R で学ぶデータ解析とシミュレーション

③

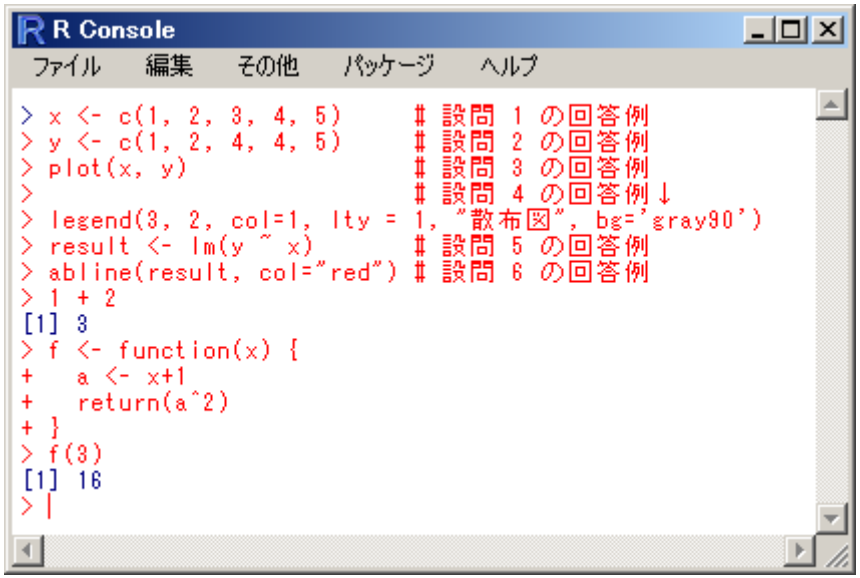
～ プログラミング＆シミュレーション入門 ～

３時間目のメニュー

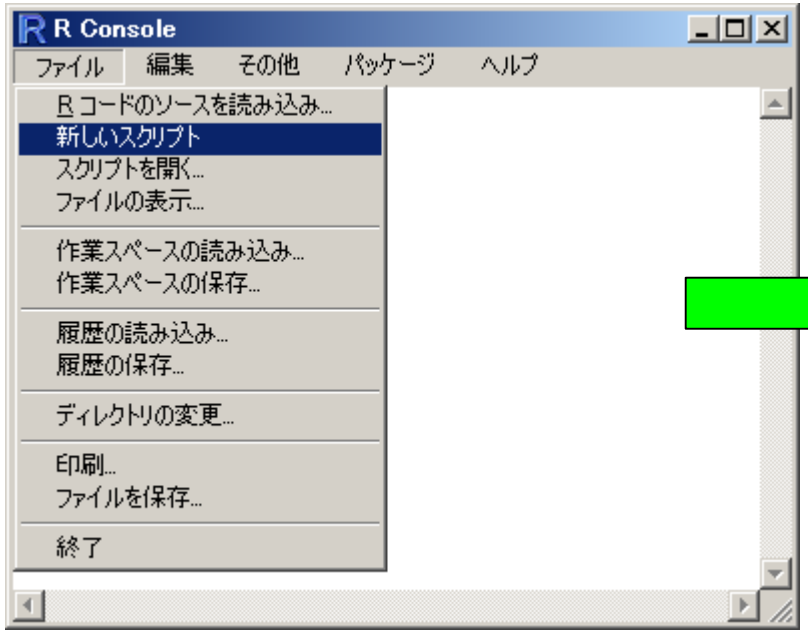
- R 専用エディタの紹介 ←
  - R 専用エディタを使用して 2 時間目の演習問題を解く
- プログラミング＆シミュレーション入門
  - プログラミング入門
    - 条件分岐〔 if 〕
    - くり返し〔 for 〕
  - シミュレーション入門
    - コイン投げと乱数
    - モンテカルロ・シミュレーション

R 専用エディタ

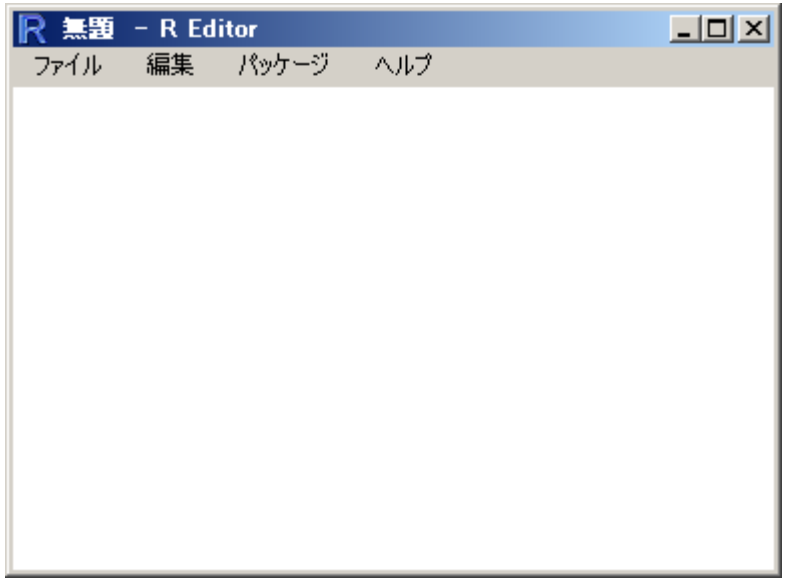
- R は一行ずつ命令を実行していく手順
  ⇒ 記述を間違うとその時点でエラーとなり，最初から記述をやり直さなければいけない
- R 専用のエディタを使えば，ある程度命令を書きためた後，命令を一度に実行することが出来る！



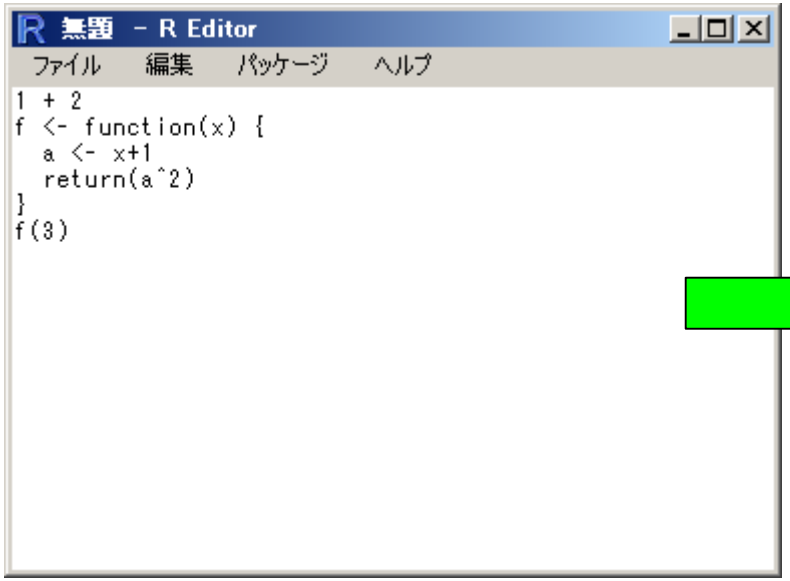
R 専用エディタ



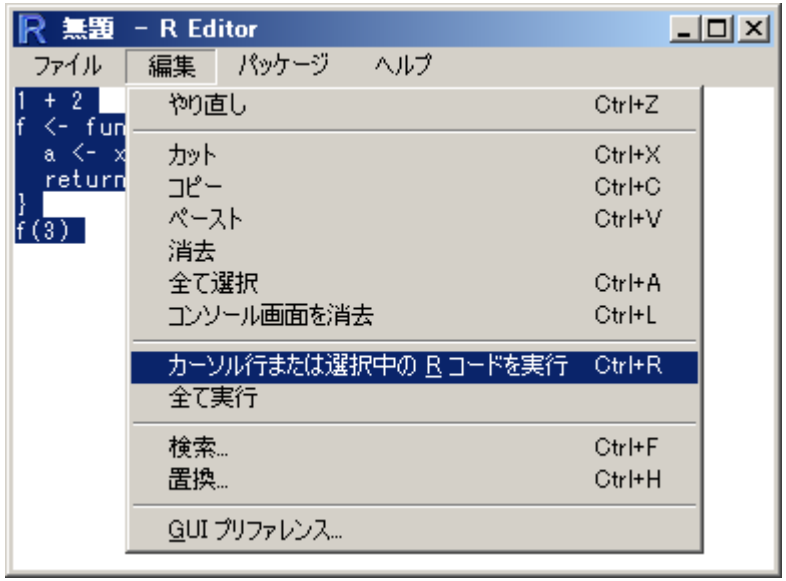
「ファイル」→「新しいスクリプト」を選択



R 専用エディタ

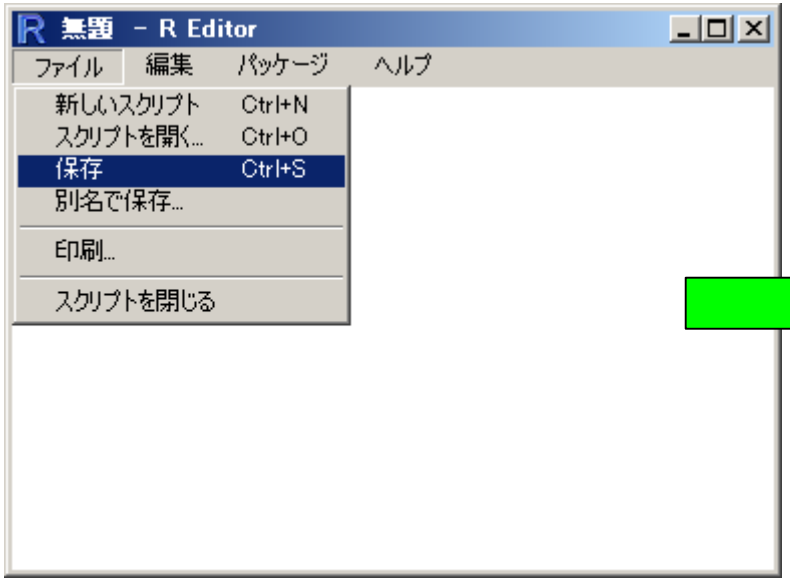


R の命令を記述した後,
「編集」→「・・・実行」
を選択して命令を実行する



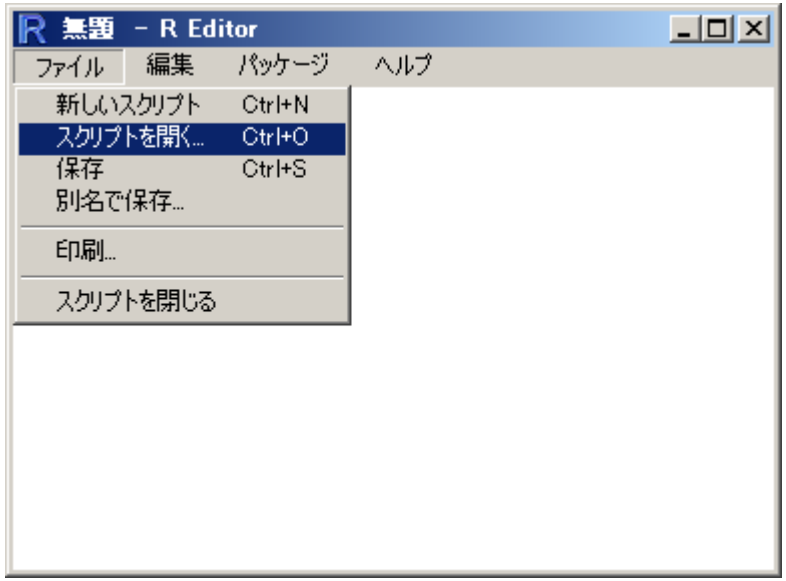
1. カーソル行または選択中の R コードを実行：選択した部分の命令を実行
2. 全て実行：記述した命令を全て実行

R 専用エディタ



記述した命令をファイルに保存することも出来る

⇒ 拡張子は「 **.R** 」で保存



保存したファイル（拡張子「 **.R** 」 ）は「ファイル」⇒「スクリプトを開く」で開くことが出来る

【演習】

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

３時間目のメニュー

- R 専用エディタの紹介
  - R 専用エディタを使用して 2 時間目の演習問題を解く
- プログラミング＆シミュレーション入門 ←
  - プログラミング入門
    - 条件分岐〔 **if** 〕
    - くり返し〔 **for** 〕
  - シミュレーション入門
    - コイン投げと乱数
    - モンテカルロ・シミュレーション

プログラミング入門

- 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
```

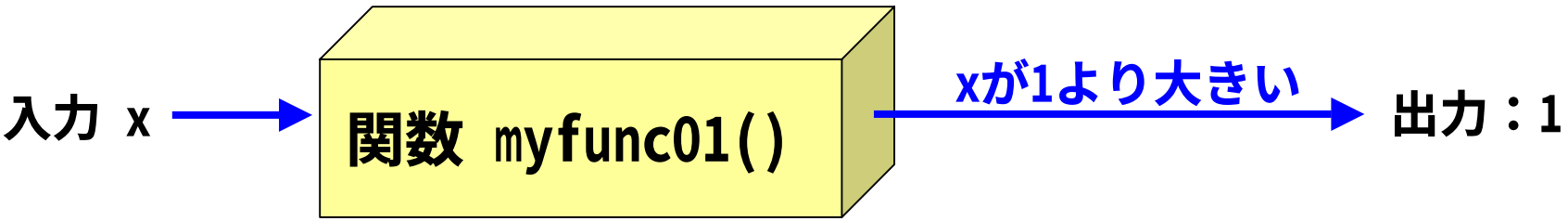
比較演算子

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

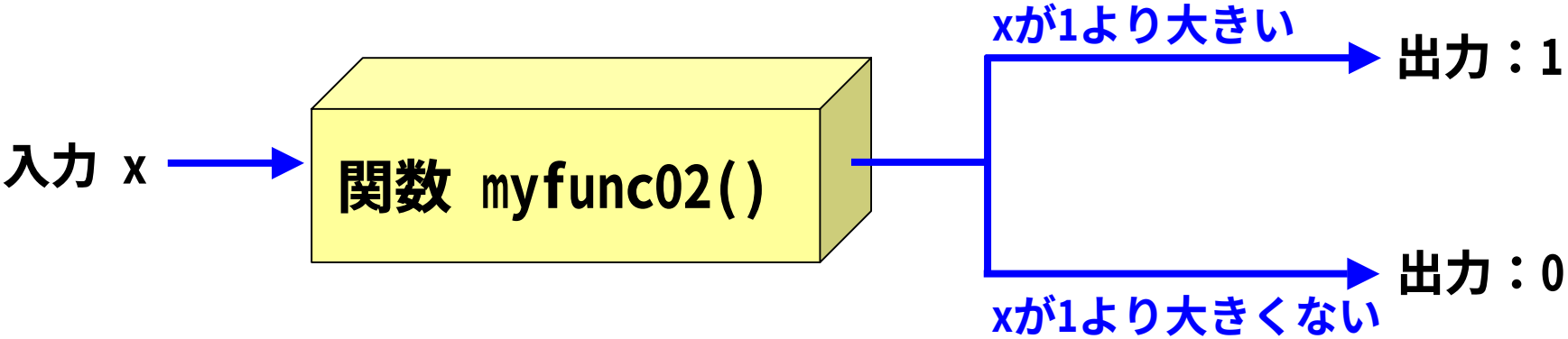
```
> myfunc01 <- function(x) {
+   if (x > 1) return(1)    # 引数 x が 1 よりも大きい場合は 1
+ }
> myfunc01(2)
[1] 1
> myfunc01(0)               # 何も起こらない
```



〔 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

３時間目のメニュー

- R 専用エディタの紹介
  - R 専用エディタを使用して 2 時間目の演習問題を解く
- プログラミング＆シミュレーション入門
  - プログラミング入門
    - 条件分岐〔 if 〕
    - くり返し〔 for 〕
  - シミュレーション入門 ←
    - コイン投げと乱数
    - モンテカルロ・シミュレーション

シミュレーションとは

- 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」をパソコン上で 5 回繰り返す
   - 「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「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」がまんべんなく散りばめられている
  ⇒「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だから関数 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

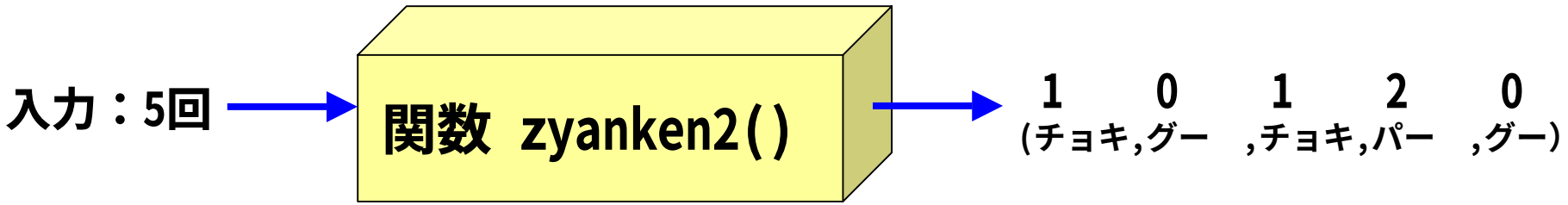
```
> runif(1)                                  # 乱数を 1 個生成
[1] 0.7215244                               # コイン投げの「表」とする

> 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
```

【演習】

1. 1 回だけじゃんけんをする関数を作成してください
2. x 回だけじゃんけんをする関数を作成してください
   - 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



【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返す

【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

３時間目のメニュー

- R 専用エディタの紹介
  - R 専用エディタを使用して 2 時間目の演習問題を解く
- プログラミング＆シミュレーション入門
  - プログラミング入門
    - 条件分岐〔 if 〕
    - くり返し〔 for 〕
  - シミュレーション入門
    - コイン投げと乱数
    - モンテカルロ・シミュレーション ←

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のこと
- 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

- 「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 で割り算したものが「１回のコイン投げで表が出る割合・表が出る確率」

★ スライド 38 頁の演習問題：

- コイン投げをして「表：1点」「裏：0点」とします
  「合計点」を「コインを投げた回数」で割り算してください
  ⇒ 割り算することで，表が出る確率の近似解が得られる！

- 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

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
+ }
```

【演習】

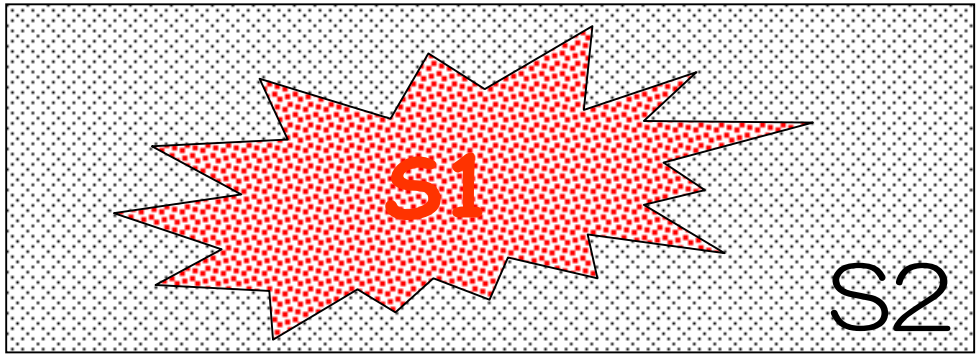
- コイン投げをして「表：1点」「裏：0点」とします

1. コイン投げ x 回の合計点を求める関数 mymonte01() を作成してください
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てください
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() を作成してください

※ くり返し数は 100 回にしてください

モンテカルロ・シミュレーション②

■ 計算が面倒な面積（S1）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

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

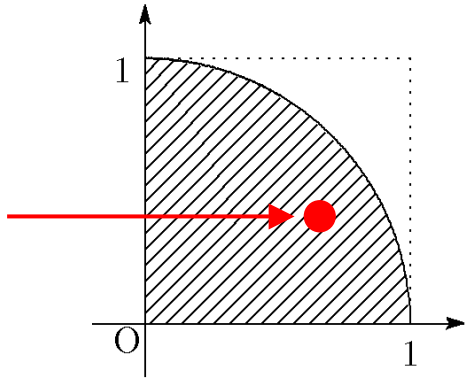
モンテカルロ・シミュレーション②

- 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

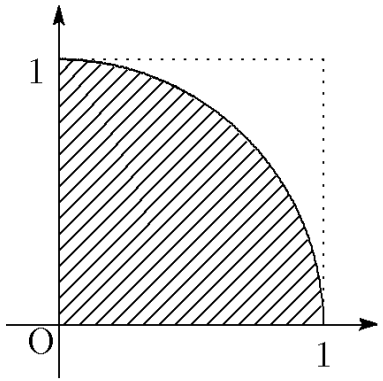



```
> pi.montecarlo <- function(n) {
+   a <- 0                                  # カウントを 0 に
+   for (i in 1:n) {
+     b <- runif(2)                         # 変数 b は
+     if (sqrt(b[1]^2 + b[2]^2) < 1) {      # (x 座標, y 座標)
+       a <- a + 1                          # のベクトルと
+     }                                     # 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

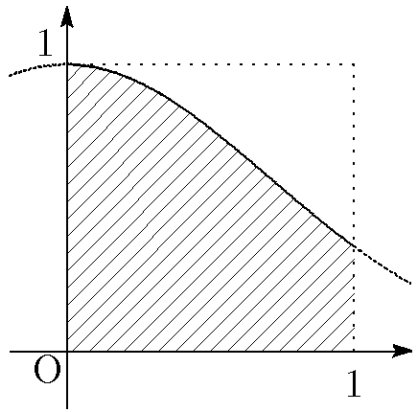
```
> pi.montecarlo(10)
[1] 3.6
> pi.montecarlo(100)
[1] 3.08
> pi.montecarlo(1000)
[1] 3.136
> pi.montecarlo(10000)
[1] 3.1528
> pi.montecarlo(100000)
[1] 3.14612
```



【例】積分計算

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

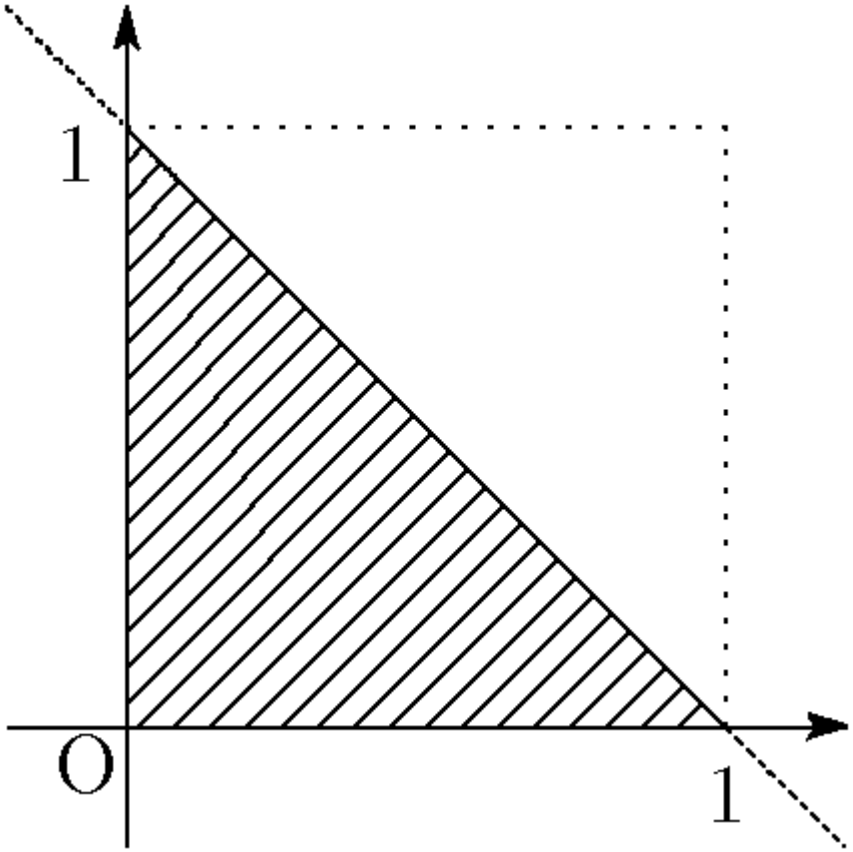
```
> exp.montecarlo <- function(n) {
+   a <- 0                      # カウントを 0 に
+   for (i in 1:n) {
+     b  <- runif(2)            # 変数 b は (x 座標, y 座標)
+     fx <- exp(-b[1]^2)        # f(x) の値
+     if (b[2] < fx) {          # b の y 座標が f(x) よりも小さければ,
+       a <- a + 1              # 求める領域に入っ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



【演習】

- f(x) = 1-x を 0 から 1 まで積分するモンテカルロ・シミュレーション関数を定義してください

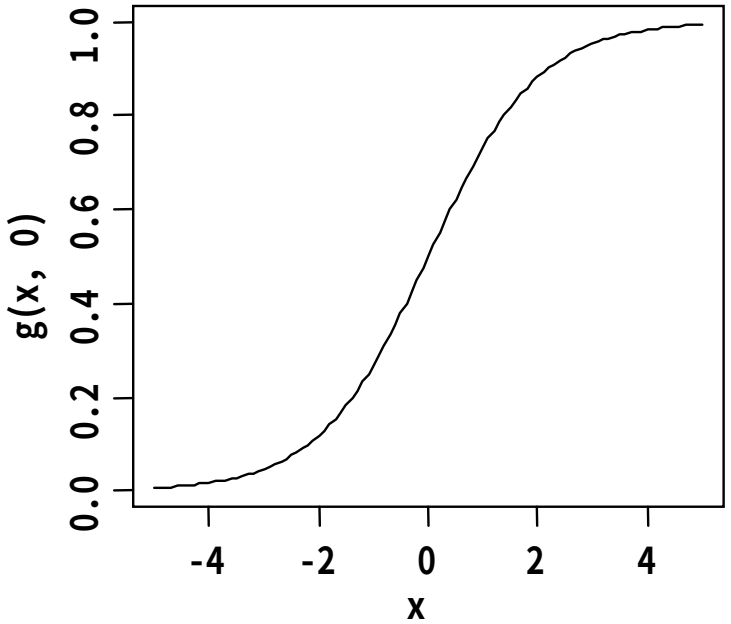
  ※ くり返し数は 100 回にしてください



★演習（7枚目のスライド分）の回答例

■ 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
```

★演習（28枚目のスライド分）1. の回答例

```
> # ジャンケンを 1 回行う関数
> 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
```

★演習（28枚目のスライド分）2. の回答例

```
> # ジャンケンを x 回行う関数
> 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
```

★演習（38枚目のスライド分）1～2 の回答例

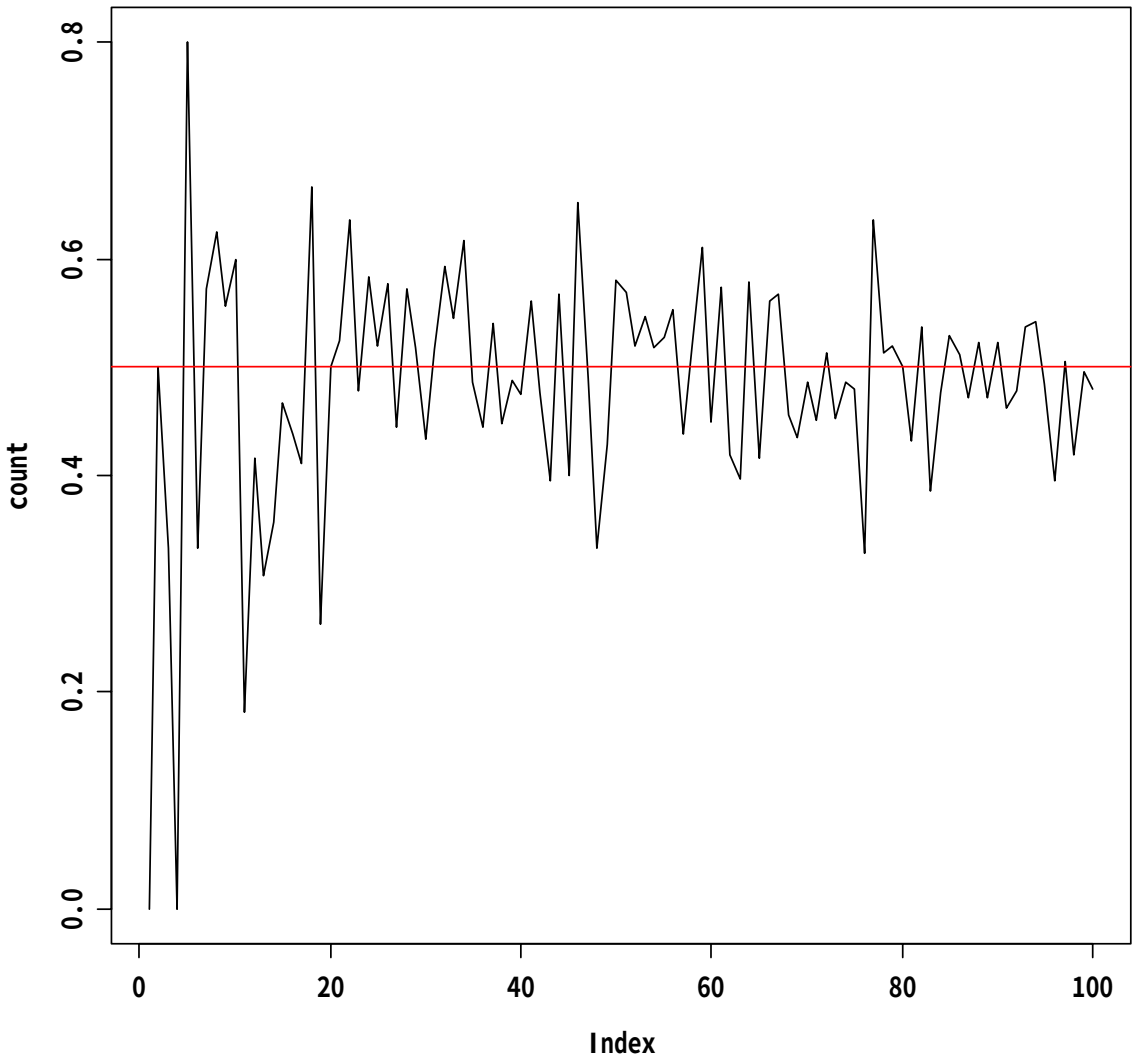
```
> mymonte01 <- function(n) {
+   count <- 0
+   for (i in 1:n) {
+     if (runif(1) > 0.5) count <- count+1  # コイン投げの「表」
+   }
+   return(count)
+ }
> mymonte01(100)/100
[1] 0.48
```

⇒ 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

★演習（38枚目のスライド分）3. の回答例

```
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)
mymonte03(100)
abline(h=0.5, col="red")
```

★演習（38枚目のスライド分）3. の回答例



徐々に **0.5** に
近づいている！

★演習（44枚目のスライド分）の回答例

```
> 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
```

３時間目にやったこと

- R 専用エディタの紹介
  - R 専用エディタを使用して 2 時間目の演習問題を解く
- プログラミング＆シミュレーション入門
  - プログラミング入門
    - 条件分岐〔 if 〕
    - くり返し〔 for 〕
  - シミュレーション入門
    - コイン投げと乱数
    - モンテカルロ・シミュレーション

終