
3. 条件分岐とくり返し
4. シミュレーション

【復習】変数とは

- 「変数名 <- 計算式」で計算結果を保存することが出来る
  （「変数」への「代入」という　；　"<-" は代入記号）

```
> x <- 1 + 2
```

- 保存した値は変数名を入入力することで再表示される

```
> x
[1] 3
```

- 丸括弧を使うと，値の保存と表示を同時に実行できる

```
> ( x <- 1 + 2 )
[1] 3
```

【復習】変数の使用例

- ある 5 人の体重について，体重の合計値を求める

```
> x1 <- (55 + 60 + 65 + 70 + 75)
> x1
[1] 325
```

- 同じ 5 人の体重について平均値を求める

```
> x2 <- (55 + 60 + 65 + 70 + 75) / 5
> x2
[1] 65
```

- 同じ 5 人の体重について，今度は分散を求め・・・
  （同じデータなのだから，手間を省く方法は無いの？？）

ベクトル

- 1 つの変数に 1 つの値を代入することが出来たように,
  1 つの変数に複数の値を代入することも出来る

```
> x <- c(55, 60, 65, 70, 75)
```

- 複数の値が代入された変数を「ベクトル」とよぶ

※ 1 つの値しか代入されていない変数も「ベクトル」とよぶ

```
> x
[1] 55 60 65 70 75
```

ベクトル用の関数

- R には「ベクトル」用の関数が多数用意されている

```
> sum(x)
[1] 325
```

- ベクトルに入っているデータ数（要素数）を計算する場合は
  関数 length() を使用する

```
> length(x)
[1] 5
```

【参考】R で用意されているベクトル用関数

ベクトル用関数

関　数	cor(x)	max(x)	mean(x)	median(x)	min(x)
意　味	相関係数	最大値	平均値	中央値	最小値
関　数	prod(x)	summary(x)	sd(x)	sum(x)	var(x)
意　味	総積	要約統計量	標準偏差	総和	不偏分散

ベクトル操作

- ベクトルの中の数値を取り出す場合は [] を使用する

```
> x[2]              # 変数 x の 2 番目の値を取り出す
[1] 60
```

- ベクトルとベクトルを結合する場合は以下の様にする

```
> c(x, 80)          # c(ベクトル，ベクトル)
[1] 55 60 65 70 75 80
```

- ベクトルの中の数値を書き換える場合は以下の様にする

```
> x[2] <- 99        # 変数 x の 2 番目の値を 99 に修正
> x
[1] 55 99 65 70 75
```

ベクトル → データフレームの作成

- ベクトルを作成した後，データフレームを作成することが出来る
- 「薬剤の種類」「QOL」などのデータをベクトルで用意した後，関数 data.frame() で１つのデータフレームに変換する

```
> x <- c("A","A","B","B","B")
> y <- c(8, 6, 7, 5, 3)
> MYDATA <- data.frame(
+   GROUP = x,                # 薬剤の種類
+   QOL   = y                 # QOL
+ )
> MYDATA
  GROUP QOL
1     A   8
2     A   6
3     B   7
4     B   5
5     B   3
```

ベクトル → データフレームの作成

- ベクトルを作成した後，データフレームを作成することが出来る
- 乱数により「薬剤の種類」「QOL」のデータを生成した後，
  関数 data.frame() で１つのデータフレームに変換することも出来る

```
> MYDATA <- data.frame(
+   GROUP = c( rep(1,20), rep(2,20) ),        # 各薬剤20例ずつ
+   QOL   = c( rnorm(20, mean=6.5, sd=3.0),  # 薬剤AのQOLの乱数
+              rnorm(20, mean=4.0, sd=3.0))  # 薬剤BのQOLの乱数
+ )
> head(MYDATA)
  GROUP       QOL
1     1  5.406269
2     1  4.722682
3     1  8.535632
4     1  6.318657
5     1  3.755226
6     1 11.039495
```

演習問題

1. 「0.8-4」の結果を変数 x に代入してください
2. 整数部分 trunc(x) と切り下げ floor(x) の違いは何でしょう?
3. 以下の 5 つのデータを変数 y に格納してください

1	2	3	4	5

4. 変数 y の値と変数 Y（大文字）の値を表示してください
5. 3. で作成した変数 y の平均と標準偏差を求めてください
6. 変数 y の後ろに，変数 x を結合し，結果を変数 z に格納してください
7. 6. で作成した変数 z の 6 番目の値を表示してください

演習問題の回答

```
> x <- 0.8-4                  # 設問 1 の回答例
> trunc(x) ; floor(x)         # 設問 2 の回答例（結果は省略）
[1]      -3          -4       # マイナスの場合は注意！
> ( y <- c(1,2,3,4,5) )       # 設問 3 と 4 の回答例
[1] 1 2 3 4 5
> Y                           # R では大文字と小文字を区別する！
 エラー：  オブジェクト "Y" は存在しません
> mean(y)                     # 設問 5 の回答例（変数 y の平均値）
[1] 3
> sd(y)                       # 設問 5 の回答例（変数 y の標準偏差）
[1] 1.581139                  # sd() は不偏標準偏差であることに注意！
> ( z <- c(y, x) )            # 設問 6 の回答例
[1] 1 2 3 4 5 6
> z[6]                        # 設問 7 の回答例
[1] 6
```

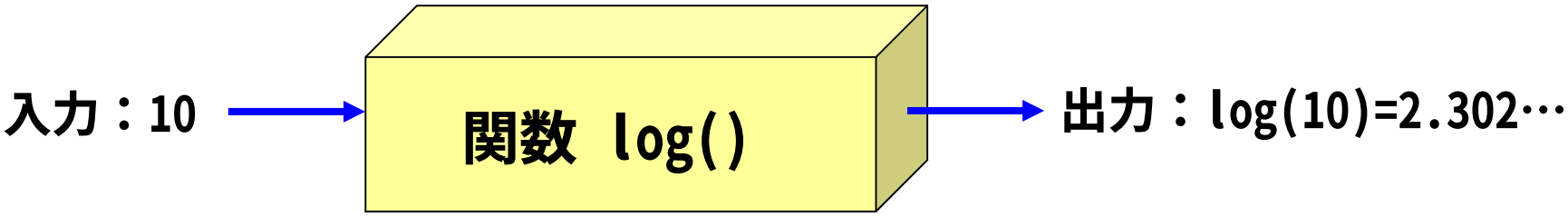
本日のメニュー

1. ベクトル
2. 関数の作成方法
3. 条件分岐とくり返し
4. シミュレーション

関数について

- R では「特別な機能を果たす命令」を「関数」という形で呼び出すことが出来る
- 関数 log(x) は「 x の値の対数を計算する関数」だが，関数に x の値を指定する場合には「引数」という形で関数に与える

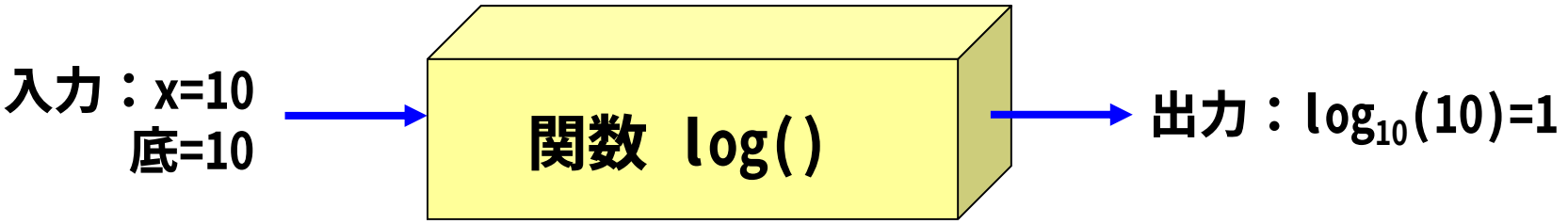
```
> log(10)
[1] 2.302585
```



関数について

- 引数に式や関数を指定すれば，それを評価した値が引数として使われる
- 引数が 2 個以上ある場合はコンマ（ , ）で区切って並べればよい

```
> log(10, base=10)    # 底を指定する引数 base に 10 を指定する
[1] 1
```

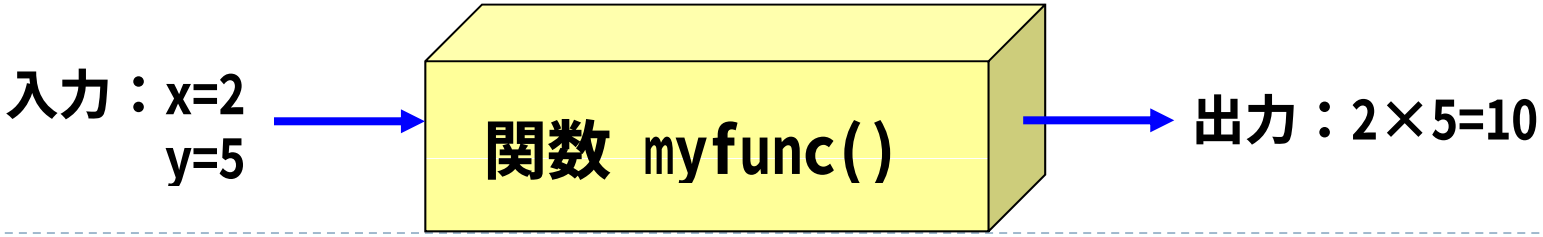


関数を定義する手順

1. 関数名を決める
2. 入力する変数の個数と種類を指定する
3. 計算手順を 1 行ずつ記述し，最後に関数 return() で計算結果を出力する

【例】入力した 2 つの数値の積が出力される関数 myprod(x,y)

```
> myfunc <- function(x, y) {          #
+   return(x*y)                       # 関数定義部分
+ }                                   #
> myfunc(2,5)                         # 関数を実行する
[1] 10
```



数学関数の定義

- 関数 f(x) = 2x

```
> f <- function(x) {
+   return(2*x)
+ }
> f(3)
[1] 6
```

- 二次元標準正規分布の密度 z(x, y) = $\frac{1}{2\pi}\exp\left\{\frac{-(x^2+y^2)}{2}\right\}$

```
> z <- function(x,y) {
+   return( 1/(2*pi)*exp(-(x^2+y^2)/2) )
+ }
> z(0,0)
[1] 0.1591549
```

数学関数のプロット

- 関数 curve(関数名, x 軸の下限, x 軸の上限) を用いる
- 関数 f(x) = x2 のプロットを行う場合：

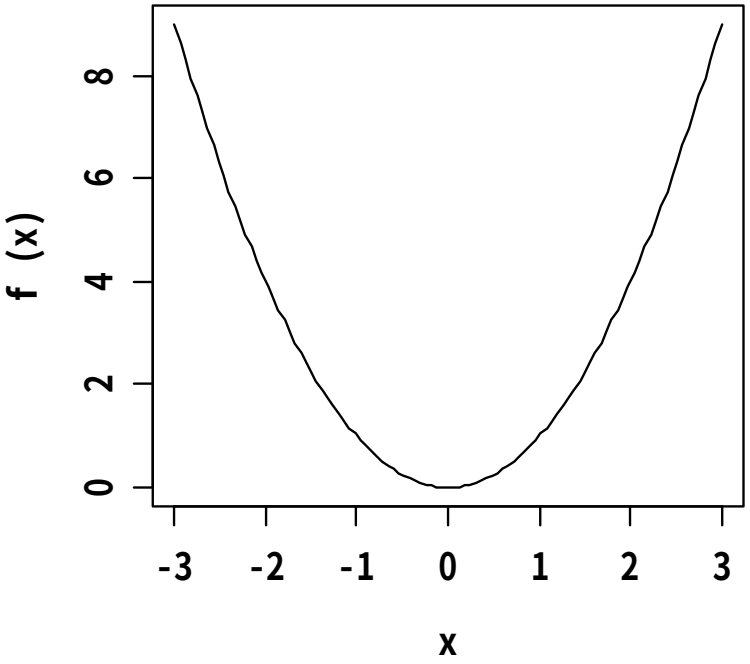
```
> f <- function(x) {
+   return(x^2)
+ }
> curve(f, -3, 3)          # -3 ～ 3 の範囲でプロット
```

- 関数 $f(x, a) = x^a$ のプロットを行う場合（a = 3 とする）：

```
> f <- function(x, a) {
+   return(x^a)
+ }
> curve(f(x, 3), -3, 3)    # -3 ～ 3 の範囲で a=1 としてプロット
```

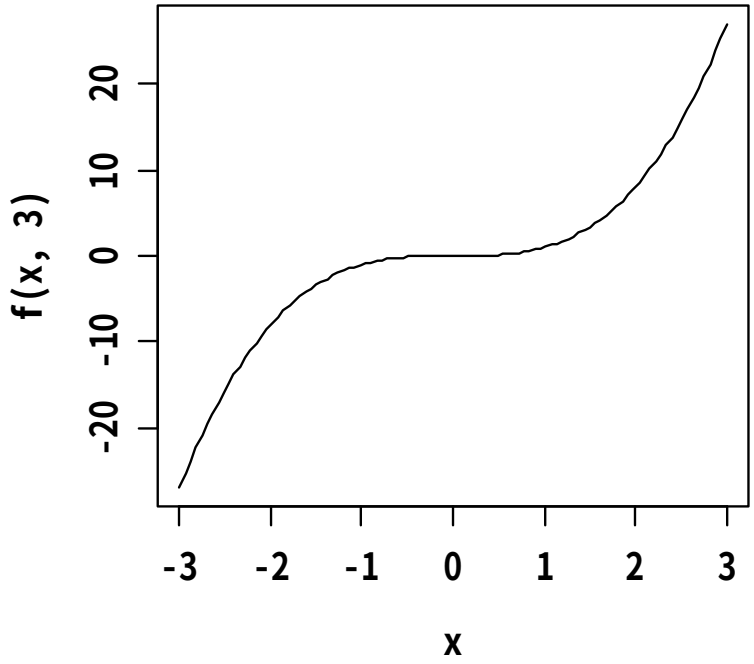
数学関数のプロット

$y = x^2$ のプロット




```
> f <- function(x) {
+   return(x^2)
+ }
> curve(f, -3, 3)
```

$y = x^a$ のプロット（a=3）




```
> f <- function(x, a) {
+   return(x^a)
+ }
> curve(f(x, 3), -3, 3)
```

数学関数のプロット

- 3 次元のグラフを描くときは関数 persp() を用いる

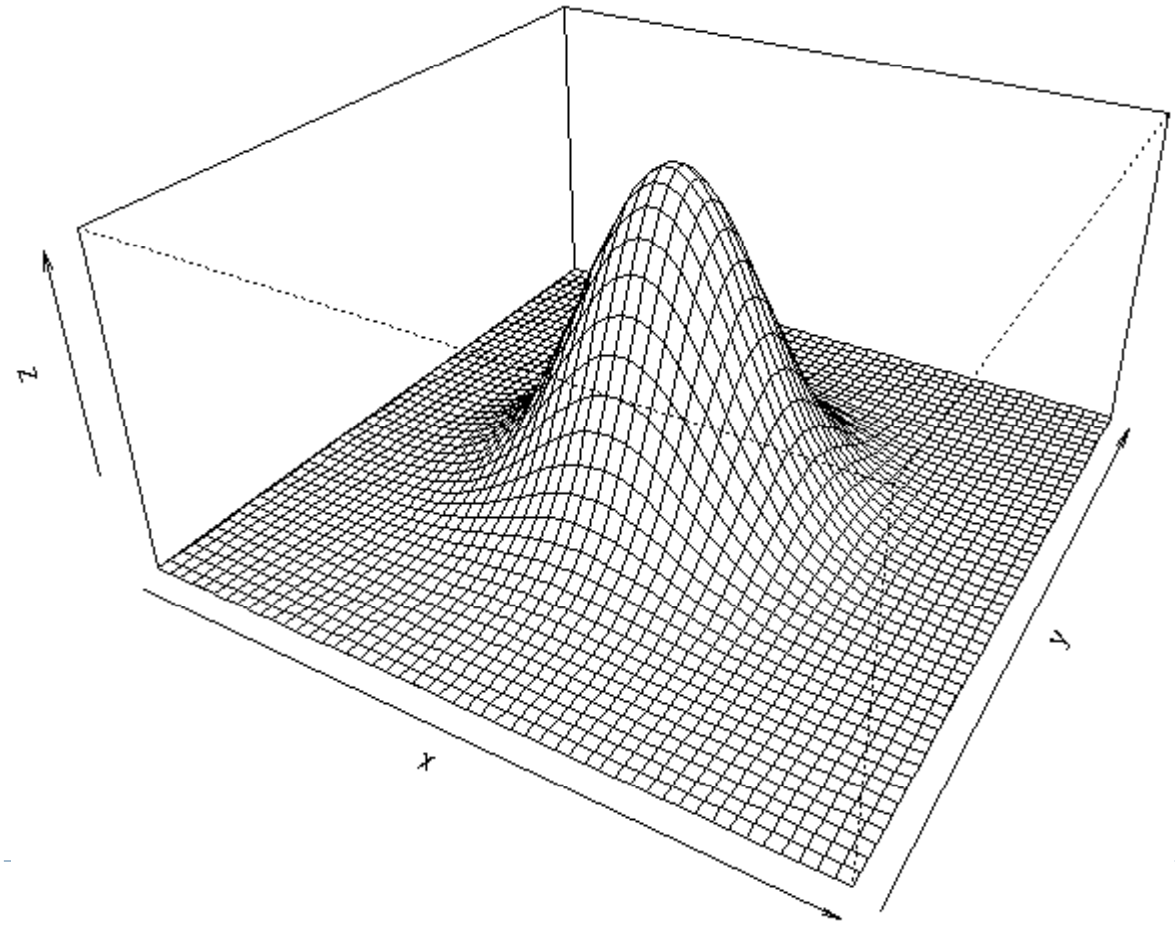
- 関数 f(x, y) = $\frac{1}{2\pi}\exp\left\{\frac{-(x^2+y^2)}{2}\right\}$ のプロット

1. 関数を定義した後，x 軸と y 軸の点を用意する
2. 関数 outer(x, y, 関数名) で z 軸の点を用意する
3. 関数 persp(x, y, z) でプロットする

```
> f <- function(x, y) return( 1/(2*pi)*exp(-(x^2+y^2)/2) )
> x <- seq(-4, 4, length= 50)
> y <- x
> z <- outer(x, y, f);
> persp(x, y, z, theta = 30, phi = 30, expand = 0.5)
```

数学関数のプロット

関数 f(x, y) = $\frac{1}{2\pi}\exp\left\{\frac{-(x^2+y^2)}{2}\right\}$ のプロット



数学関数のプロット

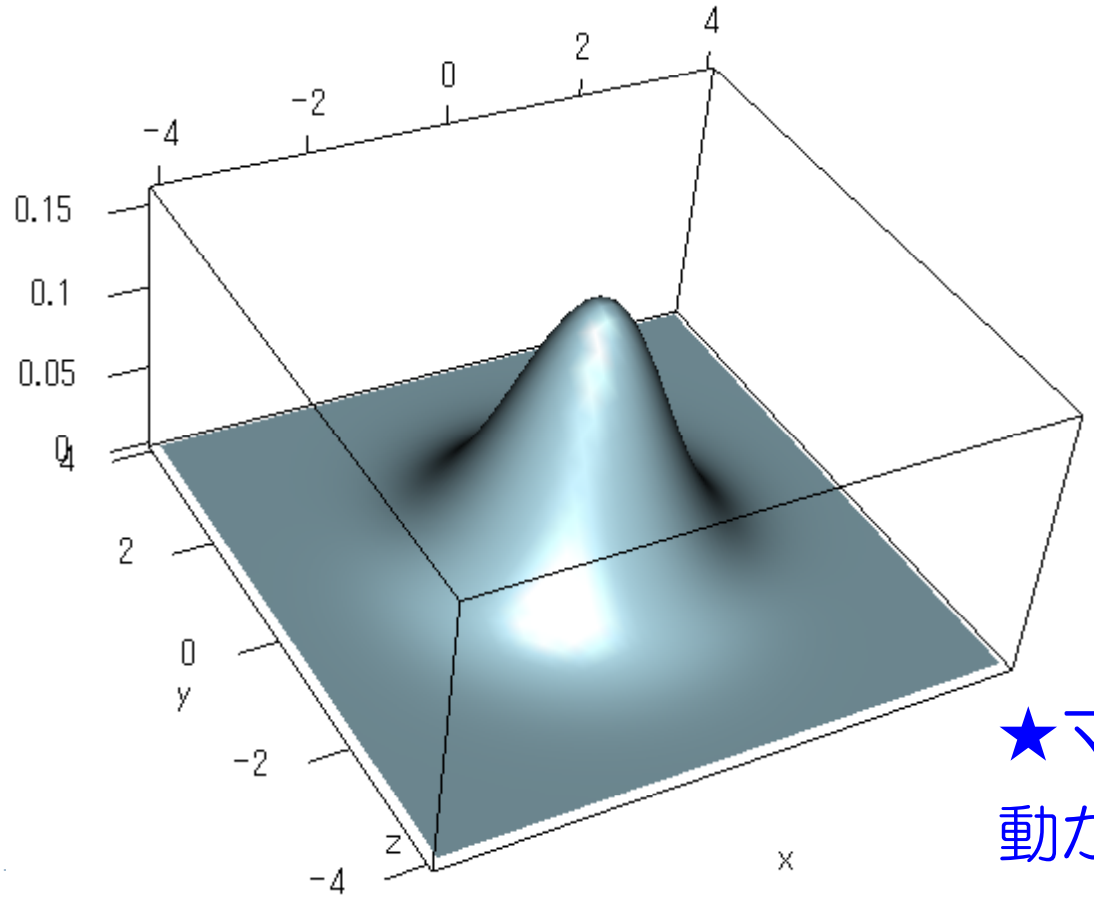
- パッケージ rgl の関数 persp3d() でも OK！

1. 関数を定義した後，x 軸と y 軸の点を用意する
2. 関数 outer(x, y, 関数名) で z 軸の点を用意する
3. 関数 persp3d(x, y, z) でプロットする

```
> f <- function(x, y) return( 1/(2*pi)*exp(-(x^2+y^2)/2) )
> x <- seq(-4, 4, length= 50)
> y <- x
> z <- outer(x, y, f)
> library(rgl)
> open3d()
> persp3d(x, y, z, aspect=c(1,1,0.5), col="lightblue")
```

数学関数のプロット

関数 f(x, y) = $\frac{1}{2\pi}\exp\left\{\frac{-(x^2+y^2)}{2}\right\}$ のプロット



★マウスでグリグリ
動かすことが出来る

演習問題

1. 関数 $g(x, a) = \dfrac{1}{1+e^{-a-x}}$ を定義してください
2. 1. で定義した関数のグラフを描いてください
3. if-else 文を用いることで，引数が 1 よりも大きいかどうかを判定する関数 myfunc00(x) が定義出来ます

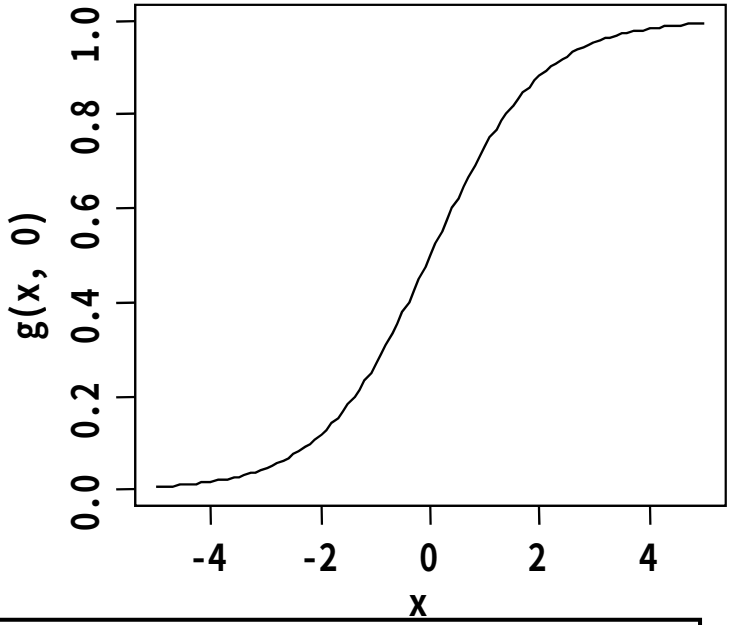
```
> myfunc00 <- function(x) {
+   if (x > 1) return(1)   # 引数 x が 1 よりも大きい場合は 1
+   else       return(0)   # そうでない場合は 0
+ }
> myfunc00(2)
[1] 1
```

関数 myfunc00(x, y) を参考にして，絶対値を求める関数 myabs(x, y) を定義してください

演習問題の回答

- 関数 g(x, a) = 1/(1+exp(-a-x))

```
> g <- function(x, a) {
+   return( 1/(1+exp(-a-x)) )
+ }
> g(0,0)
[1] 0.5
> curve(g(x, 0), -5, 5)
```

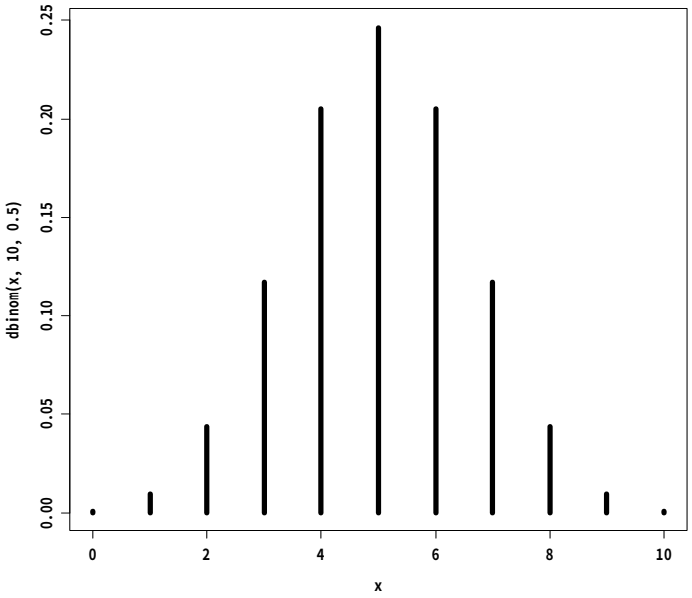


- 絶対値を求める関数 myabs(x)

```
> myabs <- function(x) {
+   if (x < 0) return(-x)    # 引数 x が 0 よりも小さい場合は -x
+   else       return(x)     # そうでない場合は x
+ }
> myabs(-2)
[1] 2
> myabs(2)
[1] 2
```

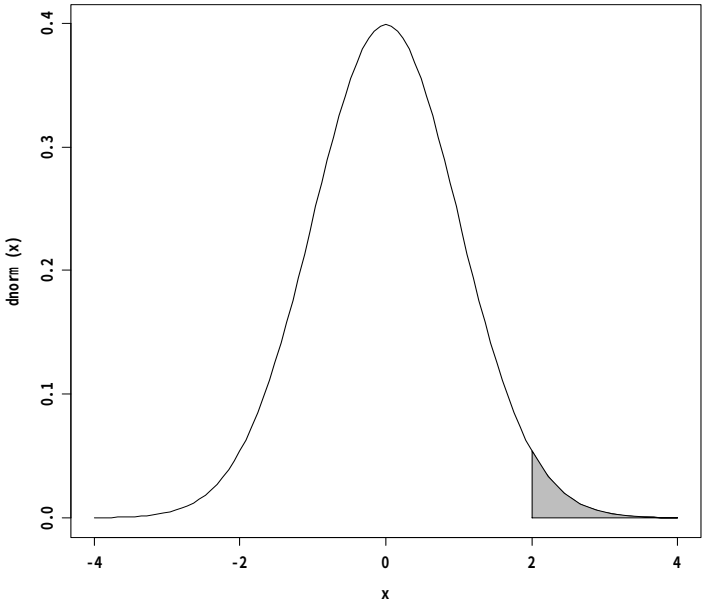
確率分布のプロット

二項分布のプロット




```
> x <- 0:10
> plot(x, dbinom(x,10,0.5),
+       type="h", lwd=5) # プロット
```

正規分布のプロット（影付き）

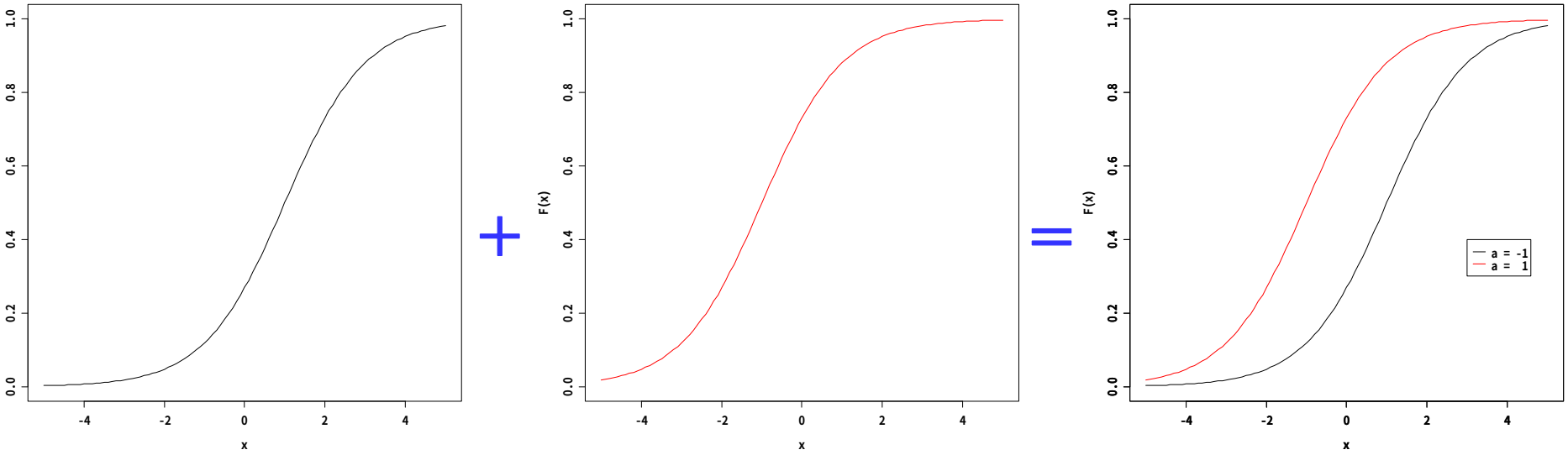



```
> curve(dnorm, -4, 4, type="l") # プロット
> xvals <- seq(2, 4, length=10) # 影を追加
> dvals <- dnorm(xvals)
> polygon(c(xvals,rev(xvals)),
+       c(rep(0,10),rev(dvals)),col="gray")
```

確率分布のプロット

分布名	関数名	分布名	関数名
ベータ分布	dbeta	ロジスティック分布	dlogis
二項分布	dbinom	多項分布	dmultinom
コーシー分布	dcauchy	負の二項分布	dnbinom
カイ二乗分布	dchisq	正規分布	dnorm
指数分布	dexp	ポアソン分布	dpois
F分布	df	Wilcoxon の符号付順位和統計量の分布	dsignrank
ガンマ分布	dgamma	t 分布	dt
幾何分布	dgeom	一様分布	dunif
超幾何分布	dhyper	ワイブル分布	dweibull
対数正規分布	dlnorm	Wilcoxon の順位和統計量の分布	dwilcox
ベータ分布	dbeta	ロジスティック分布	dlogis

【参考】重ねた図の描き方



■ ２枚の図を１枚に重ねて表示することを考える

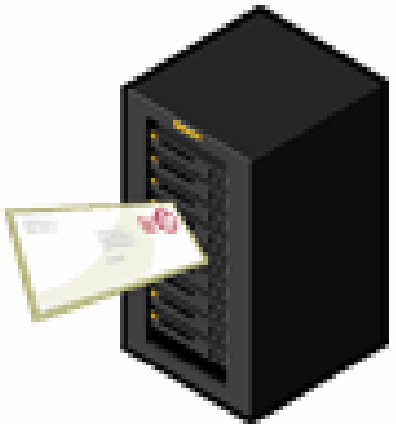
```
F <- function(x, a) { 1/(1+exp(-a-x)) }   # 作図する関数
curve(F(x,-1), col=1, xlim=c(-5,5),  ylim=c(-0,1), ylab="")
par(new=T)
curve(F(x, 1), col=2, xlim=c(-5,5),  ylim=c(-0,1), ylab="F(x)")
legend(3, 0.4, c("a = -1","a =  1"), lty=1, col=1:2)
```

本日のメニュー

1. ベクトル
2. 関数の作成方法
3. 条件分岐とくり返し
4. シミュレーション

プログラミングについて

- プログラミングとは
  - 人間がコンピュータに命令をすること
  - R の場合は「ユーザーが R のコマンドをひとつひとつ記述する」作業のこと ⇒ 関数定義！
- R におけるプログラミングのための道具は・・・
  - 条件分岐（if）
  - くり返し（for）
  - 変数，ベクトル，関数定義・・・



条件分岐〔 if 〕

- ある「条件」に合致する場合は処理を実行 ⇒ if を用いる

```
if（条件）{ 条件に合致する場合に実行する処理 }
```

- 「条件」は「比較演算子」を使って判定する

```
> if (x < 0) x <- -x    # x が 0 未満の場合はプラスに
```

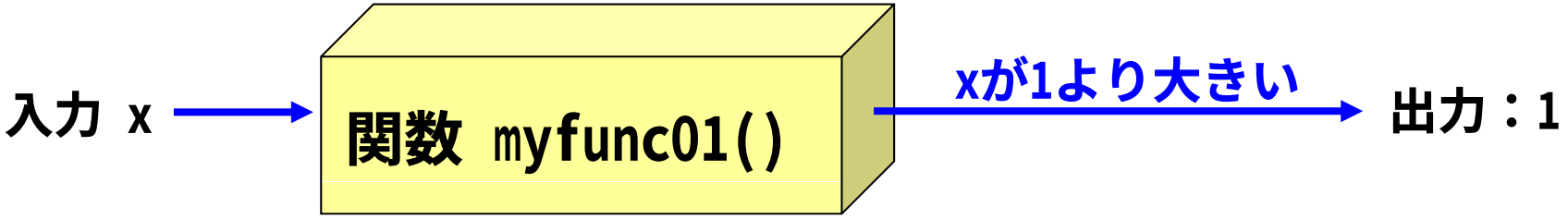
比較演算子

記号	==	!=	>=	>	<=	<
意味	等しい	≠	≧	＞	≦	＜

〔if〕の使用例

- 引数 x が 1 より大きい場合は 1 を出力する

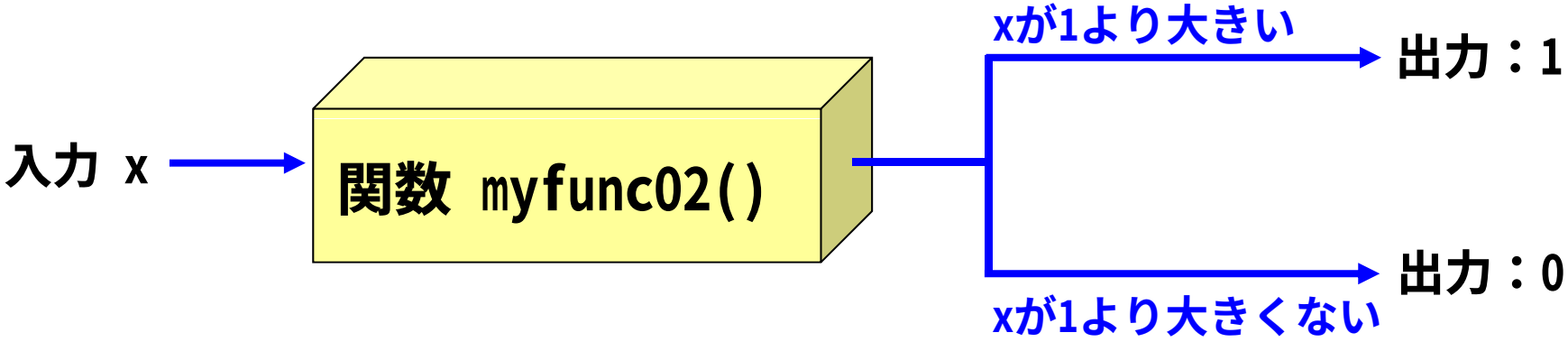
```
> myfunc01 <- function(x) {
+   if (x > 1) return(1)    # 引数 x が 1 よりも大きい場合は 1
+ }
> myfunc01(2)
[1] 1
> myfunc01(0)                    # 何も起こらない
```



〔if〕の使用例

- 引数 x が 1 より大きい場合は 1 を出力し，
  引数 x が 1 より大きくない場合は 0 を出力する

```
> myfunc02 <- function(x) {
+   if (x > 1) return(1)   # 引数 x が 1 よりも大きい場合は 1
+   else       return(0)   # そうでない場合は 0
+ }
> myfunc02(0)
[1] 0
```



〔if〕の使用例

- 引数 x が 1 より大きい場合は 1 を出力し，
  引数 x が 0 ～ 1 の場合は 0 を出力し，
  引数 x が 0 より小さい場合は -1 を出力する

```
> myfunc03 <- function(x) {
+   if      (x >  1) return( 1)   # 引数 x が 1 よりも大きい
+   else if (x >= 0) return( 0)   # 引数 x が 0 ～ 1
+   else             return(-1)   # 上 2 つのどれでもない
+ }
> myfunc03(2)
[1] 1
> myfunc03(0)
[1] 0
> myfunc03(-1)
[1] -1
```

くり返し〔 for 〕

- ある「処理」をくり返し実行する ⇒ for を用いる

```
for (i in 1:くり返し数) { くり返し実行する処理 }
```

- 「処理」として「変数 x に 1 を足す」をくり返す

```
> x <- 0                          # x に 0 を代入
> for (i in 1:5) x <- x+1         #「x に 1 を足す」を
                                  # 5 回くり返す
> x                               # x の中身を確認
[1] 5
```

〔 for 〕の使用例

- 「変数 x に k を足す」を 5 回くり返す

```
> x <- 0                          # x に 0 を代入
> for (k in 1:5) x <- x+k         # x に k を足す
> x                               # x を表示
[1] 15
```

- 「ベクトル x に i をくっつける」を 5 回くり返す

```
> x <- c()                        # 空のベクトルを用意
> for (i in 1:5) x <- c(x, i)     # x に i をくっつける
> x                               # x を表示
[1] 1 2 3 4 5
```

〔 for 〕の使用例

- 「 1 から x までの間の偶数を全て足し合わせる」という関数 myfunc04() を定義する

```
> myfunc04 <- function(x) {
+   a <- 0                          # a に 0 を代入
+   for (i in 1:x) {
+     if (i%%2 == 0) a <- a+i       # i を 2 で割った余り
+   }                               # が 0 なら i を足す
+   return(a)
+ }
> myfunc04(10)                      # 1～10 の偶数の和
[1] 30
```

【参考】演算子の種類

```
> x <- 1;  y <- 2;  z <- c(3, 4, 5)
> (x > 0) && (y > 0)
[1] TRUE                    # TRUE：条件に合致，FALSE：条件に合致しない
```

ひとつの値同士の比較に用いる論理演算子

記号	!	&&	||
意味	否定	かつ	または

```
> ( z > c(4, 4, 4) ) | ( z < c(3, 3, 3) )
[1] FALSE  FALSE  TRUE          #「TRUE：真，FALSE：偽」ともいう
```

ベクトル同士の比較に用いる論理演算子

記号	!	&	|
意味	否定	かつ	または

本日のメニュー

1. ベクトル
2. 関数の作成方法
3. 条件分岐とくり返し
4. シミュレーション

シミュレーションについて

- シミュレーションを日本語に訳すと「模擬実験」
- ある場面を想定したときに，場面が時間を追うごとにどのような変化をするのかを実験によって知るのが目的
- シミュレーションを行う必要が出てくる場面：
  - 実際に実験を行うことが難しい場合
  - 場面がどのような変化をするのか予想しにくい場合
- シミュレーションを行う場合，様々な仮定を置いた上で実験を行う
- R の場合は「乱数を利用」して「プログラム（関数）を作成する」ことが多い

シミュレーションを行う手順の一例

1. シミュレーションを行う目的を確認して，場面設定を行う
2. シミュレーションを行うためには，どのようなことをすればよいか，実際の手順（アルゴリズム）を決める
   ⇒ R ならば手順が決まった後に関数を定義する
3. シミュレーションを実行して結果を出力する
4. 結果を整理し，どのようなことが分かったのかを検討
5. シミュレーションの結果を解釈するために，統計的な処理を行う（必要に応じて）

【例】コインを 5 回投げたときの表の回数

1. 場面は「1 枚のコインを 5 回投げる」
   - コインを 1 回投げたときに表が 1/2 の確率で出ると仮定する
2. 手順は「コインを 1 回投げる」「結果を記録する」を 5 回くり返す
   - 「コインを投げる」＝「乱数を発生する」とする
   - 結果が表ならば「表が出た」回数をカウントするような関数を作成する
   - コインを 5 回投げ終わった後に「表が出た回数」を出力する
3. 「シミュレーションの実行」＝「R の関数の実行」
4. 関数の実行結果を吟味する

★ いきなり「コインを 5 回投げる」のは難しいので，まずは
「コインを 1 回投げる」ことからはじめる

コイン投げと乱数列

- コインを繰り返して投げ，結果を記録することを考える
  - 結果を一列に並べた数の列は以下の性質をもっている.
  - 等確率性：コインの投げる回数が 100 回，1000 回と大きくなるにつれて
    表と裏の出る割合はどれも 1/2 に近づく
  - 無規則性： 結果の列に規則性はない，何回目にサイコロを投げる場合でも
    表と裏が出る確率は全て同じ

    ⇒「表，裏，表，裏，表，裏，表，裏，表，裏，表，裏...」だと
    等確率性は満たしているが，無規則性は満たしていない
- 上記の 2 つの性質を満たすものを乱数列と呼ぶ

コイン投げと乱数列

- コイン投げの乱数列は「表」と「裏」がまんべんなく散らばっている
  ⇒「表が比較的多い」「裏はまとまって出やすい」ということは起こらない（これを一様という）
- この乱数列の一つ一つを乱数と呼び，この場合はコイン投げの結果（表・裏）が乱数となっている
- 乱数の例：
  - コイン投げの「表」「裏」
  - サイコロの「1」「2」「3」「4」「5」「6」
  - 宝くじ「ナンバーズ」の当選番号「12」「38」「7」・・・

R の乱数

- R の乱数（一様乱数）は等確率性と無規則性をほぼ満たす
- どうやって乱数を作っているのかを正確に説明するのは難しいので，ここでは「R がパソコンの中でサイコロを振っている」と考える
- R は内部で乱数を発生させる際にサイコロを 1 回振っている
- ただし普通の 6 面サイコロではなく，20 億面サイコロを振って乱数を発生させており，この 20 億面サイコロの目は

0，　1/20 億，　2/20 億，　・・・，　1999999999/20 億，　1

となっており，いずれかの目が等確率で出る

R の乱数

- このサイコロを振って，その目を出目（乱数）としている
- R では関数 runif() で乱数を生成することが出来る
- 0，1/20 億，2/20 億，・・・，1 から等確率で値を選ぶわけなので関数 runif() は「0 から1 の間の実数をランダムに選択している」と考えてよい

```
> runif(1)                  # 乱数を 1 個生成
[1] 0.7035544
> runif(5)                  # 乱数を 5 個生成
[1] 0.91822528 0.99971431 0.02933425 0.43168403 0.19871783
```

【例】コインを 1 回投げるシミュレーション

- コイン投げの乱数を作る関数
  - runif(1) の結果が 1/2 より大きい場合は「表」とする
  - runif(1) の結果が 1/2 より小さい場合は「裏」とする

```
> runif(1)                                  # 乱数を 1 個生成
[1] 0.7215244                               # コイン投げの「表」とする

> myfunc05 <- function() {
+   if (runif(1) > 0.5) return(1)           # コイン投げの「表」
+   else                return(0)           # コイン投げの「裏」
+ }
> myfunc05()
[1] 0          ← 裏が出た
> myfunc05()
[1] 1          ← 表が出た
```

【例】コインを x 回投げるシミュレーション

```
> myfunc06 <- function(x) {
+   a <- c()
+   for (i in 1:x) {
+     if (runif(1) > 0.5) a <- c(a, 1)   # コイン投げの「表」
+     else                a <- c(a, 0)   # コイン投げの「裏」
+   }
+   return(a)
+ }
> myfunc06(3)   # コイン投げ 3 回の結果をベクトルで保存
[1] 1 1 0
> myfunc06(5)   # コイン投げ 5 回の結果をベクトルで保存
[1] 1 1 0 1 0
```

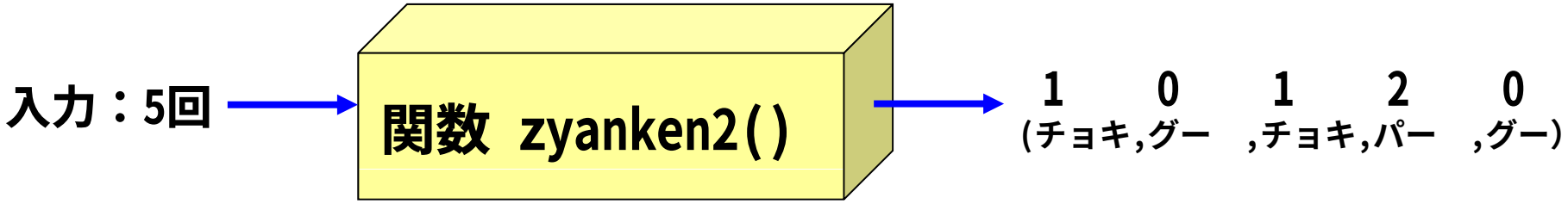
【例】じゃんけんを 1 回する関数

- じゃんけんのグー・チョキ・パーの乱数を作る関数の仕様：
  - runif(1) の結果が 0　～ 1/3 ならば「グー」とする
  - runif(1) の結果が 1/3 ～ 2/3 ならば「チョキ」とする
  - runif(1) の結果が 2/3 ～ 1　ならば「パー」とする

```
> zyanken1 <- function() {
+   x <- runif(1)
+   if      (x <= 1/3) te <- 1   # 1 : グー
+   else if (x <= 2/3) te <- 2   # 2 : チョキ
+   else               te <- 3   # 3 : パー
+   return(te)
+ }
> zyanken1()
[1] 2
```

【例】じゃんけんを x 回する関数

```
> zyanken2 <- function(x) {
+   a <- c()
+   for (i in 1:x) {
+     b <- runif(1)
+     if      (b <= 1/3) a <- c(a, 1)   # 1 : グー
+     else if (b <= 2/3) a <- c(a, 2)   # 2 : チョキ
+     else               a <- c(a, 3)   # 3 : パー
+   }
+   return(a)
+ }
> zyanken2(5)
[1] 3 1 2 3 1
```



【参考】規則性のあるベクトルを生成する関数

関数・コマンド	機能
1:5	1 から 5 までの公差が 1 の 等差数列を生成する
seq(1, 5, length=3)	長さ（要素の数が）3 の 1 から 5 までの等差数列を生成する
seq(1, 5, by=2)	1 から 5 まで 2 ずつ増加する等差数列を生成する
rep(1:5, times=3) rep(1:5, 3)	数列 1:5 を 3 個繰り返した数列を生成する
numeric(7)	0 を 7 個並べたベクトルを生成する
unique(x)	ベクトル x 中の反復した値を除いたベクトルを生成する

【参考】乱数を生成する関数

関数	機能
rbeta(10, shape1=2, shape2=3)	パラメータが (2, 3) であるベータ分布に従う乱数を 10 個生成する
rbinom(10, size=5, prob=0.3)	成功確率 0.3，試行回数 5 回である二項分布に従う乱数を 10 個生成する
rcauchy(10, location=2, scale=3)	パラメータが (2, 3) であるコーシー分布に従う乱数を 10 個生成する
rchisq(10, df=5, ncp=2)	自由度が 5，非心度が 2 である χ^2 分布に従う乱数を 10 個生成する
rexp(10, rate=1)	パラメータが 1 である指数分布に従う乱数を 10 個生成する

【参考】乱数を生成する関数

関数	機能
rlogis(10, location=2, scale=3)	パラメータが (2, 3) であるロジスティック分布に従う乱数を 10 個生成する
rmultinom(10, size=15, prob=c(0.1, 0.3, 0.6))	各確率が (0.1, 0.3, 0.6) である多項分布に従う乱数を 10 個生成する
rnbinom(10, size=5, prob=0.2)	「成功確率が0.2，5 回の成功が起こるまでの失敗の回数」である負の二項分布に従う乱数を 10 個生成する
rnorm(10, mean=0, sd=1)	平均が 0，標準偏差が 1 である正規分布に従う乱数を 10 個生成する
rpois(10, lambda=2)	パラメータが 2 であるポアソン分布に従う乱数を 10 個生成する

【参考】乱数を生成する関数

関数	機能
rf(10, df1=2, df2=3)	自由度が (2, 3) である F 分布に従う乱数を 10 個生成する
rgamma(10, shape=2, rate=3)	パラメータが (2, 3) であるガンマ分布に従う乱数を 10 個生成する
rgeom(10, prob=0.4)	成功確率が 0.4 である幾何分布に従う乱数を 10 個生成する
rhyper(10, m=6, n=3, k=2)	「白玉が 6 個，黒玉が 3 個入っている箱から玉を 2 個取り出したときの白玉の数」を 10 個生成する（超幾何分布）
rlnorm(10, meanlog=0, sdlog=1)	ログスケールで平均が 0，標準偏差が 1 である対数正規分布に従う乱数を 10 個生成する

【参考】乱数を生成する関数

関数	機能
rsignrank(10, n=10)	観測数が 10 である Wilcoxon の符号付順位和統計量の分布に従う乱数を 10 個生成する
rt(10, df=8)	自由度が 8 である t 分布に従う乱数を 10 個生成する
runif(10, min=0, max=1)	(0, 1) 区間の一様乱数を 10 個生成する
rweibull(10, shape=2, scale=1)	パラメータが (2, 1) であるワイブル分布に従う乱数を 10 個生成する
rwilcox(10, m=5, n=3)	観測数がそれぞれ (5, 3) である Wilcoxon の順位和統計量の分布に従う乱数を 10 個生成する

モンテカルロ・シミュレーションとは

- モンテカルロ・シミュレーション：
  乱数を用いたシミュレーションを何度も行なうことで，考えている問題の近似解を得る計算方法
- 手計算や理論的に解くことが出来ない問題であっても，多数回のシミュレーションを繰り返すことで下記の式で近似的に解を求めることが出来る

$$\frac{(1\text{回目の実験結果}) + \cdots + (n\text{回目の実験結果})}{n} \longrightarrow \left(\begin{array}{c} \text{実験結果の} \\ \text{正確な平均値} \end{array} \right)$$

モンテカルロ・シミュレーション①

- 例えば，「1 回のコイン投げ」を多数回繰り返し，
  結果を全て足し合わせたものをくり返し数 n で割り算すると，
  「1 回のコイン投げで表が出る割合・表が出る確率」となる
- くり返し数 n は大きければ大きいほど，
  より正確な「1 回のコイン投げで表が出る確率」となる
- ここでは 2 種類のモンテカルロ・シミュレーションについて考える
  1. 確率的な問題：コイン投げの問題
  2. 非確率的な問題：積分の近似解
- まずは「1 回のコイン投げ」を多数回繰り返し，「1 回のコイン投げで表が出る確率」を求めることを考えてみる（①の問題）

モンテカルロ・シミュレーション①

- モンテカルロシミュレーションを行う関数の雛形
  - コイン投げの結果を記録する変数 count を用意する
  - コインを投げて「表」が出たら変数 count に記録する
  - 最後に変数 count の平均値を算出 ⇒「表」が出る確率の近似値！

```
> mymontecarlo <- function(n) {
+   count <- 0          #   カウンタを 0 に戻す
+   for (i in 1:n) {
+                       ### シミュレーションを n 回くり返して
+                       ### 結果を count に記録する
+   }
+   return(count/n)     #   結果を出力
+ }
```

【例】 1 回のコイン投げで表が出る確率

- コイン投げをして「表：1 点」「裏：0 点」とする

1. コイン投げ x 回の合計点を求める関数 mymonte01() を作成する
2. 「合計点」を「コインを投げた回数」で割り算することで
   「表が出る回数の平均値」を求める
3. コイン投げ x 回の平均値を求める関数 mymonte02() を定義した後
   関数 mymonte02() のくり返し数を 1, 2, 3, ・・・, n として
   mymonte02() を実行したときの結果を保存した上で，結果を
   プロットする関数 mymonte03() を作成する

※ くり返し数は 100 回にする

【例】 1 回のコイン投げで表が出る確率

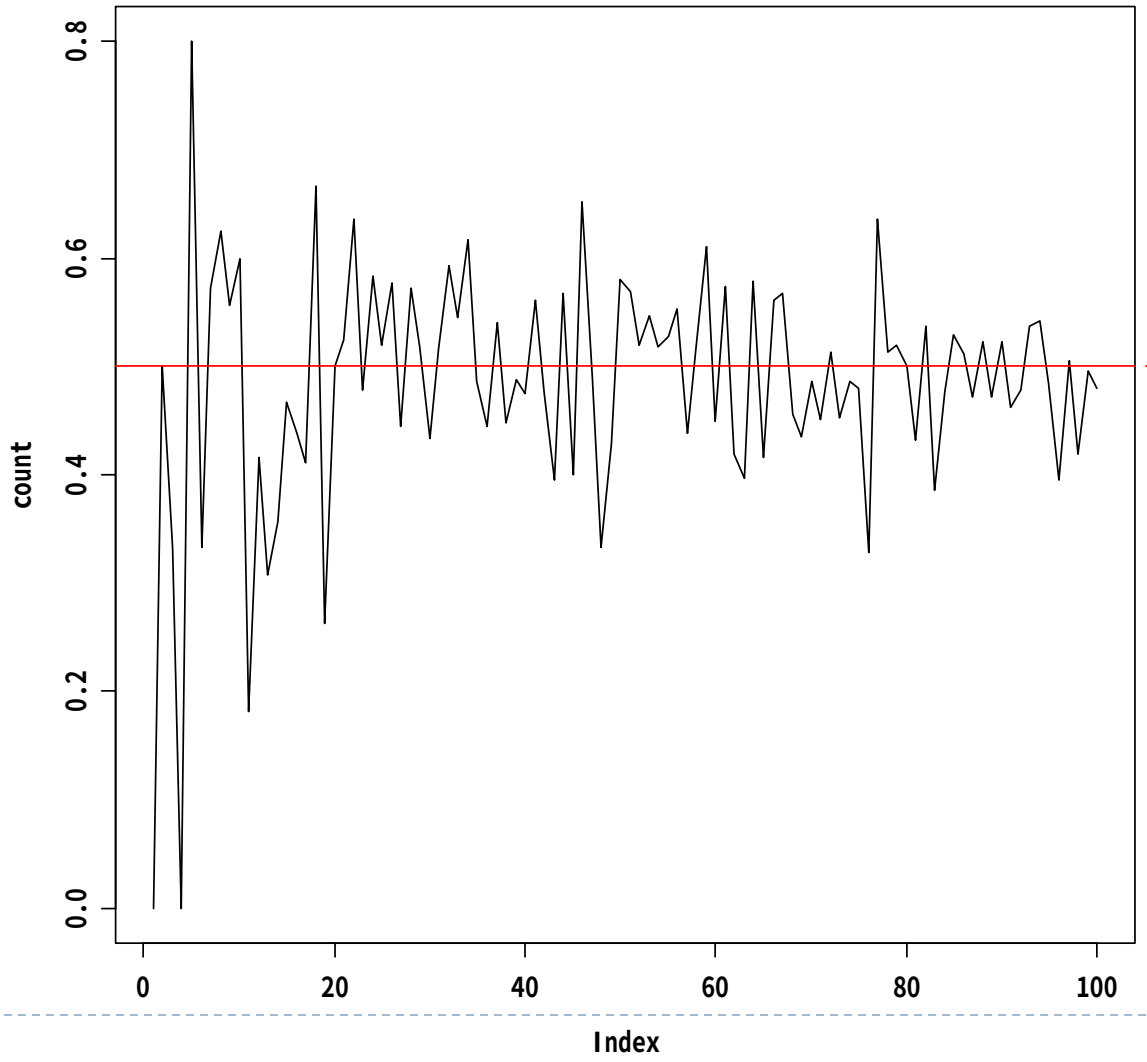
```
> mymonte01 <- function(n) {
+   count <- 0
+   for (i in 1:n) {
+     if (runif(1) > 0.5) count <- count+1  # コイン投げの「表」
+   }
+   return(count)
+ }
> mymonte01(100)/100
[1] 0.48
```

⇒ コインを投げたときに「表」が出る確率は 0.5 であり，
　モンテカルロ・シミュレーションの結果は 0.5 に近くなっている

【例】 1 回のコイン投げで表が出る確率

```
> mymonte02 <- function(n) {
+   count <- 0
+   for (i in 1:n) {
+     if (runif(1) > 0.5) count <- count+1  # コイン投げの「表」
+   }
+   return(count / n)
+ }
> mymonte03 <- function(n) {
+   count <- c()
+   for (i in 1:n) {
+     count <- c(count, mymonte02(i))
+   }
+   plot(count, type="l")
+ }
> mymonte02(100)
[1] 0.55
> mymonte03(100)
> abline(h=0.5, col="red")
```

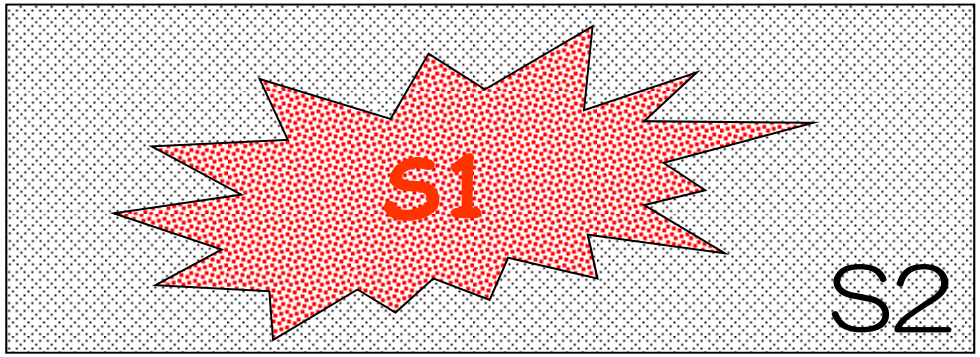
【例】 1 回のコイン投げで表が出る確率



徐々に **0.5** に
近づいている！

モンテカルロ・シミュレーション②

- 計算が面倒な面積を求める場合にもモンテカルロ・シミュレーションを使って近似解を得ることが出来る



1. 2 変数乱数（x, y）を算出する ⇒ イメージは「砂粒を落とす」
2. 1. で求めた点が S1 に含まれている場合はカウントする
3. 2. を多数回繰り返し，平均値（S1に乗った砂粒の割合）を計算する

$$\text{面積 } S_1 = \frac{\text{領域 } S_1 \text{の上に乗っている砂粒の数}}{\text{砂粒の合計数}} \times \text{面積 } S$$

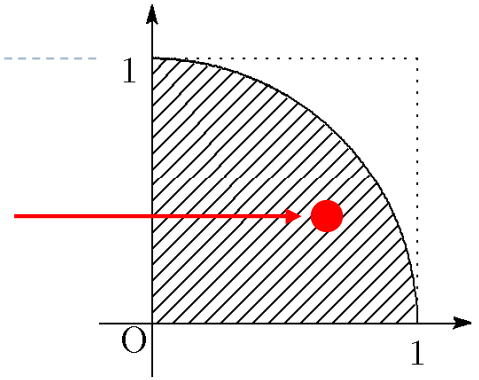
モンテカルロ・シミュレーション②

- モンテカルロシミュレーションを行う関数の雛形
  - 結果を記録する変数 count を用意する
  - 乱数（砂粒）を生成して求める領域に入ったら変数 count に記録
  - 最後に変数 count の平均値を算出 ⇒「表」が出る確率の近似値！

```
> mymontecarlo <- function(n) {
+   count <- 0                  #   カウンタを 0 に戻す
+   for (i in 1:n) {
+                               ### 砂粒を n 個敷き詰め，
+                               ### ある領域に含まれる砂粒の
+                               ### 数を count に足し算する
+   }
+   return(count / n)           #   結果を出力
+ }
```

【例】円周率の計算

1. 2 変数乱数（x, y）を算出する
2. 1.で求めた点が半円に含まれている場合はカウント
3. 平均値（S1に乗った砂粒の割合）を計算する

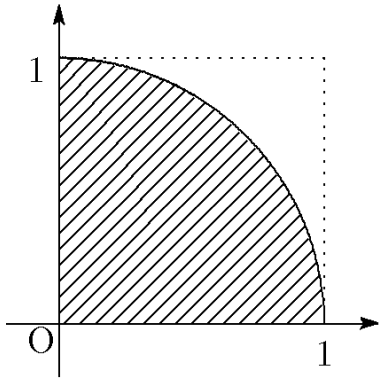



```
> pi.montecarlo <- function(n) {
+   a <- 0                                    # カウントを 0 に
+   for (i in 1:n) {
+     b <- runif(2)                           # 変数 b は
+     if (sqrt(b[1]^2 + b[2]^2) < 1) {        # (x 座標, y 座標)
+       a <- a + 1                            # のベクトルと
+     }                                       # なっている
+   }
+   return(4 * a / n)                         # πを求める
+ }
```

【例】円周率の計算

- n は大きければ大きいほど，より正確な近似解が得られる

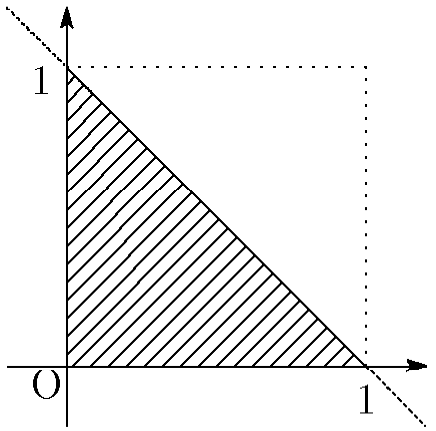
```
> pi.montecarlo(10)
[1] 3.6
> pi.montecarlo(100)
[1] 3.08
> pi.montecarlo(1000)
[1] 3.136
> pi.montecarlo(10000)
[1] 3.1528
> pi.montecarlo(100000)
[1] 3.14612
```



【例】積分計算

- f(x) = 1 - x を 0 ~ 1 で積分するモンテカルロ・シミュレーション を行う関数

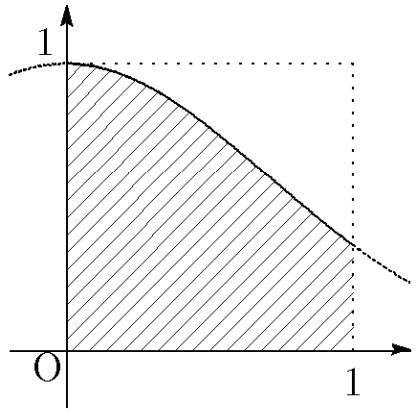
```
> tri.montecarlo <- function(n) {
+   a <- 0                      # カウントを 0 に
+   for (i in 1:n) {
+     b  <- runif(2)            # 変数 b は（x 座標，y 座標）
+     if (1-b[1] < b[2]) {      # y 座標が「1 - x 座標」より小さければ
+       a <- a + 1              # 求める領域に入っているのでカウント
+     }
+   }
+   return(a / n)
+ }
> tri.montecarlo(10000)
[1] 0.5022
```



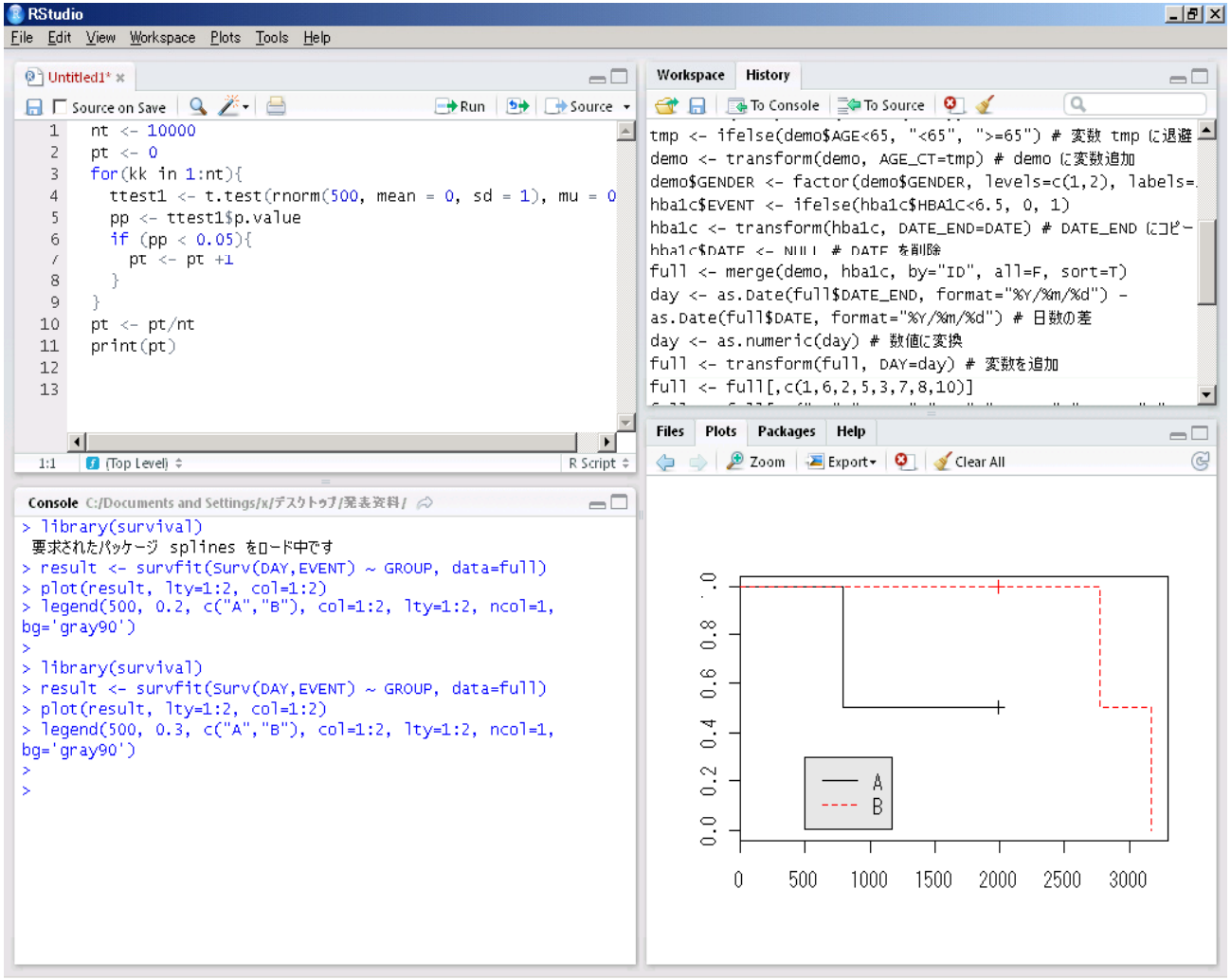
【例】積分計算

- $f(x) = \exp\{-x^2\}$ を 0 ~ 1 で積分するモンテカルロ・シミュレーションを行う関数

```
> exp.montecarlo <- function(n) {
+   a <- 0                          # カウントを 0 に
+   for (i in 1:n) {
+     b  <- runif(2)                # 変数 b は (x 座標, y 座標)
+     fx <- exp(-b[1]^2)            # f(x) の値
+     if (b[2] < fx) {              # b の y 座標が f(x) よりも小さければ,
+       a <- a + 1                  # 求める領域に入っているのでカウントする
+     }
+   }
+   return(a / n)
+ }
> exp.montecarlo(10000)
[1] 0.745
```



【宣伝】R の 統合開発環境「RStudio」



- フリーで日本語対応
- プログラム作成にもってこい
- 対応 OS：
  - Windows XP/Vista/7
  - Mac OS X 10.5+
  - Debian6+/ Ubuntu 10.04+ (32-bit, 64-bit)
  - Fedora13+/ openSUSE 11.4+ (32-bit, 64-bit)
- 詳しくは・・・ http://rstudio.org/

本日のメニュー

1. ベクトル
2. 関数の作成方法
3. 条件分岐とくり返し
4. シミュレーション

参考文献

- 統計学（白旗 慎吾 著，ミネルヴァ書房）
- 乱数の知識（脇本 和昌 著，森北出版株式会社）
- モンテカルロ法（宮武 修，中山 隆 著，日刊工業新聞社）
- The R Tips 第 2 版（オーム社）
- R 流！イメージで理解する統計処理入門（カットシステム）

Rで統計解析入門

終