

## R 超入門

櫻井彰人

### 前書: プログラムは何に使うか?

- ・アイデアが正しいかどうかを調べる
  - 数式
  - データ間の関係
- ・グラフを書く

- ・動かす
- ・記録する
- ・見せる

### 本日の目標

- ・R言語の概要を知る。
  - 説明を聞く
  - 書いてみる
  - 実行させてみる
  - 説明書を見る

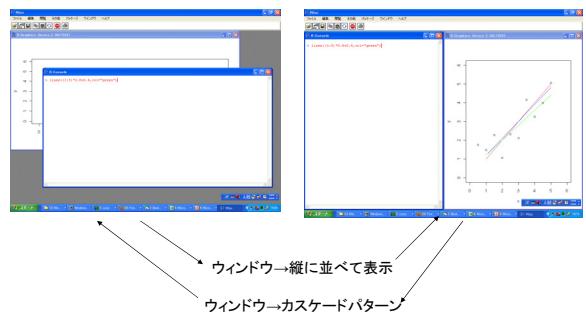
### Rをつかう

- ・インストール
  - 参考: RjpWiki→Rのインストール
- ・起動
  - デスクトップのアイコンをダブルクリック
  - スタート→すべてのプログラム→R
- ・終了
  - 右上の×ボタン→質問に「いいえ」

### 目次

- ・前書
- ・基本データ型 – ベクトル
  - 作る、代入、読み出し、加算、関数
- ・識別子、ヘルプ
- ・プロット
- ・関数の引数 – プロットを例にして
- ・再びプロット
- ・関数について – 少し丁寧に
- ・繰り返しについて
- ・補足: 結果の保存

### Rのウィンドウ



## 目次

- 前書
- 基本データ型 – ベクトル
  - 作る、代入、読み出し、加算、関数
- 識別子、ヘルプ
- プロット
- 関数の引数 – プロットを例にして
- 再びプロット
- 関数について – 少し丁寧に
- 繰り返しについて
- 補足: 結果の保存

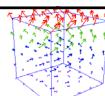
練習問題:

- 1) 次の要素を持つベクトルを作りなさい  
45,5,12,10

- 2) 次のコマンドの結果、何が起こるか?  
c(1:100)

(注) ベクトルは、Rにおいて極めて基本的なデータ構造である。R 中の殆どすべてのものは、何らかの意味でベクトルである。勿論、一般には高次元のベクトルである、すなわち、ベクトルのベクトルである

## ベクトル



- R で扱えるデータ構造の基本要素にベクトルがある
- ベクトルは、ご存じのように、数値や文字列の列である。
  - ("altec", "sony", "jbl")
  - (1, 5.2, 10, 7, 2, 21)
  - (3)
- 最後の例は、長さ1のベクトルである

<http://www.hulinks.co.jp/software/vxler/images/WebVectorPlot.png>

## 代入

irb とそっくり  
irb との違いに注意 もっとも、=も使えます

```
> 4+5 # 4 と 5 の加算
[1] 9
> a<-4 # 変数 a に 4 を代入
> b<-5 # 変数 b に 5 を代入
> a # 変数 a の値を印字
[1] 4
> b
[1] 5
> a+b # 変数の加算 a+b (4+5)
[1] 9 # 正しい!
# の右は、すべてコメント
```

## ベクトルを「作る」には

メソッドの名称 メソッドの引数  
> c(1,5,10,7,2,1) メソッド実行の結果  
[1] 1 5 10 7 2 1

> c("altec", "sony", "jbl")  
[1] "altec" "sony" "jbl"

文字列は、その左右を、ダブルクオート("")かシングルクオート('')で囲む

長さ1のベクトルを作るときは、c() を書かなくてよい  
> 3  
[1] 3

## ベクトル要素の読み出し

ベクトル要素を取り出すには次のようにする。Rubyと異なる

a[5] # 第5要素。先頭は第1要素！

次のようにしたら、どうなる？

a <- c(1,5,10,15,20)

a[c(1,3,5)]

a[1:4]

a[4:1]

即レポ1-1

## 練習

- ベクトルを反転させる(ベクトル要素の並びを左右逆転する)、少なくとも2通りの方法を考えて(探して)下さい。

```
a <- c(1, 5, 10, 15, 20)      # ベクトルの定義  
                                # で?
```

## 目次

- 前書
- 基本データ型 – ベクトル
  - 作る、代入、読み出し、加算、関数
- 識別子、ヘルプ
- プロット
- 関数の引数 – プロットを例にして
- 再びプロット
- 関数について – 少し丁寧に
- 繰り返しについて
- 補足: 結果の保存

## ベクトルへの加算

- V1 に 1から20の整数からなる配列を代入後  
> v1 + 10  
としたらどうなるか?
- 次のものを試してみよう。そして、その規則を推測してください。

```
> a <- c(10, 20, 30, 40)  
> b <- c(1, 2, 3, 4)  
> c <- c(1, 1, 2, 2)  
> d <- c(1, 2, 3)  
> a+b  
> a+c  
> a+d
```

## 識別子と名前

- 識別子(identifier)は、  
Identifiers consist of a sequence of letters, digits, the period ('.') and the underscore. They must not start with a digit nor underscore, nor with a period followed by a digit.
- つまり、文字、数字、ピリオド、下線のいずれからなり、先頭は文字(少し例外あり)。
- 名前(names)には、もっと自由な文字列が使えるが、その値の読み書きには get と assign を使う

## ベクトルに関する関数

- ベクトルに関する関数はいくつかある。名称は論理的につけられている。例えば、
  - length(): ベクトルの要素数
- そこで、練習問題
  - 要素の値が、1から10000までの整数である配列を作れ(a:b構文を用いる)
  - 要素のすべての和(sum)を求めよ
  - 要素の平均値(mean)を求めよ

即レポ1-2

## ヘルプ

- R には関数が多すぎる(それがメリットなのだが)。そのため、ヘルプが結構充実している。
  - その使い方は
    - ?関数名 # または  
help(関数名) # または  
?"関数名" # または  
help("関数名") # または  
help.search(何か) # 'fuzzy' 検索
- 練習問題
- 関数 sample() と sort() のヘルプを調べ、1から10000を要素とするベクトルに(結果を予想して)適用しなさい
  - ところで、RSiteSearch("sample") とすると何が起こるか?

## 目次

- 前書
- 基本データ型 – ベクトル
  - 作る、代入、読み出し、加算、関数
- 識別子、ヘルプ
- プロット
- 関数の引数 – プロットを例にして
- 再びプロット
- 関数について – 少し丁寧に
- 繰り返しについて
- 補足: 結果の保存

## プロット・オプション

- 次のようにオプションが指定できる

plot(あるベクトル, col="blue")

要は、関数の引数！

基本的なオプションとしては、次のようなものがある。複数個指定が可能

```
type = "l", "b"  
col = "pink"  
main = "important plot"
```

構文規則では、これらのオプションは、plot() の引数である

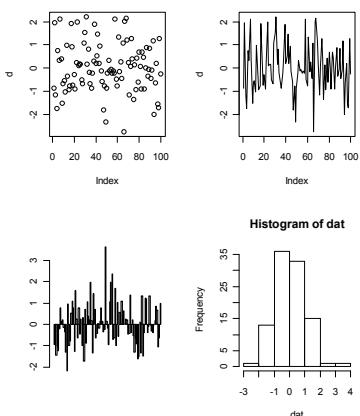
## ベクトルのプロット

- まずプロットするデータを作る。
- ```
d <- rnorm(100) # 100 個の正規乱数
```
- 次のを試してみよう:  
`plot(d)` プロットする plot も関数である  
`plot(d, type="l")`  
`barplot(d)`  
`hist(d)`
  - 何が図示されているか、確認してください。

次ページのようにするには、plot の前に、一度、`par(mfrow=c(2,2))` として下さい

## 目次

- 前書
- 基本データ型 – ベクトル
  - 作る、代入、読み出し、加算、関数
- 識別子、ヘルプ
- プロット
- 関数の引数 – プロットを例にして
- 再びプロット
- 関数について – 少し丁寧に
- 繰り返しについて
- 補足: 結果の保存



## 引数について

関数は、大抵の場合、何らかの入力が必要となる。  
例えば、`plot(d)`。ここで 'd' は名前なし(無名)引数であり、  
これがうまくいくのは、`plot()` では、第一引数が x 値('y' 値)の書  
き間違いではない)であると仮定しているからである。  
`plot()` の場合は、名前付き引数も使える。この場合は、  
`plot(x=d)` とすればよい(`plot(d)` と同じ結果)。  
「x=d」のように書く引数を名前付き引数という(xが名前である)。  
引数の個数が多いときには、多くの引数が(誤りを避けたり、  
default値を用いるようにするため)名前付き引数となっ  
いる。  
`plot( あるベクトル, col="blue", type="s" )`

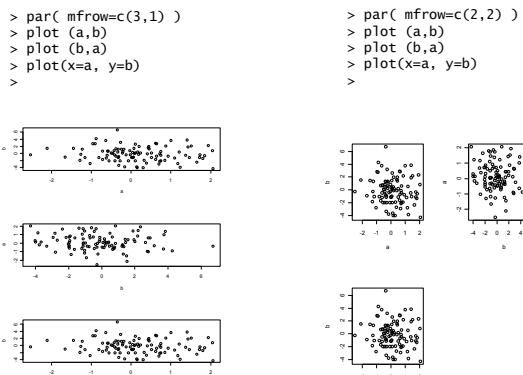
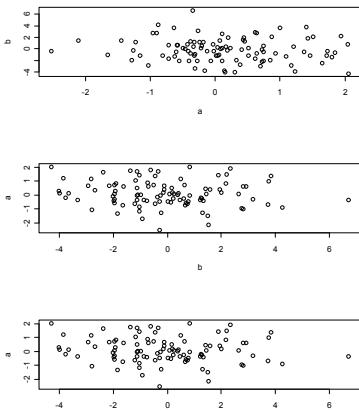
## 名前なし引数は順序が大切

- 名前なし引数は、その順序で、意味が決まっている(勿論、関数ごとに)
  - "普通"ですね。
- 次の違いを知ろう

```
a <- rnorm(100)
b <- rnorm(100)*2
plot(a,b)
plot(b,a)
plot(x=b, y=a)
```

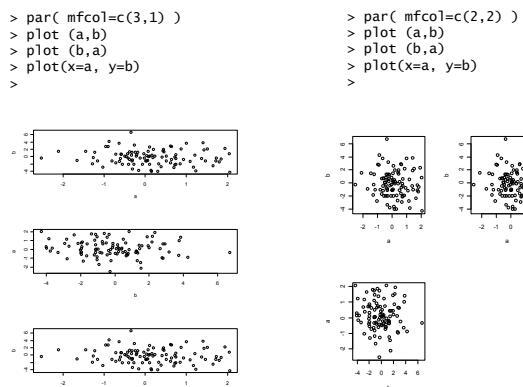
## plot()用パラメータ

- 引き続くplot()に共通なパラメータを、par()で設定することができる。非常にたくさんのパラメータがある試しに、?parとしてみてください。
- 例えば:  
mfrow()とmfcol(); 一ページ(ウィンドウ一個)に複数のプロットするためのもの。配置方向と個数を示す。
  - 長さ2のベクトルを値とする。それにより一ページ中の、プロットするセルの個数が決まる(次頁)



## 目次

- 前書
- 基本データ型 – ベクトル
  - 作る、代入、読み出し、加算、関数
- 識別子、ヘルプ
- プロット
- 関数の引数 – プロットを例にして
- 再びプロット**
- 関数について – 少し丁寧に
- 繰り返しについて
- 補足: 結果の保存



即レポ1-3

## 課題

- 一行に3個のグラフをプロットしてください

## 軸の上下限

- x軸、y軸の上下限を変更するには、

x軸の上下限: `xlim = c( 開始点, 終点 )`  
y軸の上下限: `ylim = c( 開始点, 終点 )`

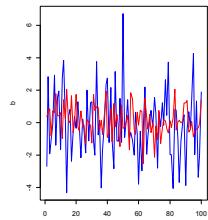
## グラフの重ね書き

- 一つのグラフの上に他のグラフを重ねたい(2個や3個のグラフを重ねがきしたい)場合には、例えば、`lines()` や `points()` を使えばよい。

```
> plot(b, type="l", col="blue")
> lines(a, col="red")
>
```

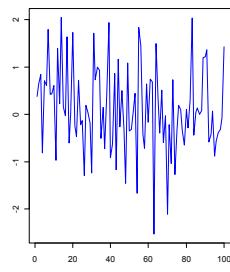
`a<-rnorm(100)`

`b<-rnorm(100)*2`



## 例

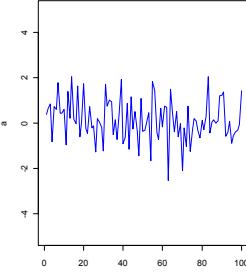
`plot(a, type="l", col="blue")`



`a<-rnorm(100)`

`b<-rnorm(100)*2`

`plot(a, type="l", col="blue", ylim=c(-5,5))`



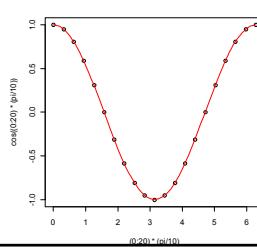
`a<-rnorm(100)`

`b<-rnorm(100)*2`

## プロットの重ね書き

- コマンドによっては、`add=T` をパラメータとすることで、重ねがきができます。

```
> plot( (0:20)*(pi/10),cos( (0:20)*(pi/10) ) )
> curve(cos,0,2*pi,col=2, add=T)
>
```



## 目次

- 前書
- 基本データ型 – ベクトル
  - 作る、代入、読み出し、加算、関数
- 識別子、ヘルプ
- プロット
- 関数の引数 – プロットを例にして
- 再びプロット
- 関数について – 少し丁寧に
- 繰り返しについて
- 補足: 結果の保存

## 関数の定義

• 何度も(そつくりなことを)行う場合、それを関数として定義しておくのが妥当

例: 引数を  $x$  とするとき、 $x^2+x+1$  を返す関数

```

    関数名: 適宜      関数であることを宣言
    my_f<- function(x) {
        + }           引数、関数の定義本体で使用する
        return(x^2 + x + 1)
    }               値を、関数の値とし、この関数の計算を終了する
    {}             {}で括られた部分が、関数の本体の定義
  
```

[http://upload.wikimedia.org/wikipedia/commons/thumb/3/3b/Function\\_machine2.svg/159px/Function\\_machine2.svg.png](http://upload.wikimedia.org/wikipedia/commons/thumb/3/3b/Function_machine2.svg/159px/Function_machine2.svg.png)

## 関数の値(戻り値、返り値)

- R では、関数本体内で、最後に実行した式の結果が、関数の値として、戻される
- ただし、`return()` という関数を使うと、`return` 関数引数の演算結果を値(return value, 戻り値、返り値)として、呼び出し元に戻る

## 関数の使用例

```

> myF <- function(x) {
+   return(x^2 + x + 1)
+ }
> myF(10)
[1] 111
> myF(20)
[1] 421
>
  
```

```

myF <- function(x) {
  +   return(x^2 + x + 1)
+ }
myF(10)
myF(20)
  
```

## 「戻り値」の例

```

> myF <- function(x){
+   x^2 + x + 1
+ }
> myF(10)
[1] 111
> myF(20)
[1] 421
>

  
```

```

myF <- function(x){
  +   v <- x^2 + x + 1
+   return( v )
}
myF(10)
myF(20)
  
```

```

> myF <- function(x){
+   v <- x^2 + x + 1
+   return( v )
}
> myF(10)
[1] 111
> myF(20)
[1] 421
>
  
```

即レポ1-4

## 練習問題

- $x^3+x^2+x+1$  を計算する関数を作りなさい。正しいことを検証しなさい。
- $x$  が与えられたとき、 $1+2+\dots+x$  を計算する関数を作りなさい。正しいことを検証しなさい。
- $x$  と  $y$  が与えられたとき、 $x+(x+1)+\dots+y$  を計算する関数を作りなさい。正しいことを検証しなさい。

## 引数の個数は重要

```

> myF <- function(x,y){
+   sum( x:y )
+ }
> myF(10)
以下にエラー x:y : 'y'が見つかりません
> myF(1,10)
[1] 55
> myF(10,1)
[1] 55
>

  
```

```

myF <- function(x,y){
  +   sum( x:y )
}
myF(10)
myF(20)
  
```

## 引数の個数に柔軟対処するには

- 仮引数に名前をつけることができ、デフォルト値が設定できる

```
> myF <- function(x=1,y=10){  
+   sum( x:y )  
+ }  
> myF(9)  
[1] 19  
> myF(,2)  
[1] 3  
> myF(x=9)  
[1] 19  
> myF(y=2)  
[1] 3  
> myF(y=2,x=9)  
[1] 44  
>
```

```
myF <- function(x=1,y=10){  
  sum( x:y )  
}  
myF(9)  
myF(,2)  
myF(x=9)  
myF(y=2)  
myF(y=2,x=9)
```

## 繰返しの例

```
> for(i in 5:8){  
+   print(i)  
+ }  
[1] 5  
[1] 6  
[1] 7  
[1] 8  
>  


```
for(i in 5:8){  
  print(i)  
}
```


```
myF <- function( x ) {  
  for(i in x){  
    print(i)  
  }  
}
```



```
myF( rnorm(4) )  
[1] 0.7225764  
[1] -1.779531  
[1] 1.212787  
[1] -0.6368527
```


```

勿論、関数の中でも、繰返しは使えます。

## 目次

- 前書
- 基本データ型 – ベクトル
  - 作る、代入、読み出し、加算、関数
- 識別子、ヘルプ
- プロット
- 関数の引数 – プロットを例にして
- 再びプロット
- 関数について – 少し丁寧に
- 繰り返しについて
- 補足: 結果の保存

## 繰返しの例(続)

- ベクトル(配列)に対する繰返しは、Rubyと同様に、要素に対する繰返しと、要素番号(index)に対する繰返しがある。

```
> b <- c(2,7,1,8)  
> for(e in b){  
+   print( e )  
+ }  
[1] 2  
[1] 7  
[1] 1  
[1] 8  
>
```

```
b <- c(2,7,1,8)  
for(e in b){  
  print( e )  
}
```

```
> b <- c(2,7,1,8)  
> for(i in 1:length(b)){  
+   print( b[i] )  
+ }  
[1] 2  
[1] 7  
[1] 1  
[1] 8  
>
```

```
b <- c(2,7,1,8)  
for(i in 1:length(b)){  
  print( b[i] )  
}
```

1からベクトルの長さまで。Rubyでは0から長さ-1である。

## 繰返し

- 同じ処理(平均値の計算)を複数個のデータ(1月の売上げ、2月の売上げ、、、、)に対し適用したいことはしばしばある。
- このような繰り返し処理を行わせる道具がloopであり、iteratorである。

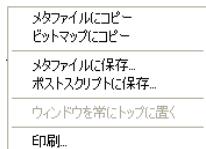
## 目次

- 前書
- 基本データ型 – ベクトル
  - 作る、代入、読み出し、加算、関数
- 識別子、ヘルプ
- プロット
- 関数の引数 – プロットを例にして
- 再びプロット
- 関数について – 少し丁寧に
- 繰り返しについて
- 補足: 結果の保存

8

## グラフのコピー

- グラフがあるウィンドウ上で、右クリックし、「メタファイルルファイルにコピー」「ビットマップにコピー」



## 付録

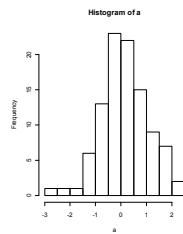
## R-console のコピー

- 編集 → 全て選択 → コピー  
予め、編集 → コンソール画面を消去 (Ctrl+I) しておき、必要なものを改めて実行してから、上記操作を行う。  
(勿論、コピーしてから編集するのもOK)

## 要約統計量(の例)

- hist(): ヒストグラム表示。
- mean(): 平均値
- median(): メディアン
- summary(): クォンタイル他

```
> summary(a)
   Min. 1st Qu. Median Mean 3rd Qu. Max.
-2.53100 -0.41550 0.07624 0.13130 0.71530 2.05200
```



## 最後に

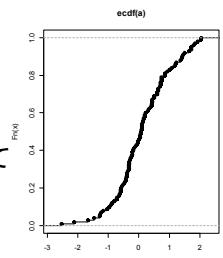
- R は、プログラミング言語である。
- 統計計算用に、いろいろな道具が用意されている
  - 具体的な内容は、順次
- 機械学習の道具もいろいろあり
  - これも、講義の進展に従い、順次
- Rを使って、機械学習のアルゴリズムを試してみよう！

## 経験度数分布

- 経験度数分布(経験相対頻度分布、累積度数分布、経験分布) ecdf (empirical cumulative distribution function) は、次のようにして表示することができる。

```
> ecdf(a)
Empirical CDF
Call: ecdf(a)
x[1:100] = -2.5311, -2.1125, -1.6676, ..., 2.0393, 2.0521
> plot(ecdf(a))

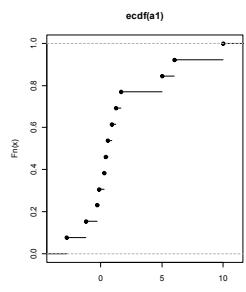
a<-rnorm(100)
b<-rnorm(100)^2
```



## 経験度数分布(続)

- データ数を減らして、見やすくしてみました。

```
> a1<-rnorm(10)
> a1 <- c(a1,5,6,10)
> plot(ecdf(a1))
>
```

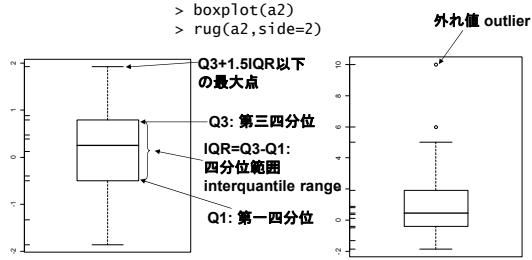


```
a<-rnorm(100)
b<-rnorm(100)+2
```

## 箱型図(箱ひげ図)

- boxplot です

```
> a1<-rnorm(10)
> a2 <- c(a1,5,6,10)
> boxplot(a1)
> rug(a1,side=2)
> boxplot(a2)
> rug(a2,side=2)
```



Q3+1.5IQR以下の最大点  
Q3: 第三四分位  
IQR=Q3-Q1:  
四分位範囲  
interquartile range  
Q1: 第一分位

外れ値 outlier