

## Node9

### Related software and documentation

R 은 Bell Laboratories 에 있는 Rick Becker, John Chambers 그리고 Allan Wilks 에 의해 S language 로 구현되었고, S-PLUS 의 기본시스템이기도 합니다.

S language 의 발전은 John Chambers 와 공동저자들에 의해 씌여진 네 권의 책에 잘 기술되어 있습니다. R 의 기본서는 Richard A. Becker, John M. Chambers 그리고 Allan R. 에 의해 씌여진 The New S Language : A Programming Environment for Data Analysis and Graphics 가 있습니다. John M. Chambers 와 Trevor J. Hastie 에 의해서 씌여져 1991년에 배포된 Statistical Models in S 에는 S 의 새로운 기능들이 잘 설명되어 있습니다. John M. Chambers 가 쓴 Programming with Data 을 토대로 formal methods 와 methods 패키지 클래스가 **작성되었으며** [appendix.references]Appendix D 에 언급된 참고서적들이 나열되어 있습니다.

현재 시중에는 R 을 이용한 데이터 분석과 통계학에 관련된 많은 책들이 **나와 있으며** S/S-PLUS 에 관련된 많은 문서들 역시 R 을 사용하는데 도움이 될 것입니다. **그러나, R 과 S는 다르다는 점에 주목할 필요가 있습니다.** 이것은 The R statistical system FAQ에 있는 "What documentation exists for R?" 에 설명되어 있습니다.

## Node10

### R and Statistics

R environment 에 **관해 제공되는** 이 안내서는 Statistics를 의미하지 않습니다. 하지만, 많은 사람들은 R 을 Statistics System으로 사용합니다. 우리는 R 이 기존부터 현대에 이르기까지 존재하는 많은 통계학적 기술들이 구현된 통합환경으로 인식되어지길 **바랍니다.** 이들 중 몇 가지는 base R 이라는 통합환경에 구현되어 있으며, 대부분 packages(패키지) 형태로 제공됩니다. "standard" 와 "recommended"로 구분된 대략 25개의 패키지가 R 과 함께 제공되어지며, 더 많은 패키지들을 인터넷 CRAN 저장소 ( <http://CRAN.R-project.org> )을 통하여 찾아볼 수 있습니다. 패키지에 관해 더 많은 정보를 얻고 싶으시다면, [chapter.packages]Chapter 13 Packages 을 **참고 하시기**를 바랍니다.

R 은 대부분의 기존의 통계학적 방법부터 최근의 풍부한 방법들까지 지원하고 있으나, 이를 위해서는 **사용자** 스스로의 노력을 요구합니다.

S(결과적으로는 R)는 다른 주요 통계학 관련 소프트웨어들과는 중요한 근본적 차이가 있습니다. S 에서 **기본적으로 통계적 분석은** 일련의 수행 및 연산과정을 통하여 얻은 결과물을 object 라는 중간매개체에 저장합니다. SAS 나 SPSS 가 regression 혹은 discriminant analysis 을 수행하여 방대한 양의 결과물을 제공하는 것과 달리, R 은 최소한의 결과물만을 제공하고, 더 자세한 결과에 대해서는 또 다른 R function 에 의하여 얻을 수 있도록 합니다.

## Node11

### R and the window system

R 을 사용하는 가장 쉬운 방법은 그래픽적인 환경을 제공하는 windowing system을 실행시키는 것입니다. 이 안내서는 **이런 기능을 지원하는 컴퓨터의 사용자를 위해** 작성되었습니다. R

environment 의 실행에는 많은 방법들이 있지만, 여기에서는 주로 X window system 을 지칭합니다.

대부분의 사용자들은 때때로 컴퓨터에서 운영체제와 직접적으로 의사소통을 할 필요성을 느낍니다. 이 안내서는 UNIX 라는 운영체제와 의사소통을 중점적으로 다루고 있으므로 Windows 혹은 Mac OS 사용자라면 약간의 수정이 필요합니다.

R 의 성능을 최적화 하는 방법은 굉장히 단순하다고 할 수 있으나, 다소 번거로울 수가 있습니다. 이러한 문제에 대해서는 이 안내서에서는 다루지 않지만, 만약 사용에 어려움을 느끼신다면 주변에 도움을 요청하십시오.

## Node12

### Using R interactively

R을 실행시키면, R은 입력명령을 기다리는 prompt를 사용자에게 보여줍니다. 기본적인 prompt 는 '>'이며, 이것은 UNIX 의 shell prompt 와 동일합니다. 그래서, 때로는 아무것도 실행이 되지 않은 것처럼 보일 때도 있습니다. 원한다면 다른 R prompt를 사용할 수도 있습니다. 우리는 UNIX shell prompt 는 '\$'라고 가정할 것 입니다.

UNIX에서 R 을 실행하는 과정은 다음과 같습니다.

1. 먼저 데이터를 저장하고, 작업을 위한 'work'라고 하는 sub-directory를 생성합니다. 이것이 당신이 앞으로 R 을 사용할 때마다의 working directory 가 될 것입니다.
2. \ \$ mkdir work
3. \ \$ cd work
4. R 을 실행시킵니다.
5. \ \$ R
6. 이제부터 R 명령어들을 입력하고 작업을 할 수 있습니다.
7. R 을 종료하기 위해서는 다음의 명령어를 입력합니다.
8. > q()

이때, R 은 당신에게 R session 에서 작업하던 데이터를 저장할 것인지 물어 볼 것입니다. 어떤 시스템은 대화창을 보여줄 수도 있으며, 또 다른 어떤 시스템은 데이터를 저장 후 종료, 저장 없이 종료, 혹은 취소하고 작업하던 R session으로 돌아가기 라는 텍스트 형태의 yes, no 또는 cancel 메뉴를 보여줄 것입니다. 이때 저장된 데이터는 다음 번 실행시의 R session 에서 다시 사용될 수 있습니다.

R session 을 다시 시작하고자 할 때는 간단합니다.

1. 'work' 라는 working directory를 생성하거나 이동한 뒤, 이전과 같은 방법을 반복합니다.
2. \$ cd work
3. \$ R
4. R 프로그램을 사용한 뒤 session 을 종료하기 위해서는 다시 한번 q())를 입력합니다.

Windows 에서 R 을 사용하기 위한 기본적인 절차는 위와 동일합니다. Working directory와 동일한 개념의 folder를 생성한 뒤, 'Start In' 메뉴에서 R 바로가기를 만듭니다. 이제 바탕화면의

바로가기 아이콘을 double clicking 하여 R 을 실행할 수 있습니다.

## Node14

### Getting help with functions and features

R 은 UNIX 에서의 man 과 같은 기능의 help (도움말) 기능이 존재합니다. 이것은 name (이름) 을 가지고 있는 function(함수)에 대해서 알고 싶을 때 사용됩니다. 예를 들어, solve 라는 함수에 대해서 알고 싶다면 다음과 같이 입력합니다.

```
> help(solve)
```

아래와 같은 방법으로도 동일한 결과를 확인해 볼 수 있습니다.

```
> ?solve
```

special character (특수문자) 에 대해서 알고 싶을 때는, 질의어는 반드시 큰따옴표 혹은 작은 따옴표를 함께 사용하여 "character string" 의 형태로 사용해야 합니다. 이러한 방식은 if, for 그리고 function 등 에서 여러 개의 단어를 표현하기 위한 문법적 요소입니다.

```
>help("[")
```

위와 같은 이러한 인용부호의 형태는 "It's important" 같은 문장을 표현하기 위해서도 사용되며, 앞으로의 편의를 고려하여 큰따옴표를 사용할 것을 권장합니다.

다음의 명령어를 이용하여 설치된 R help를 HTML 의 형식으로 볼 수 있습니다.

```
> help.start()
```

이 명령어는 hyperlink를 이용하여 help 페이지를 탐색할 수 있는 Web Browser를 실행시킵니다. UNIX 에서는 HTML 을 기반으로 하는 help system 에서 찾고자 하는 help에 대한 질의어를 전달합니다. help.start() 에 의해서 실행된 브라우저에 있는 'Search Engine and Keywords' 는 사용이 가능한 모든 함수를 검색해내는 high-level concept list를 포함하고 있기에 매우 유용합니다. 이것은 당신이 원하고자 하는 것을 빠르게 찾아내고, R 에서 지원되는 것이 무엇인가를 파악하는 가장 좋은 방법입니다.

help.search 또는 ?? 라는 명령어는 다양한 방법으로 help 를 찾아줍니다.

```
> ??solve
```

자세한 내용과 많은 예제를 찾기 위해서는 ?help.search 를 사용해보세요.

찾고자 하는 주제에 관련된 예제들은 다음의 명령어를 이용하여 실행합니다.

```
> example(topic)
```

Windows 버전의 R 은 또 다른 방법의 help system 을 가지고 있습니다.

```
> ?help
```

이에 대해서 더 알고 싶으시다면, 위의 명령어를 사용해보세요.

## Node 15

R 은 기술적인 측면에서 아주 간단한 문법을 가진 expression language 입니다. R은 UNIX 를 기반으로 하는 패키지들과 같이 대소문자를 구별(case sensitive) 합니다. 즉, 'A' 와 'a' 는 다른 것으로 인식되고 다른 변수들을 가리킬 수 있습니다. R name 에 사용되는 symbol(기호) 들은 운영체제 및 실행되는 국가 (기술적으로 locale 에 의해)에 따라 달라질 수 있으며, 일반적으로 모든 알파벳과 숫자를 조합한 형태는 사용할 수 있습니다. 게다가 '\$ . \$' 또는 알파벳으로 시작하

는 name (이름) 은 '\$ .' 과 '\_' 함께 사용되는 것이 허락됩니다. 만약, name 이 '\$ .'로 시작한  
다면 name 의 두번째 문자는 숫자여서는 안됩니다.

기본적인 명령어는 expression 과 assignment 의 형태로 구성됩니다. 만약 expression (표현  
식)이 명령어처럼 주어지면, 이것이 실행된 뒤 expression 은 보이지 않습니다. 또한,  
assignment (할당)은 expression 을 실행하고, 값을 변수에 저장된 후 그 결과는 자동적으로 보  
여지지 않습니다.

commands(명령어)들은 semi-colon ('\$ ;\$') 또는 newline 에 의해서 구분되어 집니다. 기본적  
인 종류의 commands 는 중괄호 ('\$ \{\$' 또는 '\$ \}\$') 에 의하여 하나의 그룹으로 묶을 수 있습  
니다. comments 는 프로그램상의 어느 곳에나 놓일 수 있으며, hashmark ('\$ \char93 \$') 로  
시작한 행의 전부를 주석처리 합니다.

만약 command 가 현재 행의 끝에서 끝나지 않고, 다음 행으로 넘어간다면 R 은 +  
(continuation) 모양의 prompt 를 보여줄 것입니다. 이는 command 가 끝날 때까지 계속해서  
나타나며, 이어지는 명령들을 계속해서 받아들일 것입니다. 우리는 이 문서에서 continuation  
prompt 를 생략하고, indenting 을 이용하여 이를 나타낼 것입니다.

command 행은 console 에서 character 형이 아닌 대략 4095 byte 까지 한 번에 입력 될 수 있  
습니다.

## Node 16

많은 종류의 UNIX 와 Windows 에서는 R 이 이전에 사용되었던 명령어를 다시 볼 수 있도록 하  
는 기능을 포함하고 있습니다. 키보드의 세로방향 화살표들을 command history 를 이전 명령어  
들을 앞으로 뒤로 살펴 보는데 사용 합니다. 이렇게 이전의 명령어를 불러오면, 가로방향 화살표  
를 이용하여 명령어의 character 를 DEL 키를 이용하여 지우거나 다른 키를 사용하여 추가 및 변  
경을 할 수도 있습니다. 이에 대해 더 자세한 사항에 대해서는 [appendix.editor]Appendix C  
를 참고 해 주세요. UNIX 상에서 recall 과 editing 하는 특성은 사용자의 개인 설정에 따라 다릅  
니다. 이것은 readline library 에 대하여 매뉴얼 목록 을 불러들임으로써 변경할 수 있습니다.

또한, Emacs text editor 는 R 과 함께 작업할 수 있는 풍부한 기능을 제공하고 있습니다. (via  
ESS, Emacs Speaks Statistics) 이에 대해서는 R statistical system FAQ 에 있는 "R and  
Emacs" 를 살펴보십시오.

## Node 18

R 이 생성하고 다루는 그 모든 것들은 objects 입니다. 이것들은 variables (변수), numerical  
array(숫자 배열), character strings(문자열), functions(함수), 그리고 이러한 것들로 이루어  
진 좀더 일반적인 형태의 것을 모두 포함합니다.

R session 내에서 objects 들은 name 에 의해서 생성되고, 저장되어 집니다.

```
> objects()
```

위와 같은 R 명령어를 사용하여 (다른 방법으로 `ls()`를 사용) 현재 R에서 사용하고 있는 모든 `objects` 들을 확인 할 수 있으며, 이러한 `objects` 들의 모임이 현재의 `workspace` 라는 곳에 저장 되는 것입니다.

이러한 `objects` 들은 함수 `rm` 를 이용하여 삭제할 수 있습니다.

```
> rm(x, y, z, ink, junk, temp, foo, bar)
```

R session 내에서 생성된 모든 `objects` 들은 **파일 안에** 영구적으로 저장되어 **다음 번** R session 에서 다시 사용될 수 있습니다. 모든 R session 의 마지막에서는 모든 가능한 `objects` 들을 저장 할 수 있는 기회가 주어집니다. 만약 이 작업을 수행하기를 원한다면, 모든 `objects` 들은 현재의 `directory` 안의 `'.RData'` 라는 파일에 저장되며, session 내에서 사용한 모든 `command` 들은 `'.Rhistory'` 라는 파일에 저장되게 됩니다.

같은 디렉토리 안에서 R 이 다시 실행된다면, 이 파일들로부터 `workspace` 를 관련된 **명령어 기록과** 함께 불러들입니다.

R 을 이용하여 통계분석을 할 때 여러 개의 다른 `working directory` 들을 사용할 것을 권장합니다. 보통 `objects`를 명명할 때 `x` 와 `y` 라는 이름을 사용하는 것은 매우 흔하고, 이러한 명칭은 단순한 분석 시에는 유용하지만, 같은 디렉토리 안에서 여러 개의 분석을 행할 시에는 이러한 변수 명명법은 어떤 것이 무엇을 의미하는지 혼란스럽게 만듭니다.

## Node 21

`Vector` 는 서로 상응하는 `element` 끼리 연산할수 있도록 하는 `arithmetic expression` (산술연산식) 안에 사용될 수 있습니다. 같은 연산식 안에 있는 `vector` 의 길이가 반드시 같아야 할 필요는 없지만, 만약 길이가 다르다면, 가장 긴 길이를 가지는 `vector`의 길이를 짧은 길이의 `vector`가 동일하게 맞추기 위해서 짧은 길이의 `vector` 가 `recycled`(반복) 되어지게 됩니다. (만약 필요하다면, `fractionally` 하게 반복될 수 있습니다.) 특히, `constant` (상수) 의 경우는 자동으로 긴 길이의 `vector` 만큼 `constant` 를 복사하여 길이를 맞추게 됩니다.

```
> v <- 2*x + y + 1
```

위의 연산결과는 `$ x$` 가 2.2 배로 확장되고, 1회 반복된 `$ y$` 된 후, 11번 반복으로 이루어진 `$ \mathbf{1}$` 을 형성하여 모두 같은 길이의 `vector` 가 된 후 각각 상응하는 `element` 끼리 연산 되어 11개의 숫자로 된 `vector v` 를 생성합니다.

일반적인 산술연산인 `$ +, -, *, / $` 그리고 `$ ^{\wedge}$` 는 이러한 `elementwise` 방식의 연

산을 수행합니다. 또한,  $\log\{\}$ ,  $\exp\{\}$ ,  $\sin\{\}$ ,  $\cos\{\}$ ,  $\tan\{\}$ ,  $\sqrt{\}$  와 같은 함수들 역시 **elementwise** 방식을 기초로 산술연산을 수행합니다. **vector** 내의 각각의 **element** 를 검색하여 글자 그대로 최대값을 구하기 위해서  $\max\{\}$ 를, 그리고 최소값을 찾기 위해  $\min\{\}$  함수가 사용됩니다. 이런 **elementwise** 개념을 사용하는 함수는 여러 가지가 있습니다. **range** 는  $c(\min\{x\}, \max\{x\})$  로 **vector** 의 범위를 구하는데 사용됩니다. **length(x)** 라는 함수는  $x$  내에 있는 **elements** 들의 개수를 알려주며, **sum(x)** 라는 함수는  $x$  안에 있는 모든 **elements** 들의 합을 계산해냅니다. 또한 **prod(x)** 함수는  $x$  안의 모든 **element** 들의 곱을 반환합니다. 을 구합니다.

**mean(x)** 와 **var(x)** 에 의해서 얻어지는 두 가지 통계함수는 각각  $\text{sum}(x)/\text{length}(x)$  그리고

$$\text{sum}((x - \text{mean}(x))^2) / (\text{length}(x) - 1)$$

에 의하여 구해지며, 이를 **sample mean** 그리고 **sample variance** 라고 합니다. 만약 **var()** 함수의 **argument** 가 **n-by-p matrix** 라면, **p-by-p sample covariance matrix** 를 계산해냅니다. 이때 **argument** 는 **n-by-p matrix** 를 **independent p-variate** 형태인 **n** 개의 **sample vector** 들로 인식합니다.

**sort(x)** 함수는  $x$  가 가지는 모든 **element** 들을 **ascending** 순서로 재정렬하여  $x$  와 동일한 길이의 **vector** 를 돌려줍니다. 이로 인해, R 은 정렬에 관한 더 많은 종류의 함수들을 제공하고 있습니다. (**permutation** 을 얻어내기 위해서 **order()** 또는 **sort.list()**가 사용됩니다.)

여러 개의 **vectors** 들이 주어질 때, **max** 와 **min** 는 이 **vectors** 들이 가진 전체의 **elements** 중에서의 최대값과 최소값을 찾아냅니다. **pmax** 와 **pmin** 은 **parallel maximum** 과 **parallel minimum** 으로 입력되어진 **arguments** 중 가장 긴 **argument** 의 길이와 동일한 **input vector** 를 생성합니다.

대부분의 경우 사용자들은 **vector** 안에 입력되는 수치형 데이터가 **integer**이나 **real** 또는 **complex** 인지 신경 쓰지 않습니다. 내부적으로 산술연산은 **double precision real** 을 기본으로 하여 수행되고, 만약 **complex** 가 입력된다면 **double precision complex** 이 기본으로 설정됩니다.

이때 **complex** 숫자를 사용하기 위해서는 사용자가 **complex part** 를 정확히 명시해야 합니다. 즉,

```
sqrt(-17)
```

위와 같은 입력은 **NaN** 또는 **warning** 을 보여줄 것입니다. 그러나,

```
sqrt(-17+0i)
```



위의 expression 은 complex numbers 로서 정확히 인식하고, 정상적인 연산을 수행할 것입니다.

## Node 22. Generating regular sequences

R 은 일반적인 sequence(수열)을 생성하는데 여러 가지 기능들을 가지고 있습니다. 예를 들면 `$ 1:30$` 은 `$ c(1,2,\ldots,29,30) $` 을 의미합니다. Colon operator 는 expression 안에서 연산시 가장 최우선순위를 가집니다. 그 예로, `$ 2*1:15$` 는 `$ c(2,4,6,\ldots, 28,30)$` 과 동일합니다. `$ n <- 10$`을 입력해보고 `$ 1:n-1$` 과 `$ 1:(n-1)$`을 서로 비교해보십시오.

거꾸로 배열 되는 수열<sup>1</sup>을 생성하기 위해서는 `$ 30:1$` 라고 하면 됩니다.

`seq()` 함수를 이용하면 좀 더 일반적인 형태의 수열을 생성할 수 있습니다. 만약 이 함수가 5개의 arguments를 가지고 있다면, 이것들 중 단 몇 개 만을 이용하여 한번에 원하는 형태의 sequence 를 얻을 수 있습니다. 처음 두 개의 arguments 는 sequence 의 처음과 끝을 지정하면, colon operator 를 이용하여 얻은 sequence 와 동일한 결과를 얻을 수 있습니다. 즉, `seq(2,10)` 과 `2:10` 은 같은 결과를 보여줍니다.

`seq()` 의 parameters 들은 다른 R 함수들처럼 named form 의 형태로 사용되어질 수 있습니다. 처음과 끝을 의미하는 처음 두 개의 parameters 들은 `from=value` 그리고 `to=value` 라는 named form 의 형태로 사용할 수 있습니다. 그러므로 `seq(1,30)`, `seq(from=1, to=30)`, `seq(to=30, from=1)` 은 모두 `1:30` 과 동일한 표현입니다. `seq()` 함수의 그 다음 argument 는 `by=value`로서 step size를 결정하고, 그 다음 argument 인 `length=value`는 수열의 길이를 결정하는데 사용됩니다. 만약, step size 에 대해서 아무것도 주어지지 않는다면 기본적으로 `by=1` 이 설정됩니다.

예를 들면

```
> seq(-5, 5, by=.2) -> s3
```

이것은 s3 라는 vector `$ c(-5.0, -4.8, -4.6, \ldots, 4.6, 4.8, 5.0)$` 을 생성합니다. 이와 비슷한 원리로,

```
> s4 <- seq(length=51, from=-5, by=.2)
```

역시 s3 와 동일한 sequence s4를 얻을 수 있습니다.

이 함수의 5번째 argument 는 `along=vector` 인데, 이것은 독립적으로 사용되어지며, 생성하는

---

<sup>1</sup> 등차수열? Or the opposite word

sequence 는 `$ 1, 2, \ldots, $ length(vector)` 와 같습니다. 이때 empty vector 가 입력되면, empty sequence 를 돌려줍니다.

이와 연관된 함수로 다양한 방법으로 object 를 반복하는데 이용되는 `rep()` 함수가 있습니다. 가장 간단한 형태는 아래와 같습니다.

```
> s5 <- rep(x, times=5)
```

이것은 `s5` 가 `$ x$` 를 5번 연이어 반복한 object 를 가리키는 것을 의미합니다. 이것은 아래와 같은 또 다른 유용한 형태를 제공합니다.

```
> s6 <- rep(x, each=5)
```

이것은 vector `$ x$` 의 element 를 각각 5회씩 반복했음을 의미합니다.

## Node 23. Logical vectors

R 은 numerical(수치형) vector 뿐만 아니라, logical(논리형) 연산도 가능합니다. logical vector 의 element 는 각각 TRUE, FALSE, 또는 NA (Not Available) 의 값을 가질 수 있습니다. T 와 F 는 TRUE 와 FALSE 의 약자이기는 하나, 이것이 reserved keywords는 아닙니다. 그렇기 때문에 사용자에 의해서 이 값들이 바뀌어 질 수도 있습니다. 따라서, 항상 TRUE 와 FALSE 를 사용할 것을 권장합니다.

논리형 vectors 의 값은 conditions 에 의해서 산출됩니다.

```
> temp <- x > 13
```

이것은 `x` 와 동일한 길이를 가진 vector `temp` 를 생성하며, `temp`의 각각의 element 는 주어진 조건에 일치한다면 TRUE, 그렇지 않으면 FALSE 로 채워집니다.

논리연산자는 `$ <, <=, >, >=$` 가 있으며, 같음을 비교하기 위해서는 `$ == $`을, 같지 않음을 나타내기 위해서는 `$ != $` 을 사용합니다. 또한, `c1` 과 `c2` 가 각각의 logical expression 이라면, `c1 & c2` 는 교집합("and"), `c1 | c2` 은 합집합("or"), 그리고 `!c1` 는 여집합("negation")을 의미합니다.

logical vector 는 일반적인 산술연산 역시 가능합니다. 이 때, FALSE 는 0, TRUE 는 1 로 coerced (강제변환) 되어 계산됩니다. 하지만, 종종 logical vector 와 coerced 된 값들이 일치하지 않는 경우가 있는데, 이에 대한 것은 다음 섹션에서 볼 수 있습니다.

## Node 24. Missing values



어떤 경우에는 **vector** 의 값을 알 수 없는 경우가 있습니다. 이런 경우는 통계학적 용어로 "Not available" 혹은 "missing value" 인 두 가지가 있으며, 이것은 **NA** 표현됩니다. **NA** 에 대한 어떠한 연산 역시 **NA** 를 산출합니다. 이러한 규칙은 단순히 연산에 대한 정보가 불충분하므로 연산이 불가능 하기 때문에 산출 됩니다.

**is.na(x)** 라는 함수는 **vector x** 와 동일한 길이를 가지며, **x**의 **element** 가 **NA** 인 경우에만 **TRUE** 를 생성하고, 그 나머지는 **FALSE** 인 **logical vector** 를 생성합니다.

```
> z <- c(1:3,NA); ind <- is.na(z)
```

**x == NA** 라는 **logical expression** 은 **is.na(x)** 라는 함수와 다르다는 것을 주의해야 합니다. **NA** 는 어떤 특정한 값이 아니라 단지 값을 표현할 수 없다는 **marker** 이기 때문입니다. 그러므로 이것에 대한 연산을 수행할 수 없기 때문에 **x == NA** 는 **x** 와 동일한 길이를 가지되, 모든 **element** 가 **NA** 를 가지는 **vector** 를 생성하게 됩니다.

산술연산을 할 때에는 **NaN (Not a Number)** 으로 표현되는 또 다른 형태의 **missing value** 가 존재합니다.

```
> 0/0  
> Inf - Inf
```

위의 두 표현식은 결과값을 정의할 수 없기 때문에 **NaN** 을 산출합니다.

**is.na(xx)** 함수에 대해 정리하면 **NA** 와 **NaN** 의 값인 경우 **TRUE** 를 반환합니다. 이것들의 다른 점은 **is.nan(xx)** 함수가 단지 **NaN** 의 값에 대해서만 **TRUE** 를 반환하는데 있습니다.

**Missing values** 들은 때때로 **character vector** 에서 인용부호 없이 **\$ <\$NA\$ >\$** 를 보여주기도 합니다.

## Node 25. Character vectors

**Character quantities** 와 **character vector** 는 R 에서 **plot label** 을 사용시에 자주 사용됩니다. 이것들은 "x-values", "New iteration results" 와 같이 인용부호를 사용하여 구분합니다.

**character string** 은 큰(") 또는 작은(') 따옴표를 사용하여 입력하지만, 결과물을 출력시에는 항상 큰 따옴표가 사용됩니다. 출력에 관련된 문자를 표시할 때는 \와 같은 **escape character** 를 사용하여 C-style 의 **escape sequence** 를 사용하게 됩니다. 즉, \\이 입력하면 \\가 보여지게 되며, 따옴표(") 를 표시하기 위해서는 \" 를 입력합니다. 그 외의 유용한 **escape sequence** 는 \n (newline), \t (tab), \b (backspace) 와 같은 것이 있습니다. - ?Quotes 명령어를 이용하면 이

것에 대한 모든 리스트를 확인 할 수 있습니다.

**character vectors** 는 **c()** 함수에 의해서 한 개의 문자열로 묶일 수가 있습니다. 이것에 대한 예제는 앞으로 보게 될 것입니다.

**paste()** 함수는 임의의 **arguments**를 한데 묶어 이들을 한 개의 **character string**(문자열) 로 묶어줍니다. **argument** 안에 포함되어 있는 숫자들 역시 출력시 보여지는 것과 같이 **character string** 으로 **coerce** 되어집니다. **arguments** 들은 결과물에서 **single blank character** 라는 것으로 각각의 **argument** 들을 구분합니다. 그러나 이것은 **string** 으로 바꾸어주는 **sep=string** 라는 **named parameter** 에 의해서 바뀔수 있습니다.

```
> labs <- paste(c("X","Y"), 1:10, sep="")
```

이것은 다음과 같은 **character vector** **labs** 를 생성합니다.

```
c("X1", "Y2", "X3", "Y4", "X5", "Y6", "X7", "Y8", "X9", "Y10")
```

여기에서 **c("X", "Y")** 는 **sequence 1:10** 의 길이를 맞추기 위하여 5번 연이어 반복된다는 것에 주목할 필요가 있습니다.

## Node 26. Index vectors; selecting and modifying subsets of a data set

**vector** 의 이름옆에 **square brackets** 을 붙이고 이 안에 **index vector** 를 이용하여 얻고자 하는 **vector** 의 **subset** 을 바로 얻어낼수 있습니다. **index vectors** 들은 크게 4가지 유형이 있습니다.

1. A logical vector 이러한 경우 **index vector** 는 반드시 **element** 가 선택되어지는 **vector** 와 동일한 길이를 가져야 합니다. **index vector** 의 **element** 중에서 **TRUE** 에 해당하는 것만을 남겨두고, **FALSE** 에 해당되는 **element** 는 삭제될 것입니다.

```
> y <- x[!is.na(x)]
```

이것은 **NA** 가 아닌 값만을 뽑아내어 본래의 순서대로 **y** 를 생성할 것입니다. 실제로 **x** 가 **missing value** 를 가지고 있다면 **y** 의 길이는 **x** 보다 짧을 것입니다.

```
> (x+1)[(!is.na(x)) & x>0] -> z
```

이것은 **x** 가 양수이고, **NA** 가 아닌 값들만을 골라 이에 **1** 을 더한 값을 얻어내고 **z** 라고 할 것입니다.

2. A vector of positive integral quantities 이 경우 **index vector** 내에 있는 값들은 반드시

$\{1, 2, \dots, \text{length}(x)\}$  내에 있어야 합니다. index vector 에 상응하는 elements 를 vector 를 골라내고, 이를 순서대로 접합할 수 있습니다. index vector 는 임의의 길이를 가질수 있으며, 그것을 사용하여 얻어낸 vector 역시 index vector 의 길이와 동일합니다. 예를 들면,  $x[6]$  는  $x$  의 6번째 element 를 의미합니다.

```
> x[1:10]
```

이것은  $x$  의 1 번째부터 10번째 elements 만을 선택합니다. (만약  $\text{length}(x)$  가 10 보다 작지 않다고 가정한다면)

```
> c("x","y")[rep(c(1,2,2,1), times=4)]
```

이는 "x", "y", "y", "x" 로 이루어진 vector 가 4번 반복하여 얻어진 16개의 element 로 구성된 character vector 를 생성합니다.

3. A vector of negative integral quantities 이 경우의 index vector 는 포함하는 것이 아닌 index vector 에 해당되는 부분을 제외하는 것입니다.

```
> y <- x[-(1:5)]
```

이것은  $x$  안에 있는 처음 다섯개의 elements 를 제외한 나머지를  $y$  에 저장합니다.

#### 4. A vector of character strings

이 경우는 object 가 각각의 구성요소에 names 이라는 속성을 부여하여 그것들을 서로 구분할 때에만 적용됩니다. 이 경우 names vector 의 sub-vector 는 두번째 케이스에서 설명했던 것과 같은 방법으로 활용됩니다.

```
> fruit <- c(5, 10, 1, 20)
> names(fruit) <- c("orange", "banana", "apple", "peach")
> lunch <- fruit[c("apple", "orange")]
```

alphanumeric names 은 종종 numeric indice 보다 편리합니다. 이 방법은 특히 다음 장에서 보게 될 data frames 과 연계될 때 매우 유용합니다.

indexed expression 은 vector 의 indexed 된 element에 직접적으로 assignment를 이용하여 값을 전달하기 위해서 쓰여집니다. vector의 name 을 정확히 알 수 없거나 index 가 불가능 한 경우 임의의 expression 이 표현될 때  $\text{vector}[\text{index\_vector}]$  의 형태를 지니게 됩니다.

주어진 vector 는 index vector 의 길이에 자동으로 맞춰지며, logical 의 경우에는 그 길이가 동

일해야 합니다.

```
> x[is.na(x)] <- 0
```

이는 missing value 가 있는 x의 element 에 0 값을 할당하는 것입니다.

```
> y[y < 0] <- -y[y < 0]
```

이것은 아래의 expression 과 동일한 효과를 가져옵니다.

```
> y <- abs(y)
```

## Node 27. Other types of objects

Vectors 는 R 에서 가장 중요한 object 의 형태입니다. 그러나 이것 외에도 우리는 나중에 좀 더 일반적인 형태의 여러 가지 종류의 object 들을 보게 될 것입니다.

1. matrices 또는 좀더 일반적인 형태는 arrays 는 multi-dimension 형태의 vector 입니다. 사실은 이것들은 vectors 이며, 두 개 또는 그 이상의 indice 를 가지고 특수한 방법으로 표현되는 것입니다. 이에 대해서는 [chapter.array.matrix]Chapter 5 [Arrays and matrices] 를 참고하세요.

2. factors 는 categorical data 를 다루기 위한 가장 간편한 방법을 제공합니다. 이를 위해서는 [chapter.factors]Chapter 4 [Factors] 를 참고하세요.

3. lists 는 vector 내의 모든 elements 가 같은 종류 데이터형을 필요로 하지 않는 일반적인 형태의 vector 입니다. 이것은 자기 자신의 vectors 또는 lists 가 될 수도 있습니다. lists 는 statistical computation 에 있어서 매우 유용한 방법입니다. 이는 [section.lists]Section 6.1 [Lists] 를 참고하세요.

4. data frames 는 matrix 와 같은 구조를 지니지만, 각각의 열들은 다른 데이터형을 가질수 있습니다. data frame 은 한개의 행이 numerical 과 categorical variable 을 동시에 가진 한개의 observation 이라고 생각하면 편리합니다. 많은 실험들이 data frame 으로서 표현됩니다. treatments은 categorical 이나, responses 는 numeric 입니다. 이에 대해서는 [section.data.frames]Section 6.3 [Data frames] 를 살펴보세요.

5. functions 역시 그 자신이 project의 workspace 에 저장되는 일종의 object 입니다. 이것은 R 을 자유롭게 활용 할 수 있는 간단하고도 편리한 방법입니다. 이에 대해서는 [chapter.writing.function]Chapter 10 [Writing your own functions]를 참고하세요.

## Node 29. Intrinsic attributes: mode and length

R 은 기술적으로는 **objects** 라는 알려진 정보의 단위를 기반으로 작동합니다. **numeric (real)** 또는 **complex** 값을 가지는 **vector**, 논리값을 가지는 **vector**, 그리고 **character string** 으로 이루어진 **vector** 가 그 대표적인 예라고 할 수 있습니다. 이들 **vectors** 모두가 **numeric**, **complex**, **logical**, **character**, 그리고 **raw** 라는 동일한 **mode**를 가지고 있기 때문에 **atomic** 구조라고 합니다.

**Vector** 의 **elements** 는 반드시 **all of same modes** 이어야 합니다. 따라서 어떤 종류의 **vector** 이든지 간에 한 **vector** 를 구성하는 **elements** 는 모두 **logical**, **numeric**, **complex**, **character** 또는 **raw** 중 어느 한가지로서 모두 통일되어야 합니다. (이 규칙에 대해서 **Quantities Not available** 를 의미하는 **NA** 는 예외이며, **NA**의 종류에는 여러 가지가 있습니다.) **vector** 또한 **empty** 일 수 있으며, **이도 mode**를 가진다는 사실에 대해서 주의해야 합니다. **character string** 타입의 **vector** 가 **empty** 라면 **character(0)** 이라고 표시되며, **numeric** 타입의 **vector** 는 **numeric(0)** 이라고 나타냅니다.

R 이 가지고 있는 또 다른 **objects** 는 **lists** 이며, 이것의 **mode** 역시 **list** 입니다. 이것은 각각의 다른 종류의 **mode** 를 가질 수 있는 **objects** 들의 **sequence** 입니다. **lists** 는 그 자신이 자신의 **component** 를 구성할 수도 있기 때문에 **atomic** 타입이 아닌 **"recursive"** 타입이라고 알려져 있습니다.

또 다른 **recursive** 타입의 **objects** 역시 존재하는데 이는 **function** 과 **expression** 입니다. **functions** 은 우리가 나중에 다룰 사용자 정의 함수와 함께 쓰여지는 **R system** 을 구성하는 **object** 입니다. 이것은 **modeling** 에서 사용되는 **formulae** (공식) 에 대하여 다룰 때 간접적으로 언급하는 경우를 제외하고는, **objects**로서의 **expression** 은 R 에서도 어려운 개념에 해당되므로 이 안내서에서는 다루지 않습니다.

**mode** 라는 것은 **objects**를 구성하는 기본 데이터형을 의미합니다. 그런데, 이것은 **object**가 가지는 **"property"** 들 중 한 가지 일 뿐입니다. 모든 **object** 가 가지는 또 다른 **property** 는 **length** 라는 것 입니다. 임의로 정의된 **structure** 의 **mode** 와 **length** 를 알고 싶다면 **mode(object)** 또는 **length(object)** 라는 함수를 사용하면 됩니다.

**object** 가 가지고 있는 더 많은 종류의 **property** 에 대해서는 **attributes(object)** 라는 함수를 이용하여 알 수 있습니다. 이것에 대해서는 [section.attributes]Section 3.3 [Getting and setting attributes]를 참고하세요. **mode** 와 **length** 는 **object** 의 **"intrinsic attributes"** (기본적 혹은 본질적 속성)라고 합니다.

예를 들면, 만약 **z** 가 100 이라는 **length**를 가진 **"complex vector"** 이라고 가정하면 **mode(z)** 함수는 **"complex"** 라는 결과를 보여주고, **length(z)** 는 100 이라는 값을 보여줍니다.

대부분의 경우 R 은 프로그램내의 **필요하다면 어느 곳에서나** mode를 변경할 수 있습니다.

```
> z <- 0:9
```

위와 같이 z를 정의한 뒤,

```
> digits <- as.character(z)
```

위와 같이 `as.character(z)`라는 함수를 이용하여 `c("0", "1", \ldots, "9")` 와 같이 character vector 인 digits를 생성할 수 있습니다. 더 나아가 아래의 명령어를 통하여 이것을 다시 numerical vector 의 형태로 coercion (강제변환) 할 수도 있습니다.

```
> d <- as.integer(digits)
```

이제 d 와 z 는 동일합니다. 이러한 상호변경이 가능하도록 하거나, objects 가 이미 가지고 있는 또 다른 attribute (속성)을 알아보기 위하여 `as.something()` 형태의 많은 함수가 제공됩니다. 사용자는 이러한 **함수들과** 친숙해지기 위해서는 각각의 help 파일을 참조해야 합니다.

#### Node 30. Changing the length of an object

object 가 empty 일지라도 역시 mode 를 가지고 있습니다.

```
> e <- numeric()
```

이것은 mode 가 numeric 인 e 라는 vector를 생성한 것입니다. 이와 유사하게 `character()` 라는 함수는 character 형의 empty vector를 생성합니다. 만약 임의의 크기를 가지는 object 가 생성 되었을 때, 새로운 components를 단순히 범위 외의 index를 지정 함으로서 추가할 수 있습니다.

```
> e[3] <- 17
```

이것은 길이가 3 인 e 입니다. (처음 두 elements 는 둘 다 NA 입니다.) 새로이 추가되는 component 에 **의해** 생성되는 object 의 mode는 추가 되어지는 component 의 mode 에 의해서 결정되며, 이것은 모든 structure 에 적용됩니다.

종종 object 의 길이가 자동으로 조정 될 때가 **있는데**, 입력기능에 관계된 `scan()` 함수가 그 대표적인 예입니다. 이것에 대해 더 자세한 내용은 [section.scan]Section 7.2 [The scan() function] 를 참고하세요

이와 반대로, object 의 size 를 줄이는 방법은 오직 assignment 를 **이용하는** 방법 밖에 없습니



다. 만약, `alpha` 가 길이 10 의 `object` 이라고 가정한다면,

```
> alpha <- alpha[2 * 1:5]
```

이것은 짝수인 `index` 에 해당하는 `alpha` 의 값을 골라 길이가 5 인 `object` 를 생성할 것입니다. (물론 기존의 `index` 는 무시되고, 새로이 주어집니다.) 이렇게 얻어진 `alpha` 의 처음 세 개의 값을 얻기 위해서는 아래의 명령어를 사용합니다.

```
> length(alpha) <- 3
```

`vectors`를 확장하기 위해서도 동일한 방법이 사용될 수 있습니다.

### Node 31. Getting and setting attributes

`attributes(object)` 함수는 `object` 에 기본속성 외에도 현재 정의되어 있는 모든 속성들의 `list` 를 반환합니다. `attr(object, name)` 이라는 함수는 `object` 의 특정속성만을 알고 싶을 때 사용됩니다. 그러나 이러한 함수는 몇몇의 특수한 경우를 제외하고는 자주 사용되지는 않습니다. 이것은 어떤 특수한 목적에 의해서 제작일을 추가하거나, `R object` 에 특수한 연산자를 연결하고자 하여 새로운 속성이 추가될 때입니다. 그러나 이 개념은 매우 중요합니다.

새로운 `attribute`를 추가하거나 삭제할 때 약간의 주의가 요구 되어집니다. 그 이유는 이것이 `R` 에서 사용되는 `object system` 에 있어서 없어서는 안 될 부분이기 때문입니다.

만약 이러한 함수들이 `assignment` 의 왼쪽에 놓인다면, 새로운 `attribute`를 `object` 에 연결하거나, 혹은 기존의 값을 변경하는 것입니다.

```
> attr(z, "dim") <- c(10, 10)
```

이것은 `R` 이 `z` 가 가진 `dim` 이라는 속성을 변경하여 10-by-10 matrix 로 변경합니다.

### Node 34. A specific example

예를 들면, `Australia` 의 모든 `states`와 `territories`로부터 30명의 회계사로 이루어진 한 개의 샘플을 가지고 있다고 가정한다면, 그들의 출신지에 대해서 다음과 같이 `state` 라는 `character vector`를 생성합니다.

```
> state <- c("tas", "sa", "qld", "nsw", "nsw", "nt", "wa", "wa",  
            "qld", "vic", "nsw", "vic", "qld", "qld", "sa", "tas",  
            "sa", "nt", "wa", "vic", "qld", "nsw", "nsw", "wa",  
            "sa", "act", "nsw", "vic", "vic", "act")
```

character vector 의 경우, ``sorted"(정렬)란 말은 alphabetical order(알파벳 순서)를 의미합니다. factor() 함수를 이용하여 factor를 생성해냅니다.

```
> statef <- factor(state)
```

print() 함수는 factors 의 경우에 다른 objects 들과 약간 다른 방식으로 다룹니다.

```
> statef
[1] tas sa qld nsw nsw nt wa wa qld vic nsw vic qld qld sa
[16] tas sa nt wa vic qld nsw nsw wa sa act nsw vic vic act
Levels: act nsw nt qld sa tas vic wa
```

levels() 이라는 함수를 이용하여 factor 의 level 만을 알아낼 수 있습니다.

```
> levels(statef)
[1] "act" "nsw" "nt" "qld" "sa" "tas" "vic" "wa"
```

#### Node 35. The function tapply() and ragged arrays

위의 예제를 계속 사용하여 동일한 회계사들의 수입을 다른 vector 안에 저장하였습니다. (대략적으로 적당히 높은 화폐의 단위를 사용)

```
> incomes <- c(60, 49, 40, 61, 64, 60, 59, 54, 62, 69, 70, 42, 56,
               61, 61, 61, 58, 51, 48, 65, 49, 49, 41, 48, 52, 46,
               59, 46, 58, 43)
```

각 주 별로 소득에 대한 sample mean 을 얻고자 할 때 tapply() 라는 함수가 사용됩니다.

```
> incmeans <- tapply(incomes, statef, mean)
```

이것은 levels 에 의하여 구분 되어진 데이터들을 골라내어 각각의 level 에 대한 mean vector 를 반환합니다.

```
act nsw nt qld sa tas vic wa
44.500 57.333 55.500 53.600 55.000 60.500 56.000 52.250
```

tapply() 라는 함수는 여기에서 첫 번째 argument 로 incomes을, 두 번째 argument 로 statef 로서 각 그룹화 한 뒤, 그룹화된 각 데이터를 세 번째 argument 인 mean() 이라는 함수를 통하여 각 그룹에 해당되는 통계 값을 얻어내기 위해 사용됩니다. 이 함수를 사용하여 얻어낸 결과의 길이는 factor 의 levels 를 값을 저장하고 있는 vector 의 길이와 동일합니다. 이것에 대한 더 자세한 사항은 여러분이 직접 help 문서를 살펴보아야 합니다.

만약 우리가 `state income means` 에 대한 `standard errors`를 찾아야 한다면, 우리는 이것을 계산하는 R function 을 작성해야 합니다. `var()` 이라는 내장함수가 `sample variance`를 계산해주기 때문에, 이러한 R function 은 한 줄로 표현되어 `assignment` 가 되어집니다.

```
> stderr <- function(x) sqrt(var(x)/length(x))
```

(함수를 작성하는 것은 나중에 [chapter.writing.function]Chapter 10[Writing your own function] 에서 다룰 것이며, 이러한 경우는 `sd()` 와 같은 내장함수가 존재하므로 이것에 대한 함수작성은 불필요합니다.) `standard errors` 는 위와 같은 함수의 정의 이후 아래와 같이 사용됩니다.

```
> incster <- tapply(incomes, statef, stderr)
```

이것의 결과는

```
> incster
act  nsw nt  qld  sa tas  vic  wa
1.5 4.3102 4.5 4.1061 2.7386 0.5 5.244 2.6575
```

당신은 `state mean income` 에 대하여 95% 의 `confidence interval` 을 찾아야 할 수도 있습니다. 이것을 하기 위해서는 `tapply()` 는 `sample size` 를 찾기 위한 `length()` 라는 함수와 적절한 `t distribution` 으로부터 `percentage` 를 구하기 위한 `qt()` 함수와 함께 한 번 더 사용되어야 합니다.

`tapply()` 함수는 `multiple categories` 에 의한 복잡한 `vector indexing` 에도 사용됩니다. 예를 들면, 회계사들을 `state` 와 `sex` 라는 두가지 `factors` 에 의해서 동시에 분류해 내고 싶습니다. 그러나 이 단순한 예제(한개의 `factor` 만을 고려하는 경우)에서는 우리가 다음과 같은 과정이 일어날 것이라고 생각합니다. `vector` 안의 데이터들을 각각 `factor` 의 `level` 별로 분류하고 난 뒤, 분리된 각각의 `level` 에 다시 한번 동일한 함수가 적용됩니다. 이 값들은 `factor` 의 `attribute` 인 `levels` 에 의해서 명명된 함수의 결과값에 이 저장된 `vector` 입니다.

`vector` 와 `labelling factor` 의 조합은 가끔 `a ragged array` 라고 불리는 예제입니다. 그 이유는 `subclass` 의 크기가 일정하지 않기 때문입니다. `subclass` 의 크기가 모두 동일하다면, `indexing` 은 다음 섹션에서 보는 것과 같이 효율적으로 이루어질 것입니다.

## Node 36. Ordered factors

`Factor` 의 `level` 들이 자동으로 알파벳 순서로 저장되거나, 혹은 `factor` 에 저장한 순서대로 저장할 수도 있습니다.

때때로 `level`들은 자연스럽게 우리가 저장하고자 하거나, 통계적 분석에 적합하도록 정렬되기도

합니다. `ordered()` 함수는 그런 `ordered factor` 를 생성하거나, 그렇지 않은 경우에는 `factor`와 동일한 기능을 수행합니다. 대부분의 경우 `ordered` 와 `unordered factors` 사이에 다른 점은 단지 `ordered factor` 가 정렬된 `level` 을 보여주는 것 말고는 없습니다. 그러나 `fitting linear model` 에서 `contrasts` 는 각기 다른 결과를 줍니다.

## Node 38. Arrays

`Array` 는 `data entry` 를 표현하기 위해서 여러 개의 `subscript`를 가지는 형태를 의미합니다. R 은 특히 `matrices` 같은 `arrays` 를 생성하고 다루기 위한 간편한 기능들을 제공합니다.

`Dimension vector` 는 음의 정수가 아닌 값(즉, `0,1,2,3,...`)들로 이루어진 `vector` 이며, 이것의 `length` 가 `k` 이면, 이를 `k-dimensional` 이라고 합니다. 즉, `matrix` 는 `2 dimensional array` 입니다. `Dimensions` 은 1부터 `dimension vector` 내에서 주어진 값까지 표시 할 수 있습니다.

R 에서 사용되는 `vector`가 `dim` 과 같은 `attribution` 을 가지고 있는 `dimension vector` 를 가지고 있다면 `array` 로 인식됩니다. 예를 들어 `z` 가 1500 개의 `elements`를 가진 `vector` 라고 가정할 때,

```
> dim(z) <- c(3,5,100)
```

이것은 3-by-5-by-100 array 로 인식되도록 하는 `dim`의 `attribution`을 정의합니다.

좀 더 간편하고 사용하기 쉬운 `matrix()`와 `array()` 같은 함수가 제공되며, [section.array]Section 5.4[The `array()` function]을 참조하세요.

`FORTRAN` 시스템과 동일한 방식으로 데이터의 값들은 `array` 안으로 똑같이 할당됩니다. 즉, (`column major order`)라는 방식으로 첫 번째 서브스크립트가 빠르게 갱신되며, 마지막 서브스크립트는 가장 천천히 변경됩니다.

예를 들어, `a`라고 불리는 임의의 `array`가 있고, 이것의 `dimension vector` 가 `c(3,4,2)`이면, 이것은  $3 \times 4 \times 2 = 24$  개의 `entries` 들을 입력하길 요구합니다. 그리고 이 `data vector` 는 `$ a[1,1,1], a[2,1,1], \dots, a[2,4,2], a[3,4,2]$` 와 같은 순서로 데이터 값을 보관합니다.

`Array` 는 또한 `one-dimensional` 이 될 수 있습니다. 그러나 이런 `array` 는 (주로 `printing` 할 때) `vector`와 동일하게 취급되지만, 이런 경우는 혼란을 야기합니다.

## Node 39. Array indexing. Subsections of an array

`array`가 가진 데이터들은 `array name` 다음에 오는 `square bracket` 속에 있는 `컴마(,)`로 구분되어진 `subscript`에 의해서 나타내어 집니다.

일반적으로 array 의 subsection 은 subscript 대신에 index vectors로 표현하여 얻을 수 있습니다. 그런데, 임의의 index position이 empty index vector로 주어진다면, 해당 subscript의 모든 index를 의미합니다.

이전의 예제를 계속 사용하여, `a[2, , ]` 는 dimension vector가 `c(4,2)` 인 `4 × 2` array 를 의미하며, data vector는 아래의 값들을 모두 가지게 됩니다.

```
c(a[2,1,1], a[2,2,1], a[2,3,1], a[2,4,1],  
  a[2,1,2], a[2,2,2], a[2,3,2], a[2,4,2])
```

`a[ , , ]` 은 전체 array를 의미하며 이것은 모든 subscript를 생략하고 `a`를 단독적으로 사용 한 것과 같습니다.

`Z` 라고 불리는 임의의 array가 있다면, dimension vector 는 `dim(Z)`를 사용하여 얻어낼 수 있습니다.

또한 array name 이 one subscript 또는 index vector와 함께 사용되었다면, 이에 상응하는 data vector의 값들만이 사용될 것입니다. 이 경우에는 dimension vector는 무시 될 것입니다. 그런데 만약 single index가 vector가 아닌 자기자신이라면, 이 경우에 포함되지 않습니다. 이것은 조금 뒤에 다루어질 것입니다.

#### Node 40. Index matrices

임의의 subscript position에 index vector 뿐만 아니라, array 내의 임의의 위치에 데이터의 값을 할당하거나, 임의의 위치로부터 데이터를 뽑아내기 위해서 matrix는 index matrix와 함께 사용될 수 있습니다.

다음의 예제는 이것을 명쾌하게 설명합니다. doubly indexed array가 존재한다고 가정하면, index matrix 는 두 개의 열과 원하는 만큼의 양을 가진 행으로 구성됩니다. 이것들이 doubly indexed array를 구성하는 행과 열의 indice 들로 된 index matrix 입니다. 예를 들어, 우리가 4-by-5 array인 `X`를 가지고 있고, 다음을 수행하고 싶다면,

1. `X[1,3]`, `X[2,2]` 그리고 `X[3,1]`에 해당하는 데이터를 뽑아내서, vector 형태로 저장한 뒤,
2. 이 데이터 값들을 0 으로 바꾸고 싶습니다.

이런 경우, 우리는 3-by-2 사이즈의 subscript를 가진 array가 필요합니다.

```
> x <- array(1:20, dim=c(4,5))    # Generate a 4 by 5 array.  
> x  
[,1] [,2] [,3] [,4] [,5]  
[1,] 1 5 9 13 17
```

```

[2,] 2 6 10 14 18
[3,] 3 7 11 15 19
[4,] 4 8 12 16 20
> i <- array(c(1:3,3:1), dim=c(3,2))
> i                                     # i is a 3 by 2 index array.
[,1] [,2]
[1,] 1 3
[2,] 2 2
[3,] 3 1
> x[i]                                # Extract those elements
[1] 9 6 3
> x[i] <- 0                            # Replace those elements by zeros.
> x
[,1] [,2] [,3] [,4] [,5]
[1,] 1 5 0 13 17
[2,] 2 0 10 14 18
[3,] 0 7 11 15 19
[4,] 4 8 12 16 20
>

```

음수를 가지는 indice는 index matrix에서 사용할 수 없습니다. NA 와 0 값들은 허용합니다. index matrix의 행이 0을 포함한다면 이는 무시될 것이며, NA를 포함하는 행은 NA로 표시될 것입니다.

다소 쉬운 예제는 아니지만, b level를 가진 blocks 란 factor 와 v level 을 가진 varieties 라는 factor 를 가지고 (unreduced) design matrix를 생성하려고 가정해봅시다. 더 나아가 이 실험에서 n plot이 존재한다면, 다음과 같이 행할 수 있습니다.

```

> Xb <- matrix(0, n, b)
> Xv <- matrix(0, n, v)
> ib <- cbind(1:n, blocks)
> iv <- cbind(1:n, varieties)
> Xb[ib] <- 1
> Xv[iv] <- 1
> X <- cbind(Xb, Xv)

```

N 이라고 불리는 incidence matrix를 생성하기 위해서는 다음과 같이 하면 됩니다.

```

> N <- crossprod(Xb, Xv)

```



그러나, 이 matrix를 생성하기 위한 더 간편한 방법은 table() 함수를 이용하는 것입니다.

```
> N <- table(blocks, varieties)
```

Index matrix 는 반드시 numerical이어야 합니다. matrix와 같이 사용된 logical 또는 character 같은 matrix들은 indexing vector와 같이 간주됩니다.

#### Node 41. The array() function

vector structure에 dim이라는 attribution 을 부여하는 것 외에도, arrays 는 array 함수에 의해 생성되어 질 수 있습니다.

```
> Z <- array(data_vector, dim_vector)
```

예를 들면, h라는 vector는 24개의 numerical 값을 가집니다.

```
> Z <- array(h, dim=c(3,4,2))
```

이것은 Z라는 array 가 h라는 vector를 이용하여 3-by-4-by-2 형태의 array를 가지는 것을 의미하며, 이것은 다음의 명령어와 동일합니다.

```
> Z <- h ; dim(Z) <- c(3,4,2)
```

만약 h가 24개보다 적은 양을 가지고 있다면, 24개의 elements 를 맞추기 위해서 h의 값들이 재 사용됩니다. 이것에 대해서는 [subsection.recycling.rule]Section 5.4.1[The recycling rule] 를 참조하세요 . 그러나, dim(h) <- c(3,4,2)와 같은 사용은 동일한 길이를 갖지 않는다는 error message 를 보여줄 것입니다. 이것은 특별한 경우의 예입니다.

```
> Z <- array(0, c(3,4,2))
```

이 명령어는 Z라는 array를 모두 0 으로 채웁니다. 여기에서 dim(Z) 는 dimension vector c(3,4,2)를 의미하며, Z[1:24]은 h안에 있는 data vector 를 나타냅니다. 또한, empty subscript 와 함께 사용된 Z[] 또는 아무런 subscript 가 없는 Z는 전체 array를 의미합니다.

Arrays은 산술연산에도 사용되며, 기본적으로 data vector에 대한 element-wise 방식의 연산을 수행합니다. 이때 연산자의 dim attribution 은 일반적으로 동일해야 하며, 이것이 결과치의 dimension vector 가 됩니다. 그러므로 만약 A, B 그리고 C 가 유사한 array 들이라면,

```
> D <- 2*A*B + C + 1
```

이것은 element-wise 방식에 기초한 연산을 수행한 뒤 비슷한 array D 에 해당하는 data vector

를 생성합니다. 하지만 array 와 vector, 그리고 다른 복합된 연산에 대한 정밀한 약속은 좀 더 주의 기울여야 합니다.

#### Node 42. Mixed vector and array arithmetic. The recycling rule

array 와 vector 의 복합연산이 element-wise 방식에 어떻게 영향을 미치는가는 다소 복잡하며, 레퍼런스로부터 이를 찾기는 매우 어렵습니다. 하지만 우리는 경험을 통하여 다음의 신용할 만한 가이드를 찾았습니다.

- \* expression은 왼쪽에서 오른쪽으로 읽혀집니다.
- \* 임의의 짧은 vector는 자기자신의 반복에 의해서 다른 vector의 길이를 자동으로 맞춥니다.
- \* 짧은 vector와 array가 만났을 경우만, array 는 반드시 모두 같은 dim attribution 을 가져야 하거나 error 메시지가 출력됩니다.
- \* 임의의 vector가 matrix 혹은 array 보다 긴 경우에는 예러가 출력됩니다.
- \* If array structures are present and no error or coercion to vector has been precipitated, the result is an array structure with the common dim attribute of its array operands 만약 array structures가 존재하는 동시에 어떠한 예러나 coercion이 vector를 촉진 시키지 않았다면, 그 결과는 array structure과 common dim 속성의 array operands입니다.(I'm not too sure how to interpret this sentence☹)

#### Node 43. The outer product of two arrays

array 에 있어서 중요한 연산자는 outer product 입니다. 만약 a 와 b 가 numeric array 라면, 그들의 outer product 는 이것들의 dimension vector 를 이음으로 하여 얻어진 dimension vector 를 가진 array 를 생성하며, 이것의 data vector 는 모든 가능한 a 와 b 의 data vector 의 element 들의 곱으로 얻어집니다. 이때 order는 매우 중요합니다. outer product 는 특수한 연산자인 %o% 에 의해서 이루어집니다.

```
> ab <- a %o% b
```

또 다른 방법으로는

```
> ab <- outer(a, b, "*")
```

multiplication 함수는 임의의 두 변수를 가진 함수로 대체되어 집니다. 예를 들면 우리가  $f(x; y) = \cos(y)/(1+x^{\{2\}})$  라는 함수를 R 의 vectors x 와 y 에 의해서 정의되어진 일반적인 x 와 y 의 좌표평면상에서 구하길 원한다면, 다음과 같이 할 수 있습니다.

```
> f <- function(x, y) cos(y)/(1 + x^2)
> z <- outer(x, y, f)
```

특히 두개의 vectors 에 대한 outer product 는 doubly subscripted array이며, (즉, 이것은 matrix 이며, 이것의 rank 는 최대 1 이어야 합니다.) 여기서 outer product 는 당연히 non-commutative 입니다. 이것에 대한 더 자세한 사항은 Chapter 10[Writing your own functions] 에서 찾아볼수있습니다.

#### Node 44. An example: Determinants of 2 by 2 single-digit matrices

다소 인위적이지만 이해가 쉬운 예제를 들면, 모든 entry 가 음이 아닌 정수인 2-by-2 matrix 들의 ,  $[a, b; c, d]$ , 의 가능한 determinant 들을 구한다고 가정해 봅시다. 문제는 어떻게 이런 형태의 모든 matrix 에 대해서 모든 determinant 인  $ad-bc$  들을 구할 수 있는가 입니다. 그리고 이것은 high density plot 과 같이 각각의 값이 가지고 있는 frequency 를 의미합니다. 만약 0 부터 9 까지의 정수가 개별적으로 선택되어지고, uniformly 뽑힐 확률을 가지고 있다면, determinant 에 대한 distribution 을 구할 수 있습니다.

이것은 outer() 함수를 이용하여 매우 편리하게 풀어낼 수 있습니다.

```
> d <- outer(0:9, 0:9)
> fr <- table(outer(d, d, "-"))
> plot(as.numeric(names(fr)), fr, type="h",
       xlab="Determinant", ylab="Frequency")
```

determinant 값들의 범위를 찾아내기 위해서 frequency table 의 names attribute 를 numeric 으로의 변형이 필요하다는 것에 주의하세요. 이 문제를 풀어가기 위한 가장 분명한 방법은 for loop 를 이용하는 것입니다. 이것은 Chapter 9 [Loops and conditional execution] 에서 찾아볼수 있습니다.

#### Node 45. Generalized transpose of an array

aperm(a, perm) 함수는 array a 를 치환하는데 사용합니다. argument perm 은 반드시 양의 정수  $(1, 2, \dots, k)$  의 순열 이어야 합니다. 여기에서 k 는 a 의 subscript 의 개수입니다. 이것의 결과는 a 의 크기와 동일하지만, perm[j] 에의해서 j-th dimension 을 가지게 됩니다. 이것의 연산을 쉽게 생각하는 방법은 matrix 에 대한 transposition 의 일반화 라고 생각하면 됩니다. 만약, A 가 doubly subscripted array 라면,

```
> B <- aperm(A, c(2,1))
```

이것은 단순히 A 에 대한 transpose 입니다. 이러한 특수한 경우에 대해서 t() 라는 함수가 제공 되기 때문에 간단히  $B <- t(A)$  라고 쓸 수 있습니다.

## Node 46. Matrix facilities

위에서 언급한 것과 같이 matrix 는 단순히 두 개의 subscript 를 가진 array 이지만 이 사실은 매우 중요하기 때문에 더 깊게 다루어야 합니다. R 은 matrix 만을 위한 많은 기능들을 제공하고 있습니다. 예를 들면, `t(X)` 은 위에서 언급한 것과 같이 matrix transpose 함수입니다. `nrow(A)` 와 `ncol(A)` 함수는 matrix A 의 행 과 열의 개수를 보여줍니다.

## Node 47. Matrix multiplication

연산자 `% * %` 는 matrix multiplication 을 계산하는데 사용합니다. `n-by-1` 또는 `1-by-n` matrix 는 물론 `n-vector` 라고 하는 것이 더 적합합니다. 반대로 matrix multiplication 에서 나타나는 vector 는 자연스럽게 row 또는 column vector로 나타납니다. 만약 A 와 B 가 같은 크기를 가지는 square matrix 라면

```
> A * B
```

는 elementwise product 입니다.

```
> A %*% B
```

는 matrix product 입니다. 여기에서 만약 x 가 vector 이면,

```
> x %*% A %*% x
```

은 quadratic form 입니다.

`crossprod()` 이라는 함수는 crossproduct 인데 `crossprod(X, y)` 는 `t(X) % * % y` 와 동일하며 이것이 더 효과적입니다. `crossprod()` 함수에서 두번째 argument 가 생산되면 첫번째와 동일한 결과를 얻을 수 있습니다.

`diag()` 함수는 사용되는 argument 에 따라서 여러 가지 의미를 가집니다. 만약 v 가 vector 라면, `diag(v)` 함수는 diagonal matrix 를 생성합니다. 반면에 M 이 matrix 이면, `diag(M)` 함수는 M 의 diagonal elements 을 취한 vector 를 돌려줍니다. 이것은 MATLAB 에서 사용되는 `diag()` 라는 함수와 동일합니다. 또한, 다소 혼동스럽지만, 만약 k 가 single number 라면, `diag(k)` 는 k-by-k identity matrix 를 생성합니다.

## Node 48. Linear equations and inversion

Linear equation 은 matrix multiplication 의 inversion 이 이용됩니다.

```
> b <- A %*% x
```

여기에서 b 와 A 는 주어져 있습니다. 그리고 우리는 vector x 를 linear equation system 의 solution 이라고 합니다.

```
> solve(A, b)
```

이것은 linear equation system 을 풀어낸 뒤 x 의 값을 돌려줍니다. linear algebra 에서  $x = A^{-1}b$  라 표시되고  $A^{-1}$  은 A 의 inverse 입니다. 이것은 단순히 `solve(A)`

로 구해집니다. 하지만 이것이 필요한 경우는 극히 드뭅니다. 수치적으로 `x <- solve(A) % * % b` 를 계산하는 것은 `solve(A,b)` 를 이용하는 것보다 비효율적이며, 완벽하지 못합니다. Quadratic 형태를 가지는  $x^T A^{-1} x$  는 multivariate 계산에 자주 사용되고 이것은 A 의 inverse 를 구하기 보다는 `x % * % solve(A, x)` 로 사용합니다.

## Node 49. Eigenvalues and eigenvectors

`eigen(Sm)` 은 symmetric matrix `Sm` 에 대한 eigenvalues 와 eigenvectors 를 계산합니다. 이 함수의 결과는 values 와 vectors 라는 names 를 가진 list 를 반환합니다.

```
> ev <- eigen(Sm)
```

이것은 `ev` 에 이 list 를 assignment 할 것입니다. 그리고 `ev$ val` 은 `Sm` 의 eigenvalue 의 vector 를 반환하고, `ev$ vec` 은 이에 상응하는 eigenvectors 를 보여줄 것입니다. 우리는 단순히 eigenvalues 만을 필요로 한다면,

```
> evals <- eigen(Sm)$values
```

라고 입력하고, 이것은 eigenvalues 만을 저장한뒤, 두번째 요소는 무시할 것입니다.

```
> eigen(Sm)
```

이것은 두 개의 요소가 모두 names 와 함께 출력될 것입니다. 사이즈가 큰 matrix 의 경우에 불필요한 eigenvectors 의 계산을 피하기 위해서

```
> evals <- eigen(Sm, only.values = TRUE)$values
```

를 사용합니다.

## Node 50. Singular value decomposition and determinants

`svd(M)` 함수는 argument 로서 임의의 matrix `M` 을 가지며, 이것은 `M` 에 대한 singular value decomposition 을 수행합니다. 이것은 `M` 과 동일한 열의 개수를 지닌 orthonormal matrix `U` 를 가지고 있습니다. 그리고 두 번째 matrix 인 `V` 는 `M` 의 행의 개수와 동일한 orthonormal matrix 입니다. 그리고 diagonal matrix `D` 의 elements 는 양수 입니다. 이것은  $M = U \% \% D \% \% t(V)$  입니다. 이 세가지 `d`, `u`, `v` 들이 `svd(M)` 의 결과로서 list 에 저장됩니다.

만약 `M` 이 square 라면,

```
> absdetM <- prod(svd(M)$d)
```

이것은 `M` 의 determinant 의 절대값입니다. 만약 이 계산과정이 다양한 matrix 에 대해서 자주 필요하다면, R function 으로 다음과 정의 할 수 있습니다.

```
> absdet <- function(M) prod(svd(M)$d)
```

이제 `absdet()` 을 또 하나의 R 함수로서 사용이 가능합니다. 이것과 같이 당신은 아마도 square matrix 의 trace 를 찾아주는 `tr()` 이라는 함수를 쓰고 싶어 할 수 있습니다. [Hint : 당신은 loop 이 필요하지 않습니다. `diag()` 함수를 다시 살펴보세요]

R 에는 determinant 를 계산하는 `det` 이라는 내장함수가 있습니다. 또한 `sign` 과 `modulus` 를 동시에 제공하는 `determinant` 라는 함수도 존재합니다.

## Node 51. Least squares fitting and the QR decomposition

`lsfit()` 이라는 함수는 least square fitting 의 결과를 list 에 저장합니다.

```
> ans <- lsfit(X, y)
```

이것은 `y` 가 observations 의 vector 이고, `X` 가 design matrix 일 때 least square fitting 을 한 결과를 돌려줍니다. regression diagnostic 에 대한 `ls.diag()` 함수가 이어서 사용될 수 있으

며, 이것에 대한 더 자세한 사항은 help를 참조하세요. Regression modelling을 하기 위해서 당신은 lm() 이라는 함수를 ls.fit() 보다 선호할 것입니다. 이는 Section 11.2 [Linear models] 에 자세히 나와있습니다.

이와 비슷하게 qr() 이라는 함수도 동일한 역할을 수행합니다.

```
> Xplus <- qr(X)
> b <- qr.coef(Xplus, y)
> fit <- qr.fitted(Xplus, y)
> res <- qr.resid(Xplus, y)
```

이것은 X 의 range 에 y 로부터의 orthogonal projection 을 계산하여 fit 에 저장합니다. res 안에는 orthogonal complement 에 대한 projection 을 계산된 것입니다. projection 에 대한 coefficient vector 는 b 에 저장되며, b 는 MATLAB 에서 사용되는 backslash 의 결과입니다. X 가 반드시 full column rank 일 필요는 없습니다. Redundancy는 발견되고 제거될 것입니다. 이것은 least square 계산을 이전의 방식을 대체합니다. 이것은 유용한 개념이기도 하지만, 현재는 일반적으로 statistical model feature 로 대체되었습니다. 자세한 내용은 Chapter 11 [Statistical model in R] 에 나와있습니다.

## Node 52. Forming partitioned matrices, cbind() and rbind()

matrices 는 다른 vectors 또는 matrices 로부터 cbind() 와 rbind() 로부터 생성될 수 있습니다. 간단히 cbind() 함수는 두개의 matrices 를 병렬로 합치는 것입니다. (column-wise) 그리고 rbind() 함수는 직렬로 합칩니다. (row-wise)

```
> X <- cbind(arg_1, arg_2, arg_3, ...)
```

cbind() 함수의 arguments 는 임의의 길이의 vector 또는 임의의 column size 인 matrix 가 될 수 있으나, 행의 수는 일치해야 합니다. 이것은 arguments 들의 행이 모두 합쳐진 arg\_1, arg\_2, \$ \ldots \$ 의 matrix 를 돌려줍니다. 만약 cbind() 에서 사용된 arguments 중에서 vector 가 있다면, 이것은 자동으로 matrix 의 열의 수만큼 확장됩니다.

rbind() 함수는 행에 대해서 위와 동일한 역할을 수행합니다. 어떠한 vector argument 가 있다면, 이것은 자동으로 row vector 처럼 확장됩니다.

X1 과 X2 가 같은 열의 개수를 지닌다고 가정하고, 1 이라는 column vector 를 함께 열방향으로 결합하여 matrix X 를 생성하고자 하면

```
> X <- cbind(1, X1, X2)
```

rbind() 와 cbind() 는 항상 matrix 상태입니다. 그러므로 cbind(x) 와 rbind(x) 는 vector x 가 column 또는 row matrix 로서 대체되는 가장 간단한 방법입니다.

## Node 53. The concatenation function, c(), with arrays



`cbind()` 와 `rbind()` 가 `dim attribute` 를 가지고 결합하는 함수라면, `c()` 함수는 `dim` 과 `dimnames` 를 모두 무시합니다. 이것은 때때로 더 유용하기도 합니다. `array` 를 `vector` 로 `coerce` 하는 방법은 `as.vector()` 라는 함수를 이용하는 것입니다.

```
> vec <- as.vector(X)
```

또, 단순히 한개의 `argument` 와 함께 `c()` 를 이용하여 비슷한 결과를 얻을 수 있습니다.

```
> vect <- c(X)
```

실은 이 두 가지에는 약간 다른 점이 있습니다. 이것들 중에서 어느 것을 선택하는가는 스타일의 차이일 뿐입니다.

#### Node 54. Frequency tables from factors

`factor` 는 그룹들로 나누어준다는 것을 기억하세요. 이와 비슷하게 쌍으로 묶인 `factor` 는 `two-way cross classification` 을 나타냅니다. `table()` 이라는 함수는 동일한 길이를 가진 `factors` 에 대해서 `frequency table` 을 형성합니다. 만약 `k factors arguments` 가 있다면 이것은 `k-way array` 의 `frequency table` 을 생성할 것입니다. `statef` 는 각각의 `data vector` 대해서 `state code`를 가지고 있는 `factor` 입니다.

```
> statefr <- table(statef)
```

이것은 `statefr` 안에 각각의 `state` 에 대하여 `frequency`를 저장합니다. 이 `frequency` 는 다시 정렬되고, `factor` 의 `attribute` 인 `level` 에 의해서 `labelling` 되어집니다.

```
> statefr <- tapply(statef, statef, length)
```

이것은 위의 방법보다 더 쉽게 동일한 결과를 줍니다. `incomef` 은 `income class` 를 나타내는 `factor` 입니다.

```
> factor(cut(incomes, breaks=35+10*(0:7))) -> incomef
```

이것은 다음과 같은 `two-way table of frequencies` 를 생성합니다.

```
> table(incomef, statef)
```

	act	nsw	nt	qld	sa	tas	vic	wa
(35,45]	1	1	0	1	0	0	1	0
(45,55]	1	1	1	1	2	0	1	3
(55,65]	0	3	1	3	2	2	2	1

(65,75] 0 1 0 0 0 0 1 0

## Node 56. Lists

R list 는 components 로 알려진 objects들의 집합체로 된 object 입니다. 이것은 구성요소의 mode 또는 type 이 동일할 필요가 없습니다. 예를 들면 list 는 numeric vector, logical value, matrix, complex vector, character array, function, 기타 등등으로 이루어질 수 있습니다. 아래의 명령어는 단순한 list 를 만들어 보는 것입니다.

```
> Lst <- list(name="Fred", wife="Mary", no.children=3,
             child.ages=c(4,7,9))
```

구성요소는 항상 숫자가 매겨지고 이로서 지칭됩니다. 만약 Lst 가 list 의 name 이고 4 개의 구성요소를 가진다고 한다면, 이것들은 Lst[[1]], Lst[[2]], Lst[[3]], 그리고 Lst[[4]] 로 가리켜집니다. Lst[[4]]가 vector 라고 가정한다면, Lst[[4]][1] 이것은 그것의 첫번째 entry 입니다.

Lst 가 list 라고 가정하면 length(Lst) 란 함수는 몇 개의 구성요소를 가지고 있는지를 알려줍니다. list 의 구성요소는 각각 이름이 주어지고, double square bracket([[ ]]) 을 대신하여 이 구성요소의 이름에 따라서 지칭됩니다.

```
> name$component_name
```

이것은 당신이 숫자를 기억하지 못할 때 참 편리한 방법입니다. 다음과 같이 간단한 예제를 살펴봅시다.

Lst\$name 은 Lst[[1]] 과 동일하며, 이것은 Fred 라고 합니다.

Lst\$wife 는 Lst[[2]] 이며, Mary 라고 부릅니다.

Lst\$child.ages[1] 은 Lst[[4]][1] 과 동일하고 4 살입니다.

게다가 리스트의 구성요소의 이름을 double bracket 안에 사용하는 것은 ( Lst[["name"]] ) Lst\$name 과 동일합니다 이것은 다른 변수 안에 구성요소의 이름을 저장할 때 매우 유용합니다.

```
> x <- "name"; Lst[[x]]
```

Lst[[1]] 과 Lst[1] 을 구분하는 것은 매우 중요합니다. '[[ \$ \ldots \$ ]]' 은 single element 를 선택하기 위한 연산자입니다. 반면에 '[\$ \ldots \$]' 은 subscripting 을 위한 연산자 입니다. 그러므로 먼저 것은 list 안에 존재하는 첫 번째 구성요소인 Lst를 의미하고, 나중의 것은 list 안에 존재하는 첫 번째 구성요소 Lst 인 sublist 의 구성요소인 element 를 의미합니다.

구성요소의 이름은 적은 개수의 글자를 사용하여 다른 것과 구분이 될 수 있도록 약어로 사용합니다. 그러므로 `Lst$coefficients` 는 `Lst$coe` 로서, `Lst$covariance`는 `Lst$cov` 로 표시합니다.

## Node 57. Constructing and modifying lists

새 `list` 는 `list()` 함수에 의해서 존재하는 `objects` 를 한데 묶어 생성할 수 있습니다.

```
> Lst <- list(name_1=object_1, ..., name_m=object_m)
```

이것은 `m` 구성요로 이루어진 `Lst`를 생성하는 것이며, 각각의 `argument` 에 `list` 의 `name` 을 주어줍니다. 이러한 `name` 없이 단순히 구성요소만을 나열할 수 도 있습니다. 구성요소들은 새로운 `list` 가 형성될 때 복사되어지는 것이기 때문에 `list` 가 생성될 때 본래의 구성요소들은 아무 영향을 받지 않습니다.

`List` 역시 다른 `subscript` 를 가지는 `object`들과 같이 자유롭게 구성요소를 추가할 수 있습니다.

```
> Lst[5] <- list(matrix=Mat)
```

## Node 58. Concatenating lists

`c()` 함수를 이용하여 주어진 `list` 를 결합하면 그 결과 역시 `list` 형태이며, 그것들의 구성요소들도 다 함께 일련의 순서로 `list` 로 묶입니다.

```
> list.ABC <- c(list.A, list.B, list.C)
```

`arguments` 로서의 `vector object` 를 `c()` 함수를 이용하여 모든 `arguments` 를 결합시키면 단일 `vector` 구조를 형성하며, 이런 경우 `dim` 과 같은 다른 `attribute` 는 모두 무시되는 점을 주의하세요.

## Node 59. Data frames

`data frame` 은 `data.frame` 이라는 `class` 를 가진 `list` 입니다.

\* 구성요소는 반드시 `vector` (`numeric`, `character`, 또는 `logical`), `factors`, `numeric matrix`, `list`, 또는 다른 `data frames`을 포함 하고 있어야 합니다.

\* `Matrices`와 `lists`, `data frames`는 순서대로 `coulumns`와 `elements`, `variables`를 가지고 있는 만큼 최대한 많은 `variables`를 새로운 `data frame`에 제공 합니다.

`Matrices`, `lists`, and `data frames` provide as many variables to the new data frame as they have columns, elements, or variables, respectively.

\* `Numeric vectors`와 `logicals`, `factors`는 있는 그대로를 포함 하고 있으며, `character vectors`는 `vector`안에 있는 `levels`가 `unique values`로 나타나는 `factors`로 강제 됩니다.

`Numeric vectors`, `logicals` and `factors` are included as is, and `character vectors` are

coerced to be factors, whose levels are the unique values appearing in the vector.

\* Data frame의 variable로 나타나 vector structures는 반드시 같은 length를 가지고 있어야 하고 matrix structures는 반드시 같은 row size를 가지고 있어야 합니다.

Vector structures appearing as variables of the data frame must all have the same length, and matrix structures must all have the same row size.

data frame 은 modes 와 attributes 가 분리된 matrix 와 같이 간주되어 사용이 가능합니다. 이것은 matrix 형태로 보여지고 편리하게 열과 행으로 matrix 를 index 할 수 있습니다

## Node 60. Making data frames

data frame 의 columns 위에 놓여 있는 제약을 만족하는 objects 는 data.frame() 함수를 이용하여 구성할 수 있습니다.

```
> accountants <- data.frame(home=statef, loot=incomes, shot=incomef)
```

list의 구성요소를 data frame 의 제약조건과 합치게 하기 위해서 as.data.frame() 함수를 사용하여 data frame 으로 coerce 할 수 있습니다. scratch 로부터 data frame 을 만드는 가장 간단한 방법은 read.table() 함수를 이용하는 것입니다. read.table() 함수는 외부 파일로부터 전체 data frame 을 읽어오는 것입니다. 이것은 Chapter 7 [Reading data from files]에서 자세히 다룹니다.

## Node 61. attach() and detach()

list 의 구성요소를 나타내기 위해서 accountants\$statef와 같이 사용된 \$ 기호가 언제나 편리한 것은 아닙니다. 유용한 기능은 list 혹은 data frame 의 구성요소를 어떻게 하던, 일시적으로 인용부호를 매번 사용하지 않고도 그들의 구성요소 아래에 변수처럼 보이게 하는 것입니다. attach() 함수는 argument 와 같이 list 또는 data frame 와 같은 database를 취합니다. lentils 라는 data frame 이 있고, 이것이 lentils\$u, lentils\$v, lentils\$w 라는 세 개의 변수를 가지고 있다고 한다면,

```
> attach(lentils)
```

이것은 data frame 내의 position 2 에서의 검색이 가능하도록 합니다. 변수 u, v, w 는 position 1 에서는 없고, data frame 내의 position 2 에 존재합니다. 여기에서 아래와 같은 assignment 는

```
> u <- v+w
```

data frame 내의 구성요소인 u 안에 값을 덮어 쓰지 않습니다. 이것은 또 다른 변수 u 를 working directory 안에 생성하고 position 1 검색에 놓입니다. 만약 data frame 내의 u 의 값을 변경하고 싶다면, 가장 단순한 방법은 다시 \$ 기호를 사용하는 것입니다.

```
> lentils$u <- v+w
```

하지만 구성요소 u 의 새로운 값은 data frame 이 detach 되고 다시 attach 될 때까지 갱신되지 않습니다. data frame 을 detach 하기 위해서는 다음의 함수를 사용하세요.

```
> detach()
```

보다 자세하게, detach() 함수는 position 2 에 있는 search path 를 떼어 놓습니다. 그러므로 더 이상 u, v, 그리고 w 가 더이상 존재하지 않습니다. position 2 보다 낮은 단계의 search path 가 주어진 숫자에 의해 분리가 될 수 있으나, name 을 사용하는 것이 더 안전합니다. 예를 들면 detach(lentils) 또는 detach("lentils") 입니다.

## Node 62. Working with data frames

많은 문제들을 동시에 같은 working directory 내에서 작업할 수 있게 하는 길은

\* data frame에 있는 잘 defined되고 separate 된 problem들에 관한 모든 variables를 적합한 정보를 주는 이름아래 모읍니다.

gather together all variables for any well defined and separate problem in a data frame under a suitably informative name

\* problem에 대해 작업 할 때, 적합한 data frame을 position 2에 attach 하고 level 1에 operation quantities와 temporary variable에 관한 working directory를 사용 하세요.

when working with a problem attach the appropriate data frame at position 2, and use the working directory at level 1 for operational quantities and temporary variables;

\* problem을 정리하기 전에, 미래의 reference로 간직 하고 싶은 variable이 있다면 \$ form of assignment를 사용하여 data frame으로 저장하고 detach();

before leaving a problem, add any variables you wish to keep for future reference to the data frame using the \$ form of assignment, and then detach();

\* 마지막으로 모든 원하지 않는 variables를 working directory로부터 지우고 남은 temporary variables들을 가능한 한 깨끗하게 보존 하세요.

finally remove all unwanted variables from the working directory and keep it as clean of left-over temporary variables as possible

이러한 방법들이 같은 directory 내에서 많은 문제를 한번에 다루는데 가장 편한 방법입니다.

## Node 63. Attaching arbitrary lists

attach() 는 directories 또는 data frame 을 search path 에 결합시키는 것 뿐만 아니라 object 의 다른 classes 들도 결합시키는 generic 함수입니다. 특히, mode 가 list 인 모든 object 는 같은 방법으로 결합될 수 있습니다.

```
> attach(any.old.list)
```

결합된 어떤 것이든지 `detach()` 함수를 이용하여 `position number` 또는 이름을 이용하여 분리되어질 수 있습니다.

#### Node 64. Managing the search path

`search()` 함수는 현재의 `search path` 를 보여주며, 어떤 `data frames` 과 `list, packages` 가 결합되고 분리되는지를 추적하는데 유용합니다.

```
> search()
[1] ".GlobalEnv" "Autoloads" "package:base"
```

이것은 `.GlobalEnv` 가 `workspace 1` 에 있음을 나타냅니다. `lentils` 를 결합시키고 나면

```
> search()
[1] ".GlobalEnv" "lentils" "Autoloads" "package:base"
> ls(2)
[1] "u" "v" "w"
```

위와 같은 결과가 보여지고 `ls(or objects)` 함수를 이용하여 `search path` 내의 어떠한 `contents` 역시 `position` 별로 확인할 수 있습니다. 마지막으로 우리는 `data frame` 을 분리하고, 그것이 `search path`로부터 제거되었는지를 확인할 수 있습니다.

```
> detach("lentils")
> search()
[1] ".GlobalEnv" "Autoloads" "package:base"
```

#### Node 65. Reading data from files

사이즈가 큰 `data object` 는 보통 `R session` 에서 키보드를 이용하여 입력하지 않고, 주로 외부 파일의 값들을 직접 읽어옵니다. `R` 입력기능은 단순하고, 이에 대한 요구사항은 다소 까다롭습니다. `R` 개발자는 `R` 의 입력시스템에 적합하도록 당신 스스로가 `Perl` 또는 다른 파일에디터를 이용하여 `input file` 을 가공할 수 있다고 가정했습니다. 일반적으로 이 과정은 매우 단순합니다.

우리는 모든 변수들이 `data frame` 안에 정리되도록 권장하며, 이렇게 되어 있다면, 모든 `data frame` 은 직접적으로 `read.table()` 함수에 의해서 불러들여집니다. `scan()` 함수같이 직접적으로 읽어올 수 있는 더 많은 주요 입력 함수가 존재합니다. 데이터를 `R` 로 집어넣고 빼어내는 것에 대한 자세한 사항은 `R Data Import/Export` 설명서를 참고하세요.

#### Node 66. The read.table() function

모든 `data frame` 을 직접적으로 읽어오기 위해서 외부파일은 보통 특정한 형태를 가집니다.



- \* 파일의 첫번째 줄은 **data frame** 안의 모든 변수들의 **name** 이 기록되어야 합니다.
- \* 모든 추가적인 행은 **raw label** 과 각각의 변수들의 값을 가집니다.

만약 파일의 첫번째줄의 양이 두번째보다 적다면, 이것은 강제적으로 첫번째줄에 맞춰질 것입니다. 읽혀진 파일의 **data frame** 은 다음과 같이 보여질 것입니다.

Input file form with names and row labels:

	Price	Floor	Area	Rooms	Age	Cent.heat	
01	52.00	111.0	830	5	6.2	no	
02	54.75	128.0	710	5	7.5	no	
03	57.50	101.0	1000	5	4.2	no	
04	57.50	131.0	690	6	8.8	no	
05	59.75	93.0	900	5	1.9	yes	

기본적으로 row label 을 제외한 numerical item 은 numeric 변수로 읽혀지고, 예제안에 보여지는 Cent.heat 같은 non-numeric 변수는 factor로 인식됩니다. 만약 필요하다면 이것은 바뀔 수도 있습니다. read.table() 이라는 함수는 data frame 을 직접으로 불러올 때 사용됩니다.

```
> HousePrice <- read.table("house.data")
```

종종 row labels 를 삭제하고 default label을 사용할 때가 있는데, 이 경우 파일은 row label column 을 삭제하고 다음과 같이 불러들입니다.

Input file form with names and row labels:

Price	Floor	Area	Rooms	Age	Cent.heat
52.00	111.0	830	5	6.2	no
54.75	128.0	710	5	7.5	no
57.50	101.0	1000	5	4.2	no
57.50	131.0	690	6	8.8	no
59.75	93.0	900	5	1.9	yes

data frame 은 다음과 같이 사용됩니다.

```
> HousePrice <- read.table("house.data", header=TRUE)
```

여기에서 header=TRUE 라는 옵션은 파일의 첫번째 줄이 heading 줄이라는 것은 지정하고 지정된 row label 이 없다는 것을 의미합니다.

Node 67. The scan() function

Data vectors가 같은 length이고 parallel로 읽힌다고 가정 해 봅시다. 또한 세 개의 vector들이 있다고 가정 하고, 첫 번째는 mode character, 나머지 두 vector는 mode numeric, 그리고 그 파일 이름은 'input.dat' 이라고 가정 합니다. 첫 번째 단계는 다음과 같이 세 개의 vectors를 list로 읽기 위해 scan()을 사용하는 것입니다.

Suppose the data vectors are of equal length and are to be read in parallel. Further suppose that there are three vectors, the first of mode character and the remaining two of mode numeric, and the file is 'input.dat'. The first step is to use scan() to read in the three vectors as a list, as follows

```
> inp <- scan("input.dat", list("",0,0))
```

두 번째 argument는 세 개의 vectors를 읽을 수 있게 하는 모드를 설정 하는 dummy list structure 입니다. Inp의 결과는 세 개의 vectors의 정보를 얻을 수 있는 components의 list입니다. Data items를 세 개의 separate vectors로 분리 하려면 다음과 같은 assignments를 이용 하세요.

The second argument is a dummy list structure that establishes the mode of the three vectors to be read. The result, held in inp, is a list whose components are the three vectors read in. To separate the data items into three separate vectors, use assignments like

```
> label <- inp[[1]]; x <- inp[[2]]; y <- inp[[3]]
```

더 편리하게는, dummy list는 이름이 정해진 components를 가질 수 있습니다. 경우에 따라 그 이름은 vectors의 정보를 얻기 위해 접근 하도록 사용 할 수도 있습니다. 예를 들어,

More conveniently, the dummy list can have named components, in which case the names can be used to access the vectors read in. For example

```
> inp <- scan("input.dat", list(id="", x=0, y=0))
```

만약 variables에 하나씩 액세스 하고 싶다면, 그것들은 working frame안에 있는 variables에 re-assigned를 하거나

If you wish to access the variables separately they may either be re-assigned to variables in the working frame:

```
> label <- inp$id; x <- inp$x; y <- inp$y
```

그 list는 search path에 있는 position 2에 첨부 될 수도 있습니다 (page 28, Section 6.3.4 에 있는 [Attaching arbitrary lists]를 참조 하세요). 만약 두 번째 argument가 single value지만 list가 아니고 single vector의 정보를 얻는 것이라면 모든 components들은 반드시 dummy value와 같은 mode 여야만 합니다.

or the list may be attached at position 2 of the search path (see Section 6.3.4

[Attaching arbitrary lists], page 28). If the second argument is a single value and not a list, a single vector is read in, all components of which must be of the same mode as the dummy value.

```
> X <- matrix(scan("light.dat", 0), ncol=5, byrow=TRUE)
```

좀 더 복잡한 input facilities들이 있는데 이것들은 manuals에 좀 더 자세하게 설명 되어 있습니다.

There are more elaborate input facilities available and these are detailed in the manuals

#### Node 68. Accessing builtin datasets

대략 100여 개의 데이터 셋이 R 과 함께 제공됩니다(dataset package). 그리고 다른 것들 또한 패키지 안에서 제공됩니다(R 과 함께 제공되는 recommended packages 포함). 데이터 셋의 목록을 보고 싶으시다면

```
data()
```

R version 2.0.0부터 R 에서 제공되어지는 모든 데이터 셋은 name을 이용해 찾을 수 있지만, 많은 패키지들이 아직도 R 에 데이터 셋을 불러오기 위해서 기존의 방법을 사용하고 있습니다. 예를 들면,

```
data(infert)
```

그리고 이것은 아직도 표준 패키지에서도 사용됩니다. 대부분의 경우 이것은 같은 이름을 가지는 R object 를 불러들여옵니다.

#### Node 70. Editing data

data frame 또는 matrix 가 불러들여오면, edit 은 수정을 하기위해 독립적인 스프레드시트와 같은 환경을 불러옵니다. 이것은 데이터 셋이 읽혀 올 때 작은 변화를 줄 때 매우 유용합니다. 수정을 위한 명령어는

```
> xnew <- edit(xold)
```

이 명령어는 당신의 데이터 셋 xold 를 수정하게 해줍니다. 그리고 변경된 object 를 xnew에 할당합니다. 만약에 본래의 데이터 셋 xold 를 바꾸고 싶다면, 간단하게 fix(xold) 를 사용하십시오. 이것은 xold <- edit(xold) 와 동일합니다.

```
> xnew <- edit(data.frame())
```

스프레드창을 통해서 새로운 데이터를 입력하고 싶다면 위의 명령어를 입력하세요.

## Node 72. R as a set of statistical tables

R 의 **편리한 점은** 포괄적인 statistical table 을 제공하는 것입니다. cumulative distribution function \$ P(X \leq x)\$, probability density function, quantile function (주어진 q, 가장 작은 x, \$ P(X \leq x) > q\$), 그리고 distribution 으로부터의 simulation 을 위한 함수들이 제공됩니다.

Distribution	R name	additional arguments
beta	beta	shape1, shape2, ncp
binomial	binom	size, prob
Cauchy	cauchy	location, scale
chi-squared	chisq	df, ncp
exponential	exp	rate
F	f	df1, df2, ncp
gamma	gamma	shape, scale
geometric	geom	prob
hypergeometric	hyper	m, n, k
log-normal	lnorm	meanlog, sdlog
logistic	logis	location, scale
negative	binomial	nbinom size, prob
normal	norm	mean, sd
Poisson	pois	lambda
Student's	t	t df, ncp
uniform	unif	min, max
Weibull	weibull	shape, scale
Wilcoxon	wilcox	m, n

Prefix 는 density 에 대해서 d, CDF 에 대해서 p, quantile function 에 대해서 q, random deviation 시뮬레이션에 **대해서는** r 을 사용합니다. 첫번째 argument 는 density 함수에 대해서는 x, CDF 는 q, quntile function 은 p, 그리고 simulation 에 대해서는 n 입니다. (rhyper, rwilcox 그리고 nn은 예외입니다.) 모든 케이스는 아니지만, non-centrality parameter ncp 도 가능합니다. 이것은 online help 를 참조하세요.

CDF와 quantile function 는 모두 logical argument 인 lower.tail 과 log.p 그리고 density function 은 log 를 가지고 있습니다. 이것은 cumulative harzard function, \$ H(t) = -\log\{ 1- F(t)\} \$, 를

-pxxx(t, ..., lower.tail = FALSE, log.p=TRUE)

통해서 얻을 수 있게 합니다. 또한 이것은 더욱 정교한 `log-likelihoods`(by `dxxx($ \dots $, log=TRUE)`), 를 바로 얻을 수 있게 합니다. 게다가 `ptukey` 와 `qtukey` 라는 함수는 `normal distribution` 으로부터 `samples` 들의 `studentized` 된 `range` 에 대한 `distribution` 을 얻게 합니다.

여기 몇 개의 예제를 제공합니다.

```
> ## 2-tailed p-value for t distribution
> 2*pt(-2.43, df = 13)
> ## upper 1% point for an F(2, 7) distribution
> qf(0.01, 2, 7, lower.tail = FALSE)
```

### Node 73. Examining the distribution of a set of data

주어진 `univariate` 데이터 셋에 대해서 이것의 `distribution` 을 할 수 있는 방법이 많이 있습니다. 가장 단순한 방법은 `numbers` 를 확인해 보는 것입니다. `summary` 와 `fivenum` 이라는 함수는 약간 다른 요약이며, `stem` (a stem and leaf plot) 을 통하여 숫자를 표시해 볼 수 있습니다.

```
> attach(faithful)
> summary(eruptions)
Min. 1st Qu. Median Mean 3rd Qu. Max.
1.600 2.163 4.000 3.488 4.454 5.100
> fivenum(eruptions)
[1] 1.6000 2.1585 4.0000 4.4585 5.1000
> stem(eruptions)
```

소수점 자리는 | 표시 왼쪽에 있는 한 자리입니다.

The decimal point is 1 digit(s) to the left of the |

```
16 | 070355555588
18 | 000022233333335577777777888822335777888
20 | 00002223378800035778
22 | 0002335578023578
24 | 00228
26 | 23
28 | 080
30 | 7
32 | 2337
34 | 250077
36 | 0000823577
38 | 2333335582225577
```

```
40 | 0000003357788888002233555577778
42 | 03335555778800233333555577778
44 | 02222335557780000000023333357778888
46 | 0000233357700000023578
48 | 00000022335800333
50 | 0370
```

stem-and-leaf plot 은 histogram 과 비슷합니다. 그리고 R 은 histogram 을 그려주는 hist 라는 함수를 가지고 있습니다.

```
> hist(eruptions)
## make the bins smaller, make a plot of density
> hist(eruptions, seq(1.6, 5.2, 0.2), prob=TRUE)
> lines(density(eruptions, bw=0.1))
> rug(eruptions) # show the actual data points
```

좀더 정교한 density plot 은 density 라는 함수를 통해서 얻어집니다. 그리고 우리는 density 를 이용하여 line 을 추가할 수 있습니다. bw 라는 함수는 bandwidth 를 trial-and-error 를 통해서 찾아냅니다. 이것에 대한 기본값은 주로 너무 많이 smoothing 을 하게 됩니다. (이것이 주로 우리가 관심있는 interesting density입니다.) (더 나은 자동화 방법을 통하여 bandwidth 를 선택하는 것이 가능합니다. 이 예제에서는 bw="SJ" 를 이용하여 더 나은 결과를 얻을 수 있습니다.)

우리는 또한 empirical **cumulative** distribution function 을 ecdf 함수를 이용하여 생성할 수 있습니다.

```
> plot(ecdf(eruptions), do.points=FALSE, verticals=TRUE)
```

이 distribution 은 **다른** 어떠한 standard distribution 과도 **확실하게 다릅니다**. right-hand mode 는 어떨까요? 3분 이상 지속되는 화산폭발을 살펴봅시다. 우리가 normal distribution 과 일치시켜보고, 찾아낸 CDF 를 그 위에 overlay **해보겠습니다**.

```
> long <- eruptions[eruptions > 3]
> plot(ecdf(long), do.points=FALSE, verticals=TRUE)
> x <- seq(3, 5.4, 0.01)
> lines(x, pnorm(x, mean=mean(long), sd=sqrt(var(long))), lty=3)
```

quantile-quantile (Q-Q) plot 또한 우리가 이것을 좀더 주의 깊게 확인하는 것을 도와줍니다.

```
par(pty="s") # arrange for a square figure region
```

```
qqnorm(long); qqline(long)
```

이것은 적절하게 일치하는 **것** 같습니다. 그러나 right tail 이 normal distribution 보다 짧은 것을 볼 수 있습니다. **다음은 simulated data** 를 가지고 t distribution 과 비교해보겠습니다.

```
x <- rt(250, df = 5)
qqnorm(x); qqline(x)
```

이것은 보통 우리가 기대했던 normal 보다 다소 길게 **보이는** 것을 알 수 있습니다. 우리는 q-q plot 을 generating distribution으로부터 다음을 통하여 구할 수 있습니다.

```
qqplot(qt(ppoints(250), df = 5), x, xlab = "Q-Q plot for t dsn")
qqline(x)
```

마지막으로 우리는 normality 의 일치성에 대해서 좀더 formal test를 해보도록 하겠습니다. R 은 Shapiro-Wilk test를 제공합니다.

```
> shapiro.test(long)
```

Shapiro-Wilk normality test

```
data: long
W = 0.9793, p-value = 0.01052
```

그리고 Kolmogorov-Smirnov test 역시 제공됩니다.

```
> ks.test(long, "pnorm", mean = mean(long), sd = sqrt(var(long)))
```

One-sample Kolmogorov-Smirnov test

```
data: long
D = 0.0661, p-value = 0.4284
alternative hypothesis: two.sided
```

#### Node 74. One- and two-sample tests

여지까지 normal distribution 에 대한 a single sample 에 대해서 비교해보았습니다. **이제** 두 개의 samples 을 비교하는 일반적인 방법에 대해서 알아보겠습니다. 아래에 묘사된 방법은 R 에서 일반적으로 stats 패키지로 부터 불러오는 전형적인 테스트 입니다.

Method A: 79.98 80.04 80.02 80.04 80.03 80.03 80.04 79.97



```
80.05 80.03 80.02 80.00 80.02
Method B: 80.02 79.94 79.98 79.97 79.97 80.03 79.95 79.97
```

Box plot 은 두 개의 samples 에 대한 그래픽적인 비교를 보여줍니다.

```
A <- scan()
79.98 80.04 80.02 80.04 80.03 80.03 80.04 79.97
80.05 80.03 80.02 80.00 80.02
```

```
B <- scan()
80.02 79.94 79.98 79.97 79.97 80.03 79.95 79.97
```

```
boxplot(A, B)
```

여기서, 첫 번째 그룹이 두 번째 그룹보다 더 높은 결과를 보여줍니다.

두 samples 의 means 에 대한 일치성을 비교해 보기 위해서는 unpaired t-test 를 사용합니다.

```
> t.test(A, B)
```

Welch Two Sample t-test

```
data: A and B
t = 3.2499, df = 12.027, p-value = 0.00694
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 0.01385526 0.07018320
sample estimates:
mean of x mean of y
80.02077 79.97875
```

이것은 데이터가 normality 를 따른다고 가정할 때, 통계학적으로 의미있게 다른 것임을 보여주고 있습니다. 기본적으로 R 함수는 두 samples 에 대한 variances 가 동일하다고 가정하지 않습니다. 두 samples 가 normality 를 따르는 population 으로 부터 추출되었다고 가정한다면, F test 를 이용하여 variances 가 동일한지 확인해 볼 수 있습니다.

```
> var.test(A, B)
```

F test to compare two variances

data: A and B

$F = 0.5837$ , num df = 12, denom df = 7, p-value = 0.3938

alternative hypothesis: true ratio of variances is not equal to 1

95 percent confidence interval:

0.1251097 2.1052687

sample estimates:

ratio of variances

0.5837405

이 결과는 두 variances 가 다르다고 할 통계학적인 근거가 없습니다. 그래서 우리는 variance 가 동일하다고 가정하는 전형적인 t-test 를 사용할 수 있습니다.

```
> t.test(A, B, var.equal=TRUE)
```

#### Two Sample t-test

data: A and B

$t = 3.4722$ , df = 19, p-value = 0.002551

alternative hypothesis: true difference in means is not equal to 0

95 percent confidence interval:

0.01669058 0.06734788

sample estimates:

mean of x mean of y

80.02077 79.97875

이 모든 테스트는 두 samples 가 normality 를 따른다고 가정합니다. two-sample Wilcoxon (or Mann-Whitney) test 는 null hypothesis 가 참이라고 가정할 때 공통된 continuous distribution 을 가졌다고 가정합니다.

```
> wilcox.test(A, B)
```

#### Wilcoxon rank sum test with continuity correction

data: A and B

$W = 89$ , p-value = 0.007497

alternative hypothesis: true location shift is not equal to 0

Warning message:

Cannot compute exact p-value with ties in: wilcox.test(A, B)

여기에서 warning 이 발생하는데, 이것은 각각의 sample 내에 동일한 값이 여러 개 존재하기 때문입니다. **이럴 경우 이런 데이터는 discrete distribution으로부터 발생되었다고 가정되기 때문이었습니다.** 이것은 아마도 rounding 때문에 발생했을 것입니다.

두 samples를 그래픽적으로 비교하는 데는 여러 가지 방법이 있습니다. 우리는 이전에 a pair of boxplots 를 보았습니다.

```
> plot(ecdf(A), do.points=FALSE, verticals=TRUE, xlim=range(A, B))
> plot(ecdf(B), do.points=FALSE, verticals=TRUE, add=TRUE)
```

이것들은 두 개의 empirical CDFs 을 보여줄 것이며, qqplot 은 두 samples 에 대한 Q-Q plot 을 보여줄 것입니다. Kolmogorov-Smirnov test 는 동일한 continuous distribution 이라는 가정하에서 두 가지의 ecdf 의 세로방향으로 가장 큰 distance 검사합니다.

```
> ks.test(A, B)
```

Two-sample Kolmogorov-Smirnov test

data: A and B

D = 0.5962, p-value = 0.05919

alternative hypothesis: two-sided

Warning message:

cannot compute correct p-values with ties in: ks.test(A, B)

Node 76. roped expressions

R 은 function 혹은 expression 형태의 command 를 입력하여 결과를 얻어내는 **측면에서 expression language** 라고 할 수 있습니다.

**하나의 assignment조차도 결과가 value가 부여된 expression이고 어떠한 expression이 사용될 수 있는 어느 곳에서나 사용될 수 있습니다. 특히 multiple assignments가 가능 합니다.**

Even an assignment is an expression whose result is the value assigned, and it may be used wherever any expression may be used; in particular multiple assignments are possible.

commands 는 `expr_1, $ \ldots $, expr_m` 과 같이 braces로 묶을 수 있습니다. brace 안의 맨 마지막 expression 에서 얻어지는 결과가 그룹으로 묶인 command 의 결과가 값이 될 것입니다. 그룹 또한 expression 의 하나로 간주됩니다. 예를 들어, 그룹 자체가 또 다른 braces 내에 존재하여 더 큰 expression 의 일부로서 사용될 수 있습니다.

Node78.

Conditional execution: if statements

language의 조건문은 다음의 형태를 가집니다.

```
if (expr_1) expr_2 else expr_3
```

expr\_1 은 반드시 한 개의 logical value 를 산출하고, 전체의 expression 의 결과는 분명해집니다. short-circuit 이라고 불리는 연산자 && 과 || 는 종종 if 문 속의 condition 의 일부분으로 사용됩니다. & 과 | 이 vector 에 대해서 element-wise 적 특성을 가지는 것에 반해, && 과 || 은 길이가 1 인 vector 에 사용됩니다. 그리고 이것은 필요하다면 두 번째 argument 에 이용됩니다. if/else 의 vectorized 된 형식으로 ifelse 가 있습니다. 이것은 ifelse(condition, a, b) 의 형식을 가지고 있고 **그것의 가장 긴 argument의 길이의 vector를 condition[i]가 맞다면 elements a[i]와 함께 돌려주고 다른 경우에는 b[i]를 돌려 줍니다.**

There is a vectorized version of the if/else construct, the ifelse function. This has the form ifelse(condition, a, b) and returns a vector of the length of its longest argument, with elements a[i] if condition[i] is true, otherwise b[i].

Node 79. Repetitive execution: for loops, **repeate** and while

```
> for (name in expr_1) expr_2
```

의 형태를 가지는 for loop **이** 있습니다. 여기에서 name 은 loop variable 입니다. expr\_1 은 vector expression (종종 sequence 1:20) 그리고 expr\_2 는 dummy name 에 관하여 씌여진 **sub-expression** 들로 **그룹을 이루어진 expression** 입니다. expr\_2 는 name 이 expr\_1 의 vector result 의 value 들을 통해서 name range 로서 반복적으로 계산 되어집니다.

예를 들면, ind 가 class indicator 의 vector 이고 우리는 classes 내의 y by x 의 plot 을 따로따로 생성하길 **원한다고** 하면, 첫 번째 대책은 factor 의 각각의 level에 해당하는 plot 들의 array 를 생산하는 coplot() 을 사용하는 것입니다. 이것을 하는 또 다른 방법은 모든 plots들을 한 화면에 집어 넣는 것입니다.

```
> xc <- split(x, ind)
> yc <- split(y, ind)
> for (i in 1:length(yc)) {
  plot(xc[[i]], yc[[i]])
  abline(lsfrit(xc[[i]], yc[[i]]))
}
```

split() 함수는 factor 에 의하여 구분 되어진 classes 들에 대하여 큰 vector 분리하여 얻어진 vector 들의 list 를 생성합니다. 이것은 주로 boxplot 과 관계된 매우 유용한 함수입니다. 이것에

대해서 더 알고 싶으시면 **help** 기능을 사용해 보세요.

경고 : R code 에서 사용되는 **for()** loop 는 종종 **complied** 언어들보다 적은 양을 가집니다. **whole object** 관점을 가진 코드는 더 깔끔하고 빠르게 수행될 가능성이 있습니다.

또 다른 형태의 **looping** 기능이 있습니다.

```
> repeat expr
```

그리고

```
> while (condition) expr
```

**break statement** 는 어떠한 **loop** 라고 종결시킵니다. 이것이 **repeat loop** 을 종료하는 유일한 방법입니다.

**next statement** 는 특정한 한 사이클만을 중지하고 그 다음으로 넘어가는데 사용됩니다. **control statement** 는 대부분 Chapter 10 [Writing your own function] 에서 다루어지는 내용과 관계가 있으며 많은 예제를 통하여 보여질 것입니다.

## Node 80. Writing your own functions

이 안내서를 따라오다 우리가 본 것과 같이 R language 는 사용자가 **mode function** 의 **object** 를 생성할 수 있도록 허락합니다. 이것들은 특수한 내부형태로 저장되어 있고, 나중의 **expression** 을 위해서 사용되는 **true R 함수들**입니다.

이러한 과정에서 R language 는 파워, 실용도, 그리고 깔끔한 코드를 얻을 수 있습니다. 유용한 함수를 쓰기 위해서 배우는 것은 당신이 R 을 사용함에 있어서 편안하고 생산적인 **주요 길중에** 하나입니다.

R 시스템의 부분으로서 제공되는 **mean()**, **var()**, **postscript()** 와 같은 대부분의 함수들이 R 로 썬어져 있다는 것이 강조되어야 합니다. 이것들은 실질적으로 사용자가 작성한 함수와 다를 바 없습니다. **function** 은 다음과 같은 형태로 **assignment** 뉘으로 썬 정의되어집니다.

```
> name <- function(arg_1, arg_2, ...) expression
```

**expression** 이란 값을 구하기 위하여 사용되어지는 **arg\_i** 라는 **arguments** 를 이용하는 그룹화된 **expression** 입니다. **expression** 의 값이 **function** 에게로 전달되어지는 값입니다. **function** 를 부르는 것은 보통 **name(expr\_1, expr\_2, \$ \ldots \$)** 라는 형태를 이용합니다. 그리고 함수호출은 호출을 부를 수 있는 적합한 곳이라면 어느 곳에서나 행해질 수 있습니다.

## Node 81. Simple examples

첫 번째 예제로 두 samples 의 t-statistic 을 계산하는 함수의 모든 과정을 생각해 봅시다.

물론 이것은 인위적인 예제이지만, 같은 결과를 얻을 수 있는 더 간편한 방법도 있습니다. 함수는 다음과 같이 정의 됩니다.

```
> twosam <- function(y1, y2) {  
  n1 <- length(y1); n2 <- length(y2)  
  yb1 <- mean(y1); yb2 <- mean(y2)  
  s1 <- var(y1); s2 <- var(y2)  
  s <- ((n1-1)*s1 + (n2-1)*s2)/(n1+n2-2)  
  tst <- (yb1 - yb2)/sqrt(s*(1/n1 + 1/n2))  
  tst  
}
```

정의된 함수를 이용하여 당신은 two samples t-test 를 다음과 같이 함수를 호출하여 계산할 수 있습니다.

```
> tstat <- twosam(data$male, data$female); tstat
```

두 번째 예제로, matrix  $X$  에서 column space 에 대하여 vector  $y$  의 orthogonal projection에 대한 coefficients 를 계산해내는 Matlab backslash command를 직접적으로 흉내내는 함수를 생각해봅시다. 이것은 원래 least squares estimation 에 의한 regression coefficients 를 구하는 것입니다. 이것은 본래  $qr()$  이라는 함수를 이용하여 이루어집니다. **하지만 자주** 이것을 직접적으로 이용하는 것은 다소 까다로우며, 다음과 같이 간단한 함수를 사용하는 것이 안정적입니다. vector  $y$  는  $n$ -by-1 이라고 주어져 있고,  $n$ -by- $p$  matrix  $X$  인 design matrix 역시 주어집니다. 그리고 나서,  $X^T X y$  는  $(X^T X)^{-1} X^T y$  과 같이 정의됩니다. 여기에서  $(X^T X)^{-1}$  은  $X^+$ 의 generalized inverse 입니다.

```
> bslash <- function(X, y) {  
  X <- qr(X)  
  qr.coef(X, y)  
}
```

이 object 가 생성되고 나면, 다음과 같은 statements 를 사용합니다.

```
> regcoeff <- bslash(Xmat, yvar)
```

lsfit() 이라는 전형적인 R 함수가 이 작업을 매우 잘 수행합니다. 사실은 이것은 함수  $qr()$  과  $qr.coef()$  를 이 작업의 일부분을 수행하기 위한 약간 반직관적인 방법입니다. 그러므로 만약 이

것이 자주 사용된다면, 이부분만을 따로 떼어내어 계산하는 것이 좋습니다. 만약 그렇다면, 우리는 이것을 `matrix binary operator` 로 만드는 것이 더 편리한 방법일 것입니다.

## Node 82. Defining new binary operators

우리는 이것을 `bslash()` 라는 다른 이름을 주었습니다.

`%anything%`

이것은 함수의 형태보다 다소 `binary operator` 로서 사용될 수 있습니다. `!` internal character 를 위해서 `!` 를 사용한다고 가정하면 함수의 정의는 아래와 같이 시작합니다.

```
> "%!%" <- function(X, y) { ... }
```

이때 `quote marks` 를 사용하는 것에 대해서 명시해야 합니다. `$ X\%!\%y$` 와 같이 사용될 수 있습니다.

`matrix multiplication` 연산자인 `$ \%*\%$` 과 `outer product matrix` 연산자인 `$ \%o\%$` 는 이러한 방법으로 정의된 `binary operator` 입니다.

## Node 83. Named arguments and defaults

Section 2.3 [Generating regular sequences] 에서 명시된 것과 같이 만약 함수에 불려진 arguments가 `name=object` 형태라면, 그들은 어느 순서로 주어도 무방합니다. 더 나아가, argument 의 순서는 unnamed, positional form, 그리고 named argument 의 순서로 나열됩니다. 만약 function fun1 이 다음과 같이 정의 된다면,

```
> fun1 <- function(data, data.frame, graph, limit) {  
[function body omitted]  
}
```

function 은 여러 가지 방법으로 실행될 것입니다. 예를 들면,

```
> ans <- fun1(d, df, TRUE, 20)  
> ans <- fun1(d, df, graph=TRUE, limit=20)  
> ans <- fun1(data=d, limit=20, graph=TRUE, data.frame=df)
```

은 모두 동일한 방법입니다.

대부분의 경우에 공통적으로 적용될 수가 있는 arguments가 있습니다. 기본적으로 정의된 arguments 의 값이 적절하다면, 이들은 함수 호출 시 삭제될 것입니다.



```
> fun1 <- function(data, data.frame, graph=TRUE, limit=20) { ... }
```

이것은 또한 아래와 같이 호출될 수 있습니다.

```
> ans <- fun1(d, df)
```

이것 역시 위의 세가지 경우와 일치합니다.

```
> ans <- fun1(d, df, limit=10)
```

이것은 default 값을 변경하는 것입니다. 기본값은 임의의 expression 이며, 같은 함수에 다른 arguments를 포함할 수도 있다는 점이 중요합니다. 또한, 여기에서 본 예제와 같이 constants 이라는 제약은 없습니다.

#### Node 84. The '...' arguments

또 다른 흔한 요구사항은 어떤 한 함수가 argument setting 을 또 다른 것으로 전달하는 것입니다. 예를 들면, 많은 그래픽 관련 함수는 par() 라는 함수를 이용하는데, plot() 같은 함수는 사용자가 그래픽 출력을 조정하는 par()라는 함수에 그래픽 관련 함수들을 전달합니다. par() 에 대해서 알고 싶으시면 Section 12.4.1 [The par() function] 을 살펴보세요. 이것은 함수 내에서 ...라고 표현되는 extra argument를 포함하므로 해결됩니다. 이는 다음과 같습니다.

```
fun1 <- function(data, data.frame, graph=TRUE, limit=20, ...) {  
[omitted statements]  
if (graph)  
par(pch="*", ...)  
[more omissions]  
}
```

#### Node 85. Assignments within functions

함수 내에 있는 어떤 arguments들은 local 이고 temporal 입니다. 그리고 함수가 종료되면 이 값은 잃어버립니다. 그러므로,  $X <- -qr(X)$  란 assignment 는 프로그램을 부를 때 argument 의 값에 어떠한 영향도 미치지 않습니다.

R assignments 의 범위를 다루는 규칙을 완전히 이해하기 위해서는 독자들은 frame 의 기호에 익숙해져야 합니다. 이것은 다소 발전된 개념이지만, 이 안내서에는 더 이상 다루지 않습니다.

만약 global 이고 permanent assignment 는 함수 내에서 의도되어지고, superassignment 라는 연산자를 이용합니다. 이는 <<- 또는 함수 assign()을 이용합니다. 이것에 대해 더 자세한

사항은 help 문서를 참고하세요. S-PLUS 사용자들은 <<- 가 R 에서 약간 다른 의미가 있다는

것을 눈치 채셨을 것입니다. 이것들에 대해서는 Section 10.7 [Scope] 를 살펴보세요.

### Node 87. Efficiency factors in block designs

좀더 완전한 이해를 위해서 block design 에서 efficiency factor 를 찾아내는 함수를 살펴보겠습니다. (어떤 부분들은 Section 5.3 [Index matrices] 에서 살펴본 적이 있습니다.)

block design은 blocks(b levels)와 varieties(v levels)라는 두 개의 factors 로 정의 됩니다. 만약 R 과 K 가 v-by-v replications matrix 그리고 b-by-b block size matrix 라면, N 은 b-by-v 인 incidence matrix 입니다. 그러면 efficiency matrix 는 matrix 의 eigenvalues 로 정의됩니다.

$$E = I_v - R^{-1/2} N^T K^{-1} N R^{-1/2} = I_v - A^T A$$

여기에서  $A = K^{-1/2} N R^{-1/2}$  입니다. 이것을 나타내는 함수는 아래와 같습니다.

```
> bdeff <- function(blocks, varieties) {  
  blocks <- as.factor(blocks) # minor safety move  
  b <- length(levels(blocks))  
  varieties <- as.factor(varieties) # minor safety move  
  v <- length(levels(varieties))  
  K <- as.vector(table(blocks)) # remove dim attr  
  R <- as.vector(table(varieties)) # remove dim attr  
  N <- table(blocks, varieties)  
  A <- 1/sqrt(K) * N * rep(1/sqrt(R), rep(b, v))  
  sv <- svd(A)  
  list(eff=1 - sv$d^2, blockcv=sv$u, varietycv=sv$v)  
}
```

이 경우에는 수치적으로 eigenvalue routine 을 사용하는 것보다 singular value decomposition 이용하는 것이 다소 효율적입니다. 함수의 결과는 첫 번째 구성요소와 같이 efficiency factors 뿐만 아니라 block 과 variety의 canonical contrast 에 대한 list 를 반환하는데 이러한 유용한 qualitative information 을 때때로 부가적으로 제공하기 때문입니다.

### Node 88. Dropping all names in a printed array

사이즈가 큰 matrices 또는 arrays 을 출력하는 목적으로 matrix 나 array 의 name 없이 중 close block form 을 사용하는 것이 유용합니다. dimnames attribute 를 제거하는 것은 이러한 결과에 아무 것도 하지 않습니다. 이것은 그저 주어진 dimnames attribute 를 단지 empty string 으로 만드는 것입니다. matrix X 를 출력하기 위해서는

```
> temp <- X  
> dimnames(temp) <- list(rep("", nrow(X)), rep("", ncol(X)))  
> temp; rm(temp)
```

아래의 코드는 wrap around 와 같이 같은 결과를 생성하기 위해서 no.dimnames() 라는 함

수를 이용하여 좀 더 편리한 방법을 제공합니다. 이것은 또한 다소 효율적이고 유용한 사용자 함수가 짧은지를 보여줍니다.

```
no.dimnames <- function(a) {  
  ## Remove all dimension names from an array for compact printing.  
  d <- list()  
  l <- 0  
  for(i in dim(a)) {  
    d[[l <- l + 1]] <- rep("", i)  
  }  
  dimnames(a) <- d  
  a  
}
```

이러한 함수가 정해지면, array 는 close format 형식으로 아래의 함수호출에 의해서 출력됩니다.

```
> no.dimnames(X)
```

이것은 특별히 값들보다 패턴을 살펴보기 위해서 큰 integer array 에 대해서 유용합니다.

## Node 89. Recursive numerical integration

함수는 recursive 가 가능하고 그 자신이 자신의 함수 내에서 재정의 될 수 있습니다. 그런데, 그런 함수 또는 변수들은 상위 evalution frame 에서 불러진 함수에 의해서 종속되지 않다는 것에 주의해야 합니다.

아래의 예제는 one-dimensional numerical integration 을 하는 일반적인 방법입니다. integrand 는 range 의 end point 와 middle point 에서 계산되어질 것입니다. 만약 one-panel trapezium rule 결과가 two-panel 과 충분히 매우 비슷하다면 나중의 값을 결과로서 돌려줍니다. 그렇지 않으면 같은 프로시저가 recursively 하게 각각의 패널에 대해서 이루어질 것입니다. 이 결과는 integrand 가 linear 가 아닌 형태의 regions에 있는 함수의 계산에 적합한 adaptive integration 입니다. 그런데 integrand 가 smooth 하고 계산하기 어려울 때는 다른 알고리즘들에 비해서 비용이 많이 듭니다.

이 예제는 R 프로그래밍에서 little puzzle 과 같이 부분적으로 주어졌습니다.

```
area <- function(f, a, b, eps = 1.0e-06, lim = 10) {  
  fun1 <- function(f, a, b, fa, fb, a0, eps, lim, fun) {  
    ## function 'fun1' is only visible inside 'area'  
    d <- (a + b)/2  
    h <- (b - a)/4  
    fd <- f(d)  
    a1 <- h * (fa + fd)  
    a2 <- h * (fd + fb)  
    if(abs(a0 - a1 - a2) < eps || lim == 0)  
      return(a1 + a2)  
    else {
```

```

return(fun(f, a, d, fa, fd, a1, eps, lim - 1, fun) +
fun(f, d, b, fd, fb, a2, eps, lim - 1, fun))
}
}
fa <- f(a)
fb <- f(b)
a0 <- ((fa + fb) * (b - a))/2
fun1(f, a, b, fa, fb, a0, eps, lim, fun1)
}

```

## Node 90. Scope

이 섹션에서는 이 문서에 실린 다른 부분들에 비해서 매우 기술적일 것입니다. 그런데 이것은 S-Plus 와 R 사이에서 중요한 다른 점 중에 하나 입니다. 함수의 몸체에서 쓰여지는 symbols 들은 크게 세가지 classes로 나뉩니다. formal parameters, local variables, 그리고 free variables 이 그것입니다. 함수에 있어서 formal parameter 는 함수의 argument list 에서 나타나는 그것입니다. 그것들의 값은 formal parameter 에 실제 function argument 가 묶임으로써 결정됩니다. local variables 은 함수의 몸체에서 expression 이 연산될 때마다 결정되어지는 값들입니다. formal parameter 와 local variable 이 아닌 것들을 free variable 이라고 합니다. free variables 는 local variable 이 되기도 합니다.

다음 예제를 살펴보도록 하겠습니다.

```

f <- function(x) {
y <- 2*x
print(x)
print(y)
print(z)
}

```

이 함수 내에서 x 는 formal parameter 이고 y 는 local variable 이며 z 는 free variable. 입니다. R에서 free variable binding 은 함수가 생성될 때의 환경 속에서 first looking 에서 결정되어집니다. 이것을 lexical scope 라고 합니다. 가장먼저 우리는 cube 라고 불리는 함수를 정의합니다.

```

cube <- function(n) {
sq <- function() n*n
n*sq()
}

```

sq 라는 함수 내에서 variable n 은 이 함수의 argument 가 아닙니다. 그러므로 이것은 free variable 입니다. scoping 규칙은 이것과 연관된 값을 확실히 하기 위해서 사용됩니다. S-Plus 의 static scope 를 따르면, n 이라고 명명된 global variable 과 연계된 것이 값입니다. R 에서의 lexical scope 에 따르면 이것은 cube 라고 불리는 함수의 parameter 입니다. 그 이유는 그것이 sq 라는 함수가 정의 될 때 variable n 이 active binding 을 하기 때문입니다. R 과 S-Plus 의 연산이 다른 점은 S-Plus 는 n 이라고 불리는 global variable 을 찾는 반면, R 은 먼저 n 이라

고 불리는 변수를 `cube` 라고 불리는 함수가 실행될 때 생성된 환경으로부터 찾습니다.

```
## first evaluation in S
```

```
S> cube(2)
```

```
Error in sq(): Object "n" not found
```

```
Dumped
```

```
S> n <- 3
```

```
S> cube(2)
```

```
[1] 18
```

```
## then the same function evaluated in R
```

```
R> cube(2)
```

```
[1] 8
```

lexical scope 는 함수에게 mutable state 를 제공합니다. 다음 예제를 통하여 우리는 R 이 어떻게 은행계좌를 모방하기 위해서 사용되어지는 가를 알 수 있습니다. 은행계좌를 관리하기 위해서는 balance 또는 total 이 필요하고, withdrawal 위한 함수, deposits을 위한 함수 그리고 current balance를 보여주기 위한 함수가 필요합니다.

이러한 세가지 함수를 account 안에 생성하고 그것들을 포함하는 list 를 돌려줍니다. account 가 실행되면 numerical argument 인 total 을 가져옵니다. 그리고 세가지 함수를 포함하는 list 를 내보냅니다. 이러한 함수는 total 을 포함하는 환경 속에 정의 되어 있기 때문에 그들은 이것의 값을 사용하게 됩니다.

special assignment 연산자인 `<<-` 은 total에 연관된 값을 변경하는데 사용됩니다. 이 연산자는 total 이라는 symbol 포함하는 environment 에 대해서 둘러쌀 때 기억하고, 그것이 그런 environment를 찾거나, 변경할 때, 오른쪽에 있는 값을 참조합니다.

만약 global 혹은 top-level 환경에서 total 이라는 symbol 을 찾지 않고 도달한다면 그때는 그 변수가 생성되고 할당될 것입니다. 대부분의 유저들은 `<<-` 로 global variable 을 생성하는데 생성하고 그것의 오른쪽에 있는 값을 할당할 것입니다. 또 다른 함수로부터 값과 같이 전달 받은 함수 내에서 `<<-` 이 사용된다면 여기에서 설명하는 것과 같은 특수한 behaviour 를 보여줍니다.

```
open.account <- function(total) {  
  list(  
    deposit = function(amount) {  
      if(amount <= 0)  
        stop("Deposits must be positive!\n")  
      total <<- total + amount  
      cat(amount, "deposited. Your balance is", total, "\n\n")  
    },  
    withdraw = function(amount) {  
      if(amount > total)
```

```

stop("You don't have that much money!\n")
total <- total - amount
cat(amount, "withdrawn. Your balance is", total, "\n\n")
},
balance = function() {
cat("Your balance is", total, "\n\n")
}
)
}
ross <- open.account(100)
robert <- open.account(200)
ross$withdraw(30)
ross$balance()
robert$balance()
ross$deposit(50)
ross$balance()
ross$withdraw(500)

```

Similarly a function `.Last()`, if defined, is (normally) executed at the very end of the session. An example is given below.

```

> .Last <- function() {
graphics.off() # a small safety measure.
cat(paste(date(),"\nAdios\n")) # Is it time for lunch?

```

## Node 91. Classes, generic functions and object orientation

Object의 class는 어떻게 generic function이라고 알려져 있는 것에 의해 다루어 지는지를 결정 합니다. 반대로, generic function은 그것의 arguments specific에 task나 action을 argument 자체의 class에 실행 합니다. 만약 그 argument에 어떤 class attribute이 부족하거나, 특별히 question안에 있는 generic function을 공급하지 않는 class를 가지고 있다면, 항상 default action이 제공됩니다. 다음의 예제가 이해를 도와 줄 것입니다. Class mechanism은 special purposes에 사용자에게 facility of designing과 writing generic functions을 제공 합니다. 다른 generic functions들에는 objects를 그래픽적으로 디스플레이 하는 `plot()`, various types의 summarizing analyses하는 `summary()`, 그리고 statistical models를 비교하는 `anova()`가 있습니다. Class를 특정한 방법으로 취급하는 generic functions의 숫자는 굉장히 커질 수 있습니다. 예를 들면, 어떤 class "data.frame"의 fashion object안에 적용할 수 있는 generic functions는 다음을 포함합니다.

The class of an object determines how it will be treated by what are known as generic functions. Put the other way round, a generic function performs a task or action on its arguments specific to the class of the argument itself. If the argument lacks any class attribute, or has a class not catered for specifically by the generic function in question, there is always a default action provided. An example makes

things clearer. The class mechanism offers the user the facility of designing and writing generic functions for special purposes. Among the other generic functions are `plot()` for displaying objects graphically, `summary()` for summarizing analyses of various types, and `anova()` for comparing statistical models. The number of generic functions that can treat a class in a specific way can be quite large. For example, the functions that can accommodate in some fashion objects of class "data.frame" include

```
[ [[<- any as.matrix
```

```
[<- mean plot summary
```

현재 완성된 list는 `methods()` function을 이용하여 얻을 수 있습니다:

A currently complete list can be got by using the `methods()` function:

```
> methods(class="data.frame")
```

반대로 generic function의 classes의 숫자는 ??? . 예를 들면 `plot()` function은 default method와 classes "data.frame", "density", "factor", 그리고 더 많은 것의 classes의 object의 변형물들을 가지고 있습니다. 현재 완성된 list는 다시 `methods()` function을 이용하여 얻을 수 있습니다

Conversely the number of classes a generic function can handle can also be quite large. For example the `plot()` function has a default method and variants for objects of classes "data.frame", "density", "factor", and more. A complete list can be got again by using the `methods()` function:

```
> methods(plot)
```

많은 generic functions에 대해 function body는 매우 짧습니다. 예를 들면,

For many generic functions the function body is quite short, for example

```
> coef
```

```
function (object, ...)
```

```
UseMethod("coef")
```

`UseMethod`의 존재는 이것이 generic function임을 말해줍니다. 어떤 methods가 available한지 알아보기 위해서 `methods()`를 사용할 수 있습니다.

The presence of `UseMethod` indicates this is a generic function. To see what methods are available we can use `methods()`

```
> methods(coef)
```

```
[1] coef.aov* coef.Arima* coef.default* coef.listof*
```

```
[5] coef.nls* coef.summary.nls*
```

Non-visible functions는 별표가 되어 있습니다.

이 예제에서는 여섯 개의 methods가 있는데, 이 중 어느 것도 이름을 타이핑하는 방법으로는 찾을 수가 없습니다. 이 method들을 읽기 위해서는

Non-visible functions are asterisked

In this example there are six methods, none of which can be seen by typing its name. We can read these by either of

```
> getAnywhere("coef.aov")
```

'coef.aov'와 일치하는 한 개의 object가 찾아 졌는데 이것은 namespace stats에서 coef의 S3



method로 registered되어있는 다음의 장소에서 발견 되었습니다.

A single object matching 'coef.aov' was found

It was found in the following places

registered S3 method for coef from namespace stats

namespace:stats

with value

function (object, ...)

{

z <- object\$coef

z[!is.na(z)]

}

> getS3method("coef", "aov")

function (object, ...)

{

z <- object\$coef

z[!is.na(z)]

}

독자들은 이것에 대한 기계적 처리에 대해서 R Language Definition 을 살펴보아야 합니다.

### Node 93. Statistical models in R

이번 세션은 당신이 회귀 분석이나 분산 분석 같은 통계적인 방법론을 어느 정도 알고 있다고 가정하고 쓴 것입니다. 후반부에서는 좀 더 욕심을 내서, 여러분이 Generalized Linear Model과 비선형(Nonlinear) 회귀에 대해서도 알고 있을 거라 가정하겠습니다.

통계 모형에 적합 시키기 위한 여러 조건들은 상당히 잘 정의되어 있어서 광범위한 문제들에 적용 가능한 일반적인 모형을 개발하는 것이 가능할 정도입니다.

R은 통계 모형에 접합하는 것을 매우 단순하게 만들어주는 상호 작용 가능한 기능들의 조합을 제공합니다. 우리가 서문에서 살펴 본 것처럼, 기본적인 output은 최소화 되어 있고, 좀 더 상세한 내용이 필요하다면 이를 불러올 수 있는 출력 함수를 사용하면 됩니다.

### Node 94. Defining statistical models; formulae

통계 모형의 대표적인 예인 선형 회귀(Linear Regression) 모형은 독립적(Independent), 동질적이고(homoscedastic) 오차를 가집니다.

$$y_i = \sum_{j=0}^p \beta_j x_{ij} + e_i, i = 1, \dots,$$

n,

이 모형에서 오차를 나타내는  $e_i$  는 독립적이고 동질적으로(IID,

Independent and Identical) 평균이 0 이고 분산이  $\sigma^2$ 인 정규분포( $N(0, \sigma^2)$ )를 따릅니다. 그리고 이것은 행렬(Matrix) 형태로는 이렇게 표현 됩니다.

$y =$

$X \beta + e$

여기에서  $y$ 는 반응변수 vector,  $X$ 는\emph{model

$\text{matrix}$  }\emph{또는}\emph{ design matrix}\emph{라}\emph{ }}\emph{불리  
고}\emph{,

} \emph{각} \emph{ }} \emph{설명} \emph{ }} \emph{변수에} \emph{ }} \emph{해당하  
는} \emph{ }} x\_0, x\_1,

..., x\_p를 열로 갖는 행렬입니다. 많은 경우에 x\_0은 숫자 1로만 구성된 하나의 열이며, 이것  
은 intercept term으로

정의됩니다.

## Node 95. Examples

좀 더 수학적으로 정의를 구체화하기 전에, 도움이 될만한 예제를 몇 개 살펴보겠습니다.

y, x, x0, x1, x2, ...들이 숫자 변수들이라고 가정 합시다. X는 하나의 행렬이고 A, B, C, ...라  
는 요인(Factor)들이 존재한다고도 가정합시다. 아래에서 왼쪽에 제시되는 공식(formulae)은 오  
른쪽에서 기술된 통계 모형을 구체화 합니다.

y x

y 1 + x

두 공식 모두 y를 x에 회귀시킨 단순 회귀모형을 의미합니다. 첫 번째 공식에서는 intercept  
term이 있다는 것이 **함축되어** 있는 것이고, 두 번째 공식에서는 intercept term을 따로 표현해  
준 차이가 있습니다.

y 0 + x

y -1 + x

y x - 1

세 공식은 모두 원점을 지나는 y를 x에 회귀시킨 단순 회귀모형, 즉, **intercept term이 없는 모  
형입니다.**

log(y) x1 + x2

log 변환된 y, log(y)를 x1 과 x2에 적합 시킨 중(multiple)회귀 모형. (intercept term은 **함  
축되어** 있습니다.)

y poly(x,2)

y 1 + x + I(x^2)

모두 x에 대한 2차 다항(polynomial)회귀 모형. 첫 번째 식에서는 직교(orthogonal) 다항을  
사용하고 있으며, 두 번째 식에서는 x의 각 파워(power)들을 직접 basis로 사용해서 표현했습니  
다.

\*역주: 대부분의 통계 서적에서는 두 번째 형태를 사용합니다.

y X + poly(x,2)

행렬 X와 x의 2차 항을 함께 y에 회귀시킨 중회귀 모형.

y A

A로 정의된 하나의 class 값에 대한 y의 분산분석(ANOVA, analysis of variance)모형.

y A + x

A로 정의된 class 값들과 covariate X를 사용한 y의 covariate 모형에 대한 단순  
classification 모형.

y A\*B

y A + B + A:B

y B %in% A

y A/B

A와 B 두 개 요인의 non-additive 모형. 처음 두 방법은 모두 두 요인의 교차항을 사용하여 표현하였지만, 나중의 두 방법에서는 두 개의 classification이 nested 되어 있음을 사용하여 표현한 것입니다. 이들을 좀 더 abstract한 관점에서 보면, 4 가지 방법이 모두 같은 subspace를 갖는 모형을 표현하고 있습니다..

y (A + B + C)^2

y A\*B\*C - A:B:C

모든 main effect와 두 요인간의 interaction만을 포함하는 삼요인 모형.

y A \* x

y A/x

y A/(1 + x) - 1

y를 A의 각 수준들로 이루어진 x에 회귀시킨 단순 선형 모형을 세 가지의 다른 coding 방법으로 나타낸 것. 가장 마지막 방법은 A의 각각의 level에 해당하는 서로 다른 intercept와 slope를 추정하는 모형을 의미합니다.

y A\*B + Error(C)

두 개의 treatments에 해당하는 A와 B 요인과 요인 C에 의해 결정되는 error 부분을 포함하는 실험 모형. 예를 들면, 전체 실험 구성요소(plot)들 중 한 부분은 요인 C에 의해 결정되고 있는 모형입니다.

R에서는 연산자 이 하나의 model formula 를 표현하기 위해 사용됩니다.

일반적인 선형 모형의 경우, 다음과 같이 표현할 수 있습니다.

$\text{response} \sim \text{op}_1 \text{term}_1 \text{op}_2$

$\text{term}_2 \text{op}_3 \text{term}_3 \dots$

이 경우

Response(반응 변수)

반응 변수(들)을 정의하는 하나의 벡터 혹은 행렬 (또는 이를 표현하는 식)

op\_i

+ 또는  $\diamond$  연산자, + 는 해당 변수가 그 모형에 포함되어 있음을  $\diamond$  는 해당 변수가 그 모형에서 빠져 있음을 의미합니다. (모형에 포함된 변수를 + 를 사용해서 따로 표현하는 것은 선택 사항입니다.)

term\_i

이것은 세 가지 방법으로 해석될 수 있는데

1) 하나의 벡터 또는 행렬, 혹은 1

2) 하나의 요인(factor)

3) 여러 개 요인(factor)들로 구성된 model formula. 이 경우 벡터 혹은 행렬 형태로 표현된 요인들이 연산자들에 의해 연결되어 표현됩니다..

어떤 경우에도 각각의 term은 model matrix에서 포함되거나 제외되는 열들을 모은 것으로 정의할 수 있습니다. 1 은 intercept에 해당하는 열을 의미하는 것으로, 특별히 제외시켜야 하는 경우가 아니면, model matrix에 이미 함축적으로 포함된 것입니다.

Formula 연산자들은 Glim 과 Genstat가 만든 프로그램에서 사용된 Wilkinson과 Rogers의 표현 방법과 유사 합니다. R에서의 유일한 큰 차이 점이라면 연산자  $\cdot$ 이 formula 연산자에서는  $\cdot$ 로 표현된다는 점일 겁니다. 이것은  $\cdot$ 가 R에서는 유효한 문자로 사용되기 때문입니다.

다음과 같은 방법으로 formula 연산자들이 사용됩니다. (based on Chambers & Hastie, 1992, p.29):

YM

Y 는 M에의해 모형 적합된다.

M\_1 + M\_2

M\_1 과 M\_2을 포함한다.

M\_1 - M\_2

M\_1은 포함하지만 M\_2는 제외한다.

M\_1 : M\_2

M\_1 과 M\_2의 tensor product. 만약 두 term이 모두 요인이라면, 이들의 subclass들에 해당하는 요인..

M\_1 %in% M\_2

의미는 M\_1 : M\_2 과 같음

M\_1 \* M\_2

M\_1 + M\_2 + M\_1:M\_2.

M\_1 / M\_2

M\_1 + M\_2 %in% M\_1.

M^n

n차 interaction들을 포함한 M안에 든 모든 term들을 의미

I(M)

복잡하게 표현된 M을 따로 정의하는 방법. M 안에 든 모든 연산자들은 원래대로의 산술적 의미를 그대로 갖는다. 그리고 M은 model matrix에 사용된다.

대개 하나의 함수를 포함하는 괄호(( )) 안에서는 모든 연산자들이 원래 그 연산자의 수학적 의미를 그대로 갖게 됩니다. 예를 들어 함수 I()의 경우 다른 여러 개의 산술 연산자로 표현 가능한 model formula들을 포함할 수 있는 단일한 하나의 함수 개체인 것입니다.

특히 주의할 점은, model formula들로 model matrix의 열들을 표기할 때, 그것은 동시에 모형에서의 parameter들을 표기하는 것이기도 합니다. 이것은 nonlinear model과 같이 다른 형태의 방법론에는 해당되지 않는 내용 입니다.

## Node 96. Contrasts

우리는 model formula들을 model matrix, X에서의 각 열들을 사용하여 어떤 식으로 표현할 지를 알 필요가 있습니다. 만약 우리가 연속형 변수들만을 가지고 있다면, 하나의 열만을 가진 X를 사용한 모형과 같은 방법으로 이것을 쉽게 표현할 수 있습니다. (그리고 그 모형에 1로만 구성된 intercept term이 포함된 경우도 같습니다.)

그렇다면 k-level 요인(factor) A의 경우는 어떨까요? 정답은 요인 A가 순서화 되어 있는지 그

령지 않은지에 따라 달라집니다. 순서화 되어있지 않은 요인에 대해서는 2번째부터 k 번째까지의 level을 의미하는 k-1개의 indicator들에 대응되는 k-1 개의 열을 사용할 것입니다. (즉, 이러한 parameterization은 각 수준의 반응변수 값들을 첫 번째 수준에서의 반응변수 값과 비교한다는 의미를 함축합니다.) 순서화된 요인에 대해서는, k-1개의 열들이 상수항을 제외한 1부터 k까지의 직교 다차항들에 해당합니다.

정답이 이미 상당히 복잡해지긴 했지만, 아직 결론에 도달한 것이 아닙니다. 첫째로 intercept term이 하나의 요인 형태로 포함되어 있어, intercept term이 빠진 모형의 경우가 있을 수 있습니다. 이 경우 k개의 열이 사용되어 각각의 열이 모두 각각의 level에 대응됩니다. 두 번째로는, 모든 형태가 contrast를 어떤 형태로 표현하는지에 따라 전체 모형이 완전히 바뀌게 된다는 점입니다. R에서는 초기값(default)으로 contrast가 다음과 같이 표현됩니다.

```
options(contrasts = c("contr.treatment",  
"contr.poly"))
```

이 점을 언급하는 이유는 R과 S가 순서화되지 않은 요인에 대해 서로 다른 초기값을 사용하고 있기 때문입니다. S에서는 Helmert contrasts를 사용하고 있어서, 만약 당신이 S를 사용해서 기술된 책이나 논문의 결과를 R에서 얻기 위해서는 위의 문장을 다음과 같이 표현해야 합니다.

```
options(contrasts = c("contr.helmert",  
"contr.poly"))
```

이렇게 R에서 treatment contrast를 채택한 것은 의도적인 것으로, 초보자들에게는 이 방법이 해석하기 쉬운 것이라 생각되었기 때문입니다.

model에서 함수 contrasts와 C를 사용할 수 있도록 하기 위해, 좀 더 설명이 필요할 것입니다. 우리는 아직 interaction term에 대한 부분을 고려하지 않았는데, 각각의 interaction term은 그에 대응하는 열들의 곱 형태로 나타내어 집니다.

세부 사항들은 복잡해 보이지만, R에서 사용되는 model formula들은 보통 통계 전문가라면 가정한 만한 모형들을 다룰 수 있도록 하면서도 가능한 단순하게 설계된 것입니다. 예를 들어, interaction term은 포함하지만 주 효과(main effects)를 포함하지 않는 모형의 경우 굉장히 특이한 결과를 내는데, 이런 모형에 대해서는 전문가들만 관심을 가질 것입니다.

## Node 97. Linear models

보통의 여러 개 항을 가진 모형을 적합하는 가장 기본적인 방법으로 lm()이 있다. 그리고 이 함수를 사용하기 위한 간단한 방법은 다음과 같습니다:

```
> \textit{fitted.model} <- lm(\textit{formula}, data =  
\textit{data.frame})
```

예를 들면,

```
> fm2 <- lm(y ~ x1 + x2, data =  
production)
```

위의 표현은 y에 대한 x1과 x2의 중회귀 모형에 적합시키는 방법입니다. (이 모형은 함축적으로 intercept term을 포함하고 있습니다.)

데이터 production을 사용한 모형에서 parameter를 정할 때 중요한 점은 (기술적으로는 덜 중요한 부분일 수도 있지만) 모형에 포함될 변수들은 모두 데이터 프레임인 production에 포함되어 있어야 한다는 점입니다. 이러한 원칙은 데이터 프레임 production이 탐색 경로 안에 포함되

어 있거나 그렇지 않은 경우에도 모두 지켜져야 합니다.

**Node 98. Generic functions for extracting model information** (모형에 대한 정보를 보여주기 위한 일반(generic) 함수의 사용)

`lm()` 함수는 적합된 모형에 대한 결과를 보여 줍니다: 기술적으로는 `lm`에 과 관련된 모든 결과들을 죽 나열한 것이지요. 적합된 모형에 대한 정보는 `lm` 이 포함하는 여러 개의 개체(object)들을 사용하는 보다 일반(generic) 함수들을 사용해서 보여주거나, 추출하거나, 그래프로 표현 가능합니다. 이러한 작업을 하기 위한 함수들로는

`add1` `deviance` `formula`

`predict` `step`

`alias` `drop1` `kappa`

`print` `summary`

`anova` `effects`

`labels` `proj` `vcov`

`coef` `family`

`plot`

`residuals`

가장 많이 사용되는 함수들로는 다음과 같은 것들도 있습니다.

`anova(object_1, object_2)`

새로운 모형과 현재의 모형을 비교해서 분산분석(ANOVA) 결과를 출력하는 방법

`coef(object)`

회귀 계수를 출력하는 방법 (행렬 형태)

Long form: `coefficients(object)`.

`deviance(object)`

잔차 제곱의 합, 혹은 특정한 경우 잔차들을 `weighted` 해서 구한 값

`formula(object)`

모형 `formula`를 추출하는 방법

`plot(object)`

**잔차**, 모형에 적합된 값들 그리고 몇몇 diagnostic의 결과들을 보여주는 네 개의 그림을 출력하는 방법

`predict(object, newdata=data.frame)`

이 함수를 사용하기 위해서는 새로 사용되는 데이터 프레임이 기존의 데이터 프레임과 같은 이름을 **가지는** 변수들을 반드시 포함하고 있어야 합니다. 새 데이터 프레임 `data.frame` 안에 **포함된** 변수들을 사용해서 예측된 값들의 벡터나 행렬 값을 생성.

`print(object)`

개체를 간단한 형태로 출력하는 방법. 사실 많은 경우 다른 함수에 포함되어져 가장 많이 사용되고 있는 함수.

`residuals(object)`

잔차 (혹은 그의 행렬)를 출력하는 방법. 필요한 경우, `weighted` 된 잔차 역시 출력 가능

Short form: resid(object).

step(object)

일종의 변수들에 대한 계층을 만들고 이를 바탕으로 각 term들을 첨가하거나 제거함으로써 적절한 모형을 선택하기 위해 사용하는 방법. 가장 작은 AIC(Akaike's An Information Criterion) 값을 갖는 모형을 stepwise 방법으로 찾아내어 출력해준다.

summary(object)

회귀 분석 결과를 간결한 형태로 출력하는 방법

vcov(object)

적합된 모형안에 포함된 주요 parameter들에 대한 variance-covariance(분산-공분산) 행렬을 출력하는 방법

## Node 99. Analysis of variance and model comparison

모형을 적합시키는 데 사용하는 함수의 일종인 aov (formula, data=data.frame)는 lm() 함수가 사용하는 방법과 비슷한 방법을 사용하는 가장 단순한 방법 중 하나 입니다. 또한, 위의 11.3 Generic functions for extracting model information에서 제시된 대부분의 generic 함수들을 사용하는 것이 가능합니다.

매우 중요한 장점의 하나로, aov() 함수에서는 복잡한 형태의 오차 계층을 사용하여 모형을 분석하는 것이 가능하다는 점을 언급하지 않을 수 없는데, 이러한 오차의 계층을 가정하기 때문에 split plot experiment 또는 block간의 정보를 추가로 사용하는 balanced incomplete block design이라는 모형 등을 다룰 수 있습니다.

\textit{}

\textit{모형에 대한}\textit{}

formula}\textit{는 다음과

같습니다}\textit{.}

\textit{response}\textit{}

}\textit{mean.formula}\textit{ +

Error}(\textit{strata.formula})\textit{}

\textit{}

strata.formula }\textit{부분을 통해서 오차의 계층을 표현할 수 있고}\textit{, }\textit{이  
를 통해 여러

계층으로 이루어진}\textit{ experiment}\textit{를 표현할 수 있는 것입니  
다}\textit{. }\textit{가장 단순한

경우는}\textit{ strata.formula}\textit{가 하나의 요인으로 표현되는 것이  
며}\textit{, }\textit{이

경우}\textit{ experiment}\textit{는 두 개의 계층을 가지는 것으로}\textit{, }\textit{그  
계층은 그 요인의

수준들에 대한 수준 내}\textit{(within)}\textit{와 수준 간}\textit{(between)}\textit{비  
교에

해당합니다}\textit{.}

\textit{}



\textit{예를 들면}\textit{, }\textit{모든 주요 변수 요인들을 사용하는}\textit{model formula}\textit{의 하나로 다음과 같은 모형을 생각해 볼 수 있습니다}\textit{:}\textit{}

```
> fm <- aov(yield ~ v + n*p*k + Error(farms/blocks),  
data=farm.data)
```

이 경우, \texttt{v}

\texttt{와}\texttt{ n\*p\*k}\texttt{에 대한 평균을 이용하고 세개의 오차 계층}\texttt{, }\texttt{즉

\texttt{between farms}\texttt{, }\texttt{within farms, between blocks}\texttt{그리고}\texttt{within blocks} 을 이용하는 모형을 기술하는 방법입니다.

### Node 100. ANOVA tables

여러 개의 적합 모형에 대한 분산분석(ANOVA)표 (혹은 표의 집합) 라는 것도 역시 가능합니다. 잔차 제곱의 합은 일련의 순서에 해당하는 term들이 하나씩 모형에 포함됨에 따라 계속 감소하는 것으로 나타납니다. 그러므로 orthogonal experiment에 대해서는 변수가 첨가되는 순서가 그다지 중요하지 않다고 볼 수 있습니다.

또, 복잡한 계층을 가진 실험에 대해서는 반응 변수 값을 먼저 오차의 계층들 중 하나에 project 하고 각 projection에 대한 평균을 이용해 모형을 **적합시키는** 것을, 순차적으로 이를 반복함으로써 분산분석이 가능해 집니다. 좀 더 자세한 세부 사항에 대해서는 Chambers & Hastie (1992)를 참고하시기 바랍니다.

이렇게 default로 지정되어 있는 full ANOVA를 사용하는 대신 좀 더 간단한 대안으로 함수 anova()를 사용해서 두,세개의 관심 모형을 직접 비교하는 방법을 생각해 볼 수 있습니다.

```
> anova(\textit{fitted.model.1}, \textit{fitted.model.2},  
...)
```

이 경우 ANOVA 표는

모형들이 포함된 순서대로 서로 간에 얼마나 차이가 나는지를 보여주게 됩니다. 물론, 많은 경우 비교에 사용된 적합 모형들은 나름대로 어떤 계층적 순서를 갖도록 선택합니다. 이러한 모형의 순서는 default에서와 다른 정보를 얻기 위한 것이 아니라, 오히려 모형을 이해하고 이들을 선택하는 것을 쉽게 하기 위해서 입니다.

### Node 101. Updating fitted models

update()함수는 이전에 정의되었던 적합 모형에서 단 몇 개의 term을 첨가하거나 제거해서 새로운 모형으로 적합시킬 때 매우 유용한 함수입니다. 사용법은 다음과 같습니다.

```
> \textit{new.model} <-  
update(\textit{old.model}, \textit{new.formula})
```

new.formula라는 특정 이름을 가진 새 함수를 정의함에 있어 update를 사용하는 것은 "기존의 old.model에 새로운 모형이 대응된다"는 것을 의미하는 것입니다. 앞서 언급했던 대로, 함수의 이름으로 \texttt{,}\texttt{가 포함될 수 있습니다. 좀 더 구체적인 예를 살펴보면,

```
> fm05 <- lm(y ~ x1 + x2 + x3 + x4 + x5, data =  
production)
```

```
> fm6 <- update(fm05, . . +
x6)
> smf6 <- update(fm6, sqrt(.))
.)
```

fm05는 데이터 프레임 **production** 안에 포함된(혹은 포함되어 있을 것이라 추측되는) 다섯 개의 변수 **x1 x5**에 적합된 중회귀 모형입니다. 여기에 여섯번째 회귀 계수를 첨가해서 새로운 모형 **fm6**을 적합했습니다. 그리고 여기에서 반응변수를 **square root**로 변환시켜 모형을 적합시킨 것이 **smf6**입니다.

특히, **data=** 구문이 처음 모형 적합을 위한 문장에서 사용되면, 이에 해당하는 정보는 이후에 **update()**에 의해 생성되는 일련의 모형 적합에 계속해서 사용됩니다.

부호 **~**는 다른 용도로도 사용될 수 있지만, 이 경우 그 의미는 약간 달라집니다. 예를 들면, 다음과 같은 경우가 있습니다.

```
> fmfull <- lm(y ~ ., data =
production)
```

이 모형은 **y**를 반응 변수로 그리고 그 외 데이터 프레임 **production** 안에 들어 있는 다른 모든 변수들을 회귀계수로 사용하는 적합을 의미합니다.

순차적으로 여러 개의 모형을 살펴보기 위한 함수로는 **add1()**, **drop1()** 그리고 **step()**이 있습니다. 이러한 함수들은 이름의 의미 그대로 사용되지만, 좀 더 자세한 사용법은 온라인 **help**를 참조하기 바랍니다.

## Node 102. Generalized linear models

Generalized linear model은 정규분포를 따르지 않는 반응변수와 모형의 선형성(linearity)를 위한 단순하고 분명한 변수변환을 모두 사용해서 선형 모형(linear model)을 찾는 방법입니다. Generalized linear model은 다음에 제시된 일련의 가정들에 의해 성립됩니다:

- 모형에서는 **y**라는 반응 변수와 반응 변수의 분포에 영향을 주는 **x\_1, x\_2, ...,** 같은 stimulus 변수들이 중요합니다.

- Stimulus 변수들은 오직 하나의 선형 함수를 통해서만 **y**의 분포에 영향을 미치게 됩니다. 이 선형 함수는 linear predictor이며 다음과 같이 표기 됩니다.

$$\eta = \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_p x_p,$$

따라서  $\beta_i$ 가 0인 경우,  $x_i$ 가 **y**의 분포에 미치는 영향력은 없다고 볼 수 있습니다. (if and only if)

- **y**의 분포는 다음과 같은 형태입니다.

$$f_Y(y; \mu, \phi) = \exp((A/\phi) * (y \lambda(\mu) - \gamma(\lambda(\mu)))) + \tau(y, \phi))$$

여기서  $\phi$ 는 (추정 가능한) scale parameter이고, 모든 관측치는 상수(constant)이며, **A**는 사전 가중치(prior weight)를 나타내며 이 가중치는 이미 주어졌다고 가정하지만, 관측치에 따라 다른 값을 가질 수 있습니다. 또한,  $\mu$ 는 **y**의 평균입니다. 따라서, **y**의 분포는 **y** 자신의 평균과 어떤 scale parameter에 의해서 결정되는 것입니다.

- 평균,  $\mu$ 은 linear predictor(선형으로 연결된 설명 변수)에 대한 smooth invertible함수로 표현될 수 있습니다:  $\mu = m(\eta), \eta = m^{-1}(\mu) = \ell(\mu)$

여기서 역(inverse)함수인 `ell()`은 link function 입니다.

이러한 가정들은 실전에서 유용한 다양한 종류의 모형을 모두 포함할 수 있을 만큼 충분히 **느슨한 것이지만**, 동시에 근사적으로나마 추정(estimation)과 추론(inference)을 하기 위한 하나의 통일된 방법을 개발할 수 있을 **정도만큼** 엄격한 것이기도 합니다. 이 방법에 관해서는 McCullagh & Nelder (1989) 이나 Dobson (1990) 과 같은 참고 서적을 통해 좀 더 자세하고 최신의 정보를 얻을 수 있습니다.

#### Node 104. **glm()** function

반응 변수의 분포가 오직 stimulus 변수들에 대한 하나의 함수에 의해서만 영향을 받기 때문에, 이와 같은 메커니즘에 의해 generalized model의 선형 부분이 표기될 수 있습니다. **이럴 경우**, family는 다른 방법으로 표현되어야 합니다.

generalized linear model에 적합시키기 위한 R 함수는 `glm()`이고 다음과 같은 형태로 사용됩니다.

```
> \textit{fitted.model} <- glm(\textit{formula},  
family=\textit{family.generator}, data=\textit{data.frame})
```

여기서 유일하게 새로 등장한 부분은 `family.generator` 이고, 이 부분이 family를 기술하기 위한 도구 입니다. 실제로 모형을 정의하고 추정 수행하기 위한 일련의 함수와 표현들은 이 부분의 함수 이름에 의해 모두 결정되는 셈 입니다. 처음에는 약간 복잡하게 보일 수도 있지만, 굉장히 간단한 방법**입니다**.

family generator를 설정하기 위해 표준적으로 사용되는 이름들의 목록은 이전 Families 부분에 포함된 표의 **Family Name** 항목을 통해 확인하실 수 있습니다. Link의 선택도, family name처럼 함수에 포함된 괄호 안에서 하나의 parameter로 선택 가능합니다. 특히, quasi family의 경우, variance function 역시 이런 식으로 표기 됩니다.

아래의 예제들이 이 방법을 이해하는데 도움이 될 것 입니다.

#### Node 105. **The gaussian family**

gaussian family를 사용하기 위한 방법은 다음과 같습니다.

```
> fm <- glm(y ~ x1 + x2, family = gaussian, data =  
sales)
```

다음 문장으로도 같은 결과를 얻을 수 있습니다.

```
> fm <- lm(y ~ x1+x2, data=sales)
```

하지만 이 경우는 위의 방법보다 **훨씬** 효율이 떨어집니다. 또한, 이 방법은 여러 종류의 link들 중 하나를 선택하는 함으로써 Gaussian family가 된 것이 아니므로 실은 parameter가 없는 모형 선택 방법인 것입니다. Nonstandard link를 사용하는 Gaussian Family를 사용해야 하는 경우라면, 이러한 문제는 quasi family를 사용하여 해결할 수 있으며, 조금 **뒤**에 이에 대해 살펴볼도록 하겠습니다.

#### Node 106. **The binomial family**

Silvey (1970) 등장하는 간단한 예제를 하나 살펴볼 것입니다. Kalythos라는 한 예게해의 섬에서는 남성들이 선형성 안질환을 겪고 있습니다. 그 안질환은 나이가 들어감에 따라 좀 더 진행됩니다.

니다. 섬에 거주하는 다양한 연령대의 남성들이 실명(blindness)여부를 검사하기 위해 선택되었고 그 실험결과는 다음과 같이 기록되었습니다:

Age:

20

35

45

55

70

No. tested:

50

50

50

50

50

No. blind:

6

17

26

37

44

우리는 이 데이터에 대해 logistic과 probit 두 가지 모형을 적합시키고 각 모형에서의 LD50를 추정하는 문제에 관심이 있습니다. LD50 추정은 남성 거주자의 실명 확률이 50%가 되는 나이를 찾는 것입니다.

y는 연령 x에서의 실명한 사람의 수이고 n이 검사를 한 사람의 수라면, probit과 logit 모형 모두 다  $y \sim B(n, F(\beta_0 + \beta_1 x))$ 의 형태를 갖습니다. 단, probit의 경우  $F(z) = \Phi(z)$ 는 표준 정규 분포 함수이고, logit의 경우 (default로 사용되는 모형)에서는  $F(z) = e^z / (1 + e^z)$ 입니다. 두 경우 모두, LD50는  $LD50 = -\beta_0 / \beta_1$ 이며, 이것은 분포함수  $F(z)$ 의 값을 0.5로 만들어주는 z 값에 해당합니다.

첫 번째 단계는 모형에서 다루게 될 data를 data frame으로 만드는 것입니다.

```
> kalythos <- data.frame(x = c(20,35,45,55,70), n =
```

```
rep(50,5),
```

```
y =
```

```
c(6,17,26,37,44))
```

glm()를 사용해서 binomial model을 적합 시키기 위해서는 반응 변수가 다음 세가지 중의 하나에 해당될 것 입니다:

- 반응변수가 벡터인 경우, 이 family의 경우 데이터가 binary 데이터라는 점을 이미 가정하고 있으므로 이 벡터는 0과 1의 값 만을 포함합니다.
- 만약 반응변수가 2열의 행렬 형태로 되어 있다면, 첫 번째 열은 각 시행에서 성공한 숫자를, 두 번째 열은 실패한 숫자를 의미 합니다.
- 만약 반응변수를 하나의 요인(factor)으로 되어 있다면, 첫 번째 수준은 실패(0)로 그 외

나머지 모든 수준은 성공(1)로 간주해야 합니다.

이번 예제에서는 이러한 관습적 데이터 형태 중에서 두 번째 형태를 사용할 것이므로, 원래의 data frame에 다음과 같은 행렬을 첨가해야 합니다:

```
> kalythos$Ymat <- cbind(kalythos$y, kalythos$n -  
kalythos$y)
```

모형을 적합시키는 방법은 다음과 같습니다.

```
> fmp <- glm(Ymat ~ x, family  
= binomial(link=probit), data = kalythos)  
>
```

```
fml <- glm(Ymat ~ x, family = binomial, data =  
kalythos)
```

Logit link는 default 이므로 두 번째 문장에서는 link를 지정(link를 지정 생략?) 생략할 수 있습니다. 각 경우에 해당하는 적합 결과를 보기 위해 다음 문장을 사용할 수 있습니다.

```
> summary(fmp)  
>
```

```
summary(fml)
```

두 모형 다 적합 결과가 좋습니다. (실제로 이 정도로 좋은 힘듭니다.)

LD50를 추정하기 위한 함수는 다음과 같이 간단하게 정의 됩니다:

```
> ld50 <- function(b)  
-b[1]/b[2]  
> ldp <- ld50(coef(fmp)); ldl <-  
ld50(coef(fml)); c(ldp, ldl)
```

이 데이터를 사용해서 직접 구한 LD50의 추정치는 각각 43.663과 43.601입니다.

## Node 107. Poisson models

Poisson family의 경우 default link는 log이고, 실제로 이 family는 주로 빈도(frequency) 데이터를 Poisson log-linear로 적합시키는데 많이 사용됩니다. 원래, 이러한 빈도 데이터의 실제 분포는 많은 경우 multinomial 분포를 따르는데도 말이지요. 이 문제는 굉장히 방대하고도 중요한 주제 중 하나이므로 여기서는 다루지 않도록 하겠습니다. 이 모형을 사용하는 것은 non-gaussian generalized model의 사용의 거의 대부분을 차지할 정도로 중요합니다.

적지 않은 경우, 실전에서는 Poisson data를 log나 square-root로 변환시킨 후 Gaussian data처럼 분석하기도 했었습니다. 하지만, Poisson generalized linear이 후에 이러한 방법을 대체하게 되었고, 아래와 같은 방법으로 사용 가능 합니다:

```
> fmod <- glm(y ~ A + B + x, family = poisson(link=sqrt), data =  
worm.counts)
```

## Node 108. Quasi-likelihood models

모든 family들의 대해, 반응변수의 variance는 mean에 의해 결정되고 scale parameter는 multiplier의 형태라는 공통점을 발견할 수 있습니다. Variance의 mean에 의한 영향력의 형태는 반응변수의 분포의 특성이기도 합니다: 예를 들면 반응변수가 Poisson 분포인 경우,  $\text{Var}(y) =$

mu.

quasi-likelihood를 이용한 추정과 추론은 정확한 반응변수의 분포를 찾아내는 것이 아니라, 오히려 link function과 variance function이 어떤 식으로 mean과 관련이 있는지와 관련이 있습니다. Quasi-likelihood를 이용한 추정은 Gaussian distribution을 사용할 경우와 같은 방법들을 사용하기 때문에, 이 family는 사실 non-standard link function, 또는 이와 동일하게 variance function을 사용한 Gaussian 모형 적합을 가능하게 해주었습니다.

예를 들어, 비선형 회귀인  $y = \theta_1 z_1 / (z_2 - \theta_2) + e$ 에 대한 적합을 고려한다고 합시다. 이 모형은  $y = 1 / (\beta_1 x_1 + \beta_2 x_2) + e$ 의 형태로도 표현하는 것이 가능하며, 이때  $x_1 = z_2/z_1$ ,  $x_2 = -1/z_1$ ,  $\beta_1 = 1/\theta_1$ , 그리고  $\beta_2 = \theta_2/\theta_1$  입니다. 모형에 대응하는 적당한 데이터 프레임이 존재한다고 가정하면, 이 비선형 회귀는 다음과 같이 적합될 수 있습니다.

```
> nlfit <- glm(y ~ x1 + x2 -  
1,  
family = quasi(link=inverse,  
variance=constant),  
data =  
biochem)
```

좀 더 자세한 설명이 필요하다면, 매뉴얼과 help를 참조하시기 바랍니다.

#### Node 109. Nonlinear least squares and maximum likelihood models

비선형 모형 중 어떤 특별한 형태에 해당하는 경우, Generalized Linear Models (glm())을 사용할 수 있습니다. 그러나 이 경우에도 대부분 비선형 최적화 방법 중의 하나인 비선형 곡선 적합이라는 관점으로 문제에 접근하고 있는 것입니다. R에서는 비선형 최적화 방법을 위해 optim(), nlm()(R 2.2.0 부터 사용가능) 과 S-Plus에서 ms() 과 nlminb()이 제공하던 것과 같은 (혹은 더 발전된) 기능들을 제공하는 nlminb()이 사용되고 있습니다. 이 방법은 몇몇 적합결여 (lack-of-fit) 지표(index)들을 이용하여 이들을 최소화하는 parameter 값을 찾는 것인데, 이러한 R 함수들은 여러 개의 parameter 값들에 대해 이 방법을 반복적으로 적용해서 모형을 적합시킵니다. 하지만, 선형 회귀 방법과는 다르게 이러한 procedure가 만족스러운 수준의 추정치들로 converge할 것이라는 점은 보장할 수 없습니다. 그래서 여기에 사용되는 모든 방법들을 사용할 때는 어떤 parameter들을 대상으로 최적화를 할 것인지 그리고 convergence가 초기값의 선택에 따라 크게 달라질 수도 있다는 점을 미리 고려해야 합니다.

#### Node 110. Least squares

비선형 모형에 적합시키는 한 가지 방법은 제공한 오차(error) 또는 잔차(residual)들의 합을 최소화시키는 것입니다. 이러한 방법은 관측된 오차들이 정규 분포와 상당히 유사할 때 사용해야 합니다.

Bates & Watts (1988), page 51에 나오는 예제를 하나 살펴 보겠습니다. 데이터가 아래와 같이 주어지고,

```
> x <- c(0.02, 0.02, 0.06, 0.06, 0.11, 0.11, 0.22, 0.22, 0.56,  
0.56, 1.10, 1.10)
```



```
> y <- c(76, 47, 97, 107,
123, 139, 159, 152, 191, 201, 207, 200)
```

잔차 제곱합을 최소화하는 모형 적합 기준을 사용하기 위해 다음과 같은 함수를 정의 합니다.

```
> fn <- function(p) sum((y - (p[1] * x)/(p[2] +
x))^2)
```

모형 적합을 위해서는 parameter들에 대한 초기 추정값이 필요합니다. 적절한 초기값을 찾는 방법 중 하나는 데이터를 plot하여 몇 개의 parameter 값들을 추측해낸 후, 이러한 추정값들을 사용했을 때의 모형 curve를 원래 data plot과 겹쳐 놓고 비교하는 것입니다.

```
> plot(x, y)
> xfit <- seq(.02, 1.1, .05)
> yfit <- 200 *
xfit/(0.1 + xfit)
> lines(spline(xfit,
yfit))
```

물론 더 나은 값이 존재할 수도 있겠지만, 일단 200과 0.1을 초기값으로 사용하는 것도 나쁘지 않을 것 같습니다. 다음으로 모형을 적합시키면:

```
> out <- nlm(fn, p = c(200, 0.1), hessian =
TRUE)
```

모형 적합 후, out\$minimum은 SSE를, out\$estimate은 parameter들의 최소제곱(least squares)추정치를 출력합니다. 추정치의 근사 오차(SE, standard errors)를 구하기 위한 방법은 다음과 같습니다:

```
> sqrt(diag(2*out$minimum/(length(y) - 2) *
solve(out$hessian)))
```

위 명령분에 나타난 숫자 2는 parameter의 수를 의미 합니다. 95% 신뢰구간은 parameter추정치에 1.96 SE를 더하고 빼서 구합니다. 최소제곱 추정으로 구한 모형을 새로운 plot으로 나타내 보겠습니다:

```
> plot(x,
y)
> xfit <- seq(.02, 1.1,
.05)
> yfit <- 212.68384222 * xfit/(0.06412146
+ xfit)
> lines(spline(xfit,
yfit))
```

기본 패키지인 `stats`은 비선형 모형을 최소제곱 방법으로 추정하기 위한 보다 다양한 기능들을 제공합니다. 바로 위에서 사용한 모형 적합 방법은 Michaelis-Menten 방법에 의한 것으로, 다음과 같은 방법으로 사용 가능합니다.

```
\textbf{}
> df
<- data.frame(x=x, y=y)
```

```

> fit <- nls(y
SSmicmen(x, Vm, K), df)
>
fit
Nonlinear regression
model
model: y SSmicmen(x, Vm,
K)
data:
df
Vm
K
212.68370711
0.06412123
residual sum-of-squares:
1195.449
>
summary(fit)
Formula: y SSmicmen(x, Vm,
K)
Parameters:
Estimate Std. Error t value Pr(>|t|)
Vm 2.127e+02 6.947e+00 30.615 3.24e-11
K
6.412e-02 8.281e-03 7.743 1.57e-05
Residual standard error: 10.93 on 10 degrees
of freedom
Correlation of Parameter
Estimates:
Vm
K
0.7651

```

### Node 111. **Maximum likelihood**

최대 우도방법(Maximum likelihood)은 오차들이 정규분포를 따르지 않는 경우에도 사용 가능한 비선형 모형 적합 방법 중의 하나입니다. 이 방법은 log likelihood를 최대화하거나, 이와 동등하게 Negative(음의 부호를 첨가한) log likelihood를 최소화 시켜주는 parameter 값들을 찾아내는 방법입니다. Dobson (1990), pp. 108-111 에 나오는 예제를 하나 살펴보도록 하겠습니다. 이 예제에서는 dose-response 데이터를 logistic 모형에 적합시키고 있습니다. 물론, 이 경우 glm() 함수를 사용하는 것도 가능합니다.



데이터는 다음과 같습니다:

```
> x <- c(1.6907, 1.7242, 1.7552, 1.7842,
1.8113,
1.8369, 1.8610,
1.8839)
> y <- c( 6, 13, 18, 28, 52, 53, 61,
60)
> n <- c(59, 60, 62, 56, 63, 59, 62,
60)
\texttt{}
\texttt{
{ Negative log-likelihood }\texttt{ 값을 최소화하기
위해서는 }\texttt{:}
> fn <-
function(p)
sum( - (y*(p[1]+p[2]*x) -
n*log(1+exp(p[1]+p[2]*x))
+
log(choose(n, y)) ))
```

그럴 듯한 초기값을 선택한 후 모형을 적합 시키겠습니다:

```
> out <- nlm(fn, p = c(-50,20), hessian =
TRUE)
```

모형 적합된 후, `out$minimum`은 negative log-likelihood에 값을 `out$estimate`은 maximum likelihood(최대 우도) 방법을 사용한 parameter의 추정치들을 출력하게 됩니다. 추정치에 대한 SE의 근사값을 구하는 방법은 다음과 같습니다.

```
>
sqrt(diag(solve(out$hessian)))
95% 신뢰구간은 parameter estimate에 1.96*SE를 더한 값과 뺀 값 입니다.
```

## Node 112. Some non-standard models

이번 강의 마지막 순서로, 특별한 형태의 회귀 및 데이터 분석을 하기 위해 R에서 어떤 기능을 추가적으로 제공하고 있는지 간단히 언급하도록 하겠습니다.

- Mixed models.

**패키지** nlme에 든 lme() 와 nlme() 함수를 사용해서 선형과 비선형 혼합 효과(mixed effect) 모형을 다룰 수 있다. **Mixed effect** 모형이란 선형과 비선형 회귀모형을 의미하는 것으로 coefficient들 중의 일부가 random effect에 해당됩니다. 이러한 함수들은 모형을 기술하기 위해 많은 수의 formula들을 이용합니다.

- Local approximating regressions.

함수 loess()는 일부 계수에만 weight를 주는 가중회귀(weighted regression) 모형을 사용하여 비모수(nonparametric) 회귀를 적합시킵니다. 이러한 회귀 모형은 지지분한 형태의 데이터의

추세를 파악하거나 굉장히 큰 크기의 데이터 셋에 대한 간단한 이해를 위해 데이터 차원을 축소시키는 데 사용될 수 있습니다.

함수 `loess()`는 표준 패키지인 `stats`에 포함되어 있으며, `projection`을 위한 코드를 함께 사용하면 회귀 분석을 할 수도 있습니다.

- Robust regression.

데이터 안에 포함되어 있는 몇 개의 이상치(extreme value)들로 인한 영향력을 줄여 안정적인 회귀 모형에 적합시키기 위한 함수들은 여러 개 존재합니다. 많이 사용되는 패키지인 MASS에서는 함수 `lqs()`를 제공하는데 이 함수는 안정적인 모형적합을 위한 최고의 방법이라 할 수 있습니다. 조금 덜 안정적이긴 하지만 좀 더 효율적인 방법으로는 MASS 패키지 안에 든 `rlm` 함수를 사용하는 것 등이 가능합니다.

- Additive models.

이 방법은 이미 정해져 있는 변수들을 사용하여 회귀 모형을 만드는 경우에 사용되는 것으로, 변수들에 smooth additive 함수들을 적용합니다. 각각의 변수에 하나의 smooth additive 함수를 대응시켜 사용하는 것이 일반적이는데, R에서는 사용자 개발 패키지인 `acepack`에서의 `avas()`와 `ace()` 함수, 그리고 `mda`에서의 `bruto` 와 `mars`가 이러한 기능을 제공하는 예라 할 수 있습니다. 이를 확장한 것으로 Generalized Additive Model이라는 것이 있으며, 이 방법은 `gam`과 `mgcv` 같은 사용자 개발 패키지에 의해 구현 가능합니다.

- Tree-based models.

예측이나 모형 해석을 위해서는 global 선형 모형을 찾는 대신, tree 형태의 모형에서는 데이터를 이미 결정된 변수들의 임계점(critical point)들에서 데이터를 양갈래로 나누어 궁극적으로 여러 개의 그룹으로 구성된 모형을 찾아냅니다. 이러한 최종 그룹들은 가능한 그룹 내에서는 서로 동질적이며, 그룹 간에는 서로 다른 성질을 갖는데, 이러한 모형은 종종 다른 통계 방법으로는 볼 수 없는 데이터에 대한 새로운 관점을 제시한다.

이러한 tree 모형은 또한 보통의 선형 모형의 형태로 표현될 수 있습니다. 이러한 tree 모형을 적합시키는 함수는 `tree()` 입니다. 하지만 `plot()`이나 `text()` 처럼 좀 더 일반적인 작업에 사용되는 `함들일`이 tree 형태의 모형에 적합된 결과를 시각적으로 표현하기 위해 효과적으로 사용되기도 합니다.

R에서는 tree 모형을 `rpart`나 `tree` 같은 사용자 개발 package들을 이용하여 구현할 수 있습니다.

## Node 114. Graphical procedures

R의 그래픽 기능은 R 구성 환경 중에서도 굉장히 중요하고 많은 역할을 담당하고 있습니다. 이는 이러한 기능들을 통해 여러 종류의 통계적인 그래프를 그릴 수 있을 뿐 아니라 완전히 새로운 종류의 그래프를 만들어내는 것도 가능합니다.

그래픽 기능들은 interactive나 batch 모드에서 모두 사용될 수 있지만, 대부분의 경우, interactive 모드에서 사용하는 것이 더욱 효과적입니다. 특히, R에서는 프로그램이 시작될 때 interactive 그래픽을 보여주는 특정 graphic window를 열어주는 device driver가 초기화되므로 이를 이용하기가 더 쉽습니다. 이러한 초기화는 자동적으로 이루어지지만, Unix 환경에서는 `X11()`, Windows 환경에서는 `windows()` 그리고 Mac OS X에서는 `quartz()` 가 device driver를 초기화시키기 위해 사용될 수 있음을 알아두는 편이 유용할 것 입니다.

역주: interactive mode ♦ 사용자가 새로운 명령을 입력할 때마다 거기에 대응되는 graphic 상의 변화를 확인할 수 있는 방법

batch mode ♦ 일련의 그래픽 함수들이 별도의 지정 없이도 자동 실행되어 graphic을 실행하는 방법

일단 device driver가 작동되기 시작하면, R plot명령문들은 다양한 종류의 그래픽을 표현하거나 또는 완전히 새로운 형태의 표현을 위해 사용 될 수 있습니다.

Plot를 그리는 명령문은 다음과 같이 세가지로 구분됩니다:

♦ High-level 함수는 정해진 그래픽 device 내에서 축, 레이블, 제목 등을 변경해서 새로운 plot을 생성 합니다.

♦ Low-level 함수는 이미 존재하는 plot 위해 추가로 점, 선, 레이블 등을 첨가하여 좀 더 많은 정보를 표현할 수 있도록 합니다.

♦ Interactive 함수들은 이미 존재하는 plot에서 마우스와 같은 pointing device를 이용하여 interactive하게 정보를 추가하거나 제거하는 기능을 합니다.

또, R에서는 일련의 graphical parameter들을 조작하여 당신이 원하는 대로 plot을 쉽게 변경할 수 있는 기능을 제공합니다.

이 매뉴얼에서는 『base』 그래픽에 해당하는 내용만을 대상으로 합니다. base와는 별개로 팩키지 grid 안에 별개의 그래픽 sub시스템이 따로 존재하는데, 이것은 좀 더 유용하긴 하지만 사용하기가 더 어렵습니다. Grid 를 바탕으로 만들어진 팩키지 lattice도 추천할 만한데, 이 팩키지는 S의 Trellis 시스템에서 제공하는 것과 비슷한 multi-panel 플롯을 그릴 수 있는 기능들을 제공합니다.

## Node 115. High-level plotting commands

High-level 플롯 함수들은 데이터를 인수(argument) 형태로 만들어 이 함수를 이용하여 복잡한 형태의 plot을 그리기 위해 고안되었습니다. (특별히 따로 지정하지 않으면) 적절한 위치에 축, 레이블 그리고 제목 등이 자동적으로 생성됩니다. High-level 플롯 명령문은 항상 새로운 플롯을 시작하기 때문에 필요한 경우에는 현재 플롯을 자동적으로 지워버리기도 합니다.

## Node 116. The plot() function

R의 plot함수들 중에서 plot()은 가장 많이 사용되는 것 중의 하나일 것입니다. 이것은 일종의 일반(generic) 함수로, plot의 형태는 입력된 첫 번째 인수(argument)의 종류(type)나 class에 의해 결정됩니다.

plot(x, y)

plot(xy)

x와 y가 둘 다 벡터이면, 이 문장은 x에 대한 y의 산점도(scatterplot)을 그립니다. 두 번째 문장 역시 이것과 같이 작동하지만, xy는 x와 y를 둘 다 포함하는 하나의 열(list)이나 2열 행렬형태의 인수여야 합니다.

plot(x)

x가 하나의 시계열인 경우, 이 문장으로 시계열 플롯을 그립니다. x가 하나의 숫자 벡터인 경우, 이 문장으로 각 값들의 인덱스에 대한 해당 벡터 값의 플롯을 그립니다. x가 복소수(complex) 벡터인 경우라면, 이 문장으로 각 벡터의 실수 부분에 대한 허수(imaginary) 부분의 플롯이 나옵니다.

다.

```
plot(f)
```

```
plot(f, y)
```

f가 하나의 요인(factor)이고, y는 하나의 숫자 벡터여야 합니다. 첫 번째 문장은 f에 대한 막대 그래프를, 두 번째 문장은 각 f의 수준에서의 y의 boxplot들을 생성합니다.

```
plot(df)
```

```
plot( expr)
```

```
plot(y expr)
```

위 문장을 사용하기 위해서, df는 하나의 데이터 프레임이어야 하고, y는 어떤 형식이든 상관 없지만, expr 는 여러 개의 객체(object)들이 []+[]에 의해 연결된 형태로 표현되어야 합니다. (e.g., a + b + c) 처음 두 문장은 일종의 변수들에 대한 분포 플롯을 그립니다. (역주 ♦ 각 변수에 대한 다른 변수의 분포를 보여주는 플롯, 일종의 pairwise scatterplots이 출력됨) 이때 plot 함수는 데이터 프레임 안에 포함된 모든 변수들을 대상으로 하거나 (첫 번째 형태), expr 안에서 사용된 몇 개의 변수들만을 대상으로 합니다(두번째 형태). 세 번째 문장의 경우, expr에서 사용된 모든 변수들에 대한 y의 값의 플롯들을 각 변수 별로 출력합니다.

### Node 117. Displaying multivariate data

R은 다변량(multivariate) 데이터를 표현할 수 있는 매우 유용한 두 가지 함수를 제공합니다. X가 숫자 행렬이거나 데이터 프레임인 경우, 그 명령문은

```
> pairs(X)
```

이 문장은 X에 포함되어 있는 열들에 해당하는 변수들을 대상으로 한 행렬의 pairwise 산점도(scatterplot)을 출력합니다. 이것은 X의 한 열(변수)에 대해 X의 다른 모든 열들에 대한 플롯을 그리므로 그 결과 총  $n*(n-1)$ 개의 플롯이 각 열이나 각 행 별로는 일정한 plot scale(단위, 축)을 가진 행렬의 형태로 출력됩니다.

세네 개의 변수들로 작업하는 경우, coplot이 좀 더 효과적인 방법이 될 수 있을 것입니다. a와 b가 숫자 벡터이고 동시에 c가 숫자 벡터이거나 요인(factor)인 경우 (물론 셋 다 같은 길이를 가져야 합니다.), 다음과 같은 명령문이 사용됩니다.

```
> coplot(a b | c)
```

이 문장은 c의 각 값 별로 b에 대한 a의 산점도들을 그립니다. 만약 c가 하나의 요인이라면, 이것은 단순히 c의 수준에 대해 b에 대해 a가 그려진 것을 의미한다. c가 숫자 변수 이면, c의 값들은 몇 개의 조건부 구간(conditioning interval)으로 나뉘지고 c의 구간 별로 대응되는 b에 대한 a의 각 구간을 그립니다. 구간의 수와 위치는 coplot()에서 given.values=선언에 의해 조절할 수 있으며 함수 co.intervals()로 구간 선택을 조절할 수 있습니다. 또한 두 개의 변수를 조건부로 하여 다음과 같은 명령문을 만드는 것도 가능합니다.

```
> coplot(a b | c + d)
```

c와 d에 대한 모든 joint conditioning interval에서 b에 대한 a의 산점도를 그립니다.

coplot()과 pairs()함수는 모두 panel=선언을 사용할 수 있는데, 이는 각 패널을 표현하는 플롯의 종류를 지정하는데 사용됩니다. 디폴트로는 산점도를 그리는 points()가 지정되어 있는데 벡터 x와 y를 사용하는 다른 종류의 low-level그래픽 함수를 panel=에 지정하면 어떤 종류의 플롯이라도 표현 가능합니다. 예를 들면 coplot에서 사용할 수 있는 panel 함수로는

panel.smooth()가 있습니다.

### Node 118. Display graphics

다른 high-level 그래픽 함수들로 다른 여러 종류의 플롯들을 그릴 수 있습니다. 몇 가지 예로 다음과 같은 것들이 있습니다:

```
qqnorm(x)
qqline(x)
qqplot(x, y)
```

이들은 분포-비교에 사용되는 플롯들입니다. 처음 명령문은 expected Normal order score에 대한 숫자 벡터인 x를 그리고(normal score 플롯이라고도 한다.), 두 번째 명령문에서 분포와 data에 대한 quartile들을 통과하는 직선을 그립니다. 세 번째 문장에서는 두 변수의 각각의 분포를 비교하여 y의 quantile에 대한 x의 quantile을 그립니다.

```
hist(x)
hist(x, nclass=n)
hist(x, breaks=b, ...)
```

숫자 벡터인 x에 대한 히스토그램을 그립니다. 범주(class)의 숫자는 대체로 적절하게 선택되지 만, nclass= 선언을 통해 범주의 숫자를 선택할 수도 있습니다. 또 다른 방법으로는, breaks= 라는 선언을 통해서 정확하게 breakpoint를 결정할 수도 있습니다. probability=TRUE 선언이 포함된 경우에는, 막대들은 전체 개수에 대한 상대 도수가 아니라 막대 넓이에 대한 상대 도수를 의미하게 됩니다.

```
dotchart(x, ...)
```

x에 포함된 데이터의 dotchart를 그립니다. dotchart에는 y-축은 x에 포함된 데이터에 의해 이름이 붙고, x-축은 x의 값을 그대로 사용합니다. 예를 들어, 이 방법으로는 정해진 특정 범위 내에 만 포함된 데이터 값을 이용해서 모든 데이터 값들을 쉽게 표현할 수 있습니다.

```
image(x, y, z, ...)
contour(x, y, z, ...)
persp(x, y, z, ...)
```

세 개 변수에 대한 플롯을 그립니다. 첫 번째 명령문에서는 서로 다른 z의 값을 나타내는 여러 가지 색으로 x와 y에 값들을 표현한 직사각형의 틀(grid)을 그립니다. 두 번째 명령문은 같은 z 값을 연결한 등고선(contour line)을 이용한 contour plot을 그리며, 세 번째 명령문은 3D 표면으로 나타내는 조감도(persp plot)를 그립니다.

### Node 119. Arguments to high-level plotting functions

high-level 그래픽 함수에서 사용할 수 있는 선언들은 다음과 같은 것들이 있습니다:

```
add=TRUE
```

해당 함수가 low-level 그래픽 함수처럼 사용되도록 해서, 현재의 플롯 위에 해당 플롯이 덧 그려지도록 합니다. (제한된 함수만 사용 가능)

```
axes=FALSE
```

축의 생성을 제한합니다-axis() 함수를 사용해서 사용자 정의로 축을 표현하고 싶을 때 사용할 수 있습니다. 디폴트로 axes=TRUE 되어 있고, 축을 포함할 것을 의미합니다.

```
log="x"  
log="y"  
log="xy"
```

x, y 또는 두 축 모두를 로그화 합니다. 이 함수는 많은 종류의 플롯에서 작동하지만, 작동되지 않는 경우도 있습니다.

```
type=
```

플롯의 종류를 지정 합니다:

```
type="p"
```

각각 포인트를 점으로 표현 합니다. (디폴트)

```
type="l"
```

선으로 표현 합니다.

```
type="b"
```

점으로 표현하고 선으로 연결 합니다. (두가지 모두)

```
type="o"
```

선 위로 포인트들을 겹쳐지게 합니다.

```
type="h"
```

0 라인부터 각 점까지의 거리를 세로선으로 합니다. (high-density)

```
type="s"
```

```
type="S"
```

Step-function을 합니다. 첫 번째 형태에서는 함수의 끊어진 부분의 윗부분이 점으로 나타나고, 두 번째 형태에서는 아래 부분이 점으로 나타납니다.

```
type="n"
```

플롯을 그리지 않습니다. 그렇지만 (디폴트에 의해) 축은 여전히 나타나며 데이터에 대한 좌표 (coordinate system)는 세워져 있는 상태입니다. Subsequence에 대해 low-level 그래픽 함수를 적용한 플롯을 만들어 내기 위해 사용할 수 있습니다.

```
xlab=string
```

```
ylab=string
```

x와 y축에 대한 축의 이름을 지정합니다. 디폴트로 지정된 축 이름을 다른 것으로 바꾸기 위해 사용되는데, 디폴트 축 이름으로는 주로 해당 high-level 그래픽 함수를 불러내기 위해 사용되었던 변수의 이름들이 나타납니다.

```
main=string
```

그림에 대한 제목, 플롯 위 부분에 큰 폰트를 사용해서 표시 됩니다.

```
sub=string
```

작은(sub)-제목, x축 바로 아래에 좀 더 작은 폰트로 나타냅니다.

## Node 120. Low-level plotting commands

high-level 그래픽 함수로는 정확히 원하는 형태의 플롯을 그리는 것이 불가능한 경우가 있습니다. 이런 경우, low-level 명령문으로 현재의 플롯에 (포인트, 선 또는 문자와 같은) 추가적인 정보를 표현하는 것이 가능합니다.

유용하게 사용될 수 있는 low-level 그래픽 함수로는 다음과 같은 것들이 있습니다:

```
points(x, y)
```



`lines(x, y)`

현재의 함수에 포인트나 이를 연결한 선을 첨가합니다. `plot()`와 이에 포함된 `type="n"` 선언이 이러한 함수들과 비슷하게 사용될 수 있습니다. (디폴트인 `type="p"`는 `points()`과 `"l"`은 `lines()`과 같이 사용될 수 있습니다.)

`text(x, y, labels, ...)`

`x`와`y` 값으로 표현된 점들에 문자를 첨가합니다. 보통 정수나 문자의 벡터 형태로 레이블이 주어지며, 이 경우 특정 `i`에 대해 `(x[i], y[i])`위치에 `labels[i]` 값이 표시됩니다. 디폴트로 `1:length(x)`이 지정되어 있습니다.

주의 : 이 함수는 다음과 같은 순서로 사용되는 경우가 많습니다.

```
> plot(x, y, type="n"); text(x, y, names)
```

그래픽 parameter인 `type="n"`이 포인트가 표현되는 것을 제한하고 있지만, 축은 이미 세워져 있고, `text()` 함수는 각 포인트들에 대한 문자 vector 형태의 이름이 표현되도록 합니다.

`abline(a, b)`

`abline(h=y)`

`abline(v=x)`

`abline(lm.obj)`

기울기 `b`이고 `y`절편 `a`인 선을 현재의 플롯에 첨가합니다. `h=y` 는 `y`좌표값을 높이로 하는 가로선이 이 그래프를 가로질러 그리며, 마찬가지로 `v=x` 은 `x` 좌표값을 시작점으로 하는 세로선을 그립니다. 또, `lm.obj`는 (모형적합 함수의 결과로 얻어진) 길이 2의 coefficient 성분들을 표시하는데, 이 경우에는 순서대로 `y`절편과 기울기 값에 해당합니다.

`polygon(x, y, ...)`

`(x,y)`에 의해 정의된 꼭지점들을 순서대로 연결해서 만든 다각형을 그립니다. 해당 그래픽 장치가 기능을 지원하면, (옵션을 사용해서) 그려진 다각형의 내부에 음영을 주거나 색을 채우는 것이 가능합니다.

`legend(x, y, legend, ...)`

특정한 위치에 현재 플롯의 범례를 첨가합니다. 플롯에 사용된 문자나 선의 종류 혹은 색 등이 범례에 포함된 문자 레이블을 통해 구분 가능해집니다. 다음과 같이 플롯 구성 단위가 가질 수 있는 값들 중 최소한 한 개 이상의 `v` (범례와 같은 길이를 가진 벡터를 사용하여) 선언이 필요합니다:

`legend( , fill=v)`

상자에 칠해진 색상들

`legend( , col=v)`

포인트나 선에 사용될 색상들

`legend( , lty=v)`

선의 종류Line styles

`legend( , lwd=v)`

선의 굵기

`legend( , pch=v)`

문자표시를 포함할 것 (문자 벡터 형태)

`title(main, sub)`

현재 플롯의 윗부분에 큰 글씨로 주요(main) 제목을 첨가하고 (선택에 의해) 좀 더 작은 크기의 글씨로 부(sub)제목은 아래 부분에 표시한다.

`axis(side, ...)`

현재 플롯에 첫 번째 인수(1부터 4까지의 숫자로 나타나며, 바닥에서부터 시계 방향으로 해당 사면을 의미함)에 의해 주어진 면에 새로운 축을 첨가합니다. 다른 인수들로 플롯 안에 포함되거나 플롯 옆에 나타나는 축을 조절할 수도 있고, 위치나 레이블에 대한 v선언을 하기도 합니다. 사용자 정의로 축을 사용하기 위해서는 `plot()` 함수 안에 `axes=FALSE` 인수를 지정하는 편이 좋을 것입니다.

Low-level 그래픽 함수들은 대체로 새로운 플롯 구성 요소를 어디에 배치할지 결정하기 위해 위치에 관한 (x, y좌표 같은) 정보를 요구합니다. 이런 좌표들은 이전의 high-level 그래픽 함수 명령문에 의해 정의된 사용자 좌표에 의해 표현되며, 이것은 주어진 데이터에 따라 결정됩니다.

X와 y 인수 값이 필요한 경우, x 와 y로 명명된 성분들을 연결해서 표현한 하나의 인수로 표현하는 것 역시 가능합니다. 마찬가지로 두 개의 열을 가진 행렬을 사용하는 것도 역시 유효한 방법입니다. 이 방법을 사용할 때는 `locator()` (아래 참조) 같은 함수를 사용해서 플롯을 표현하기 위한 여러 위치들을 interactive하게 조정하는 것이 가능합니다.

## Node 121. Mathematical annotation

어떤 경우에는, 수학적 기호나 공식을 플롯에 첨가하는 것이 유용합니다. R에서는 이런 작업을 `text`, `mtext`, `axis` 또는 `title`같은 함수를 사용해서 하나의 문자열을 입력하는 것 보다는 하나의 표현(expression)을 사용해서 해결합니다. 예를 들면, 다음의 코드로는 이항분포 함수(binomial probability function)의 공식을 표현 됩니다:

```
> text(x, y, expression(paste(bgroup("(", atop(n, x), ")"), p^x, q^{n-x})))
```

이 함수에서 사용할 수 있는 모든 기능들을 포함해서 좀 더 많은 정보를 얻을 수 있는 R 명령문들은 다음과 같습니다:

```
> help(plotmath)
> example(plotmath)
> demo(plotmath)
```

## Node 122. Hershey vector fonts

`Text`와 `contour` 함수를 사용할 때, `text`를 조절하기 위해 Hershey폰트를 사용하는 것이 가능합니다. Hershey폰트를 사용하는 것은 다음의 세가지 이유에서 때문입니다:

◆ Hershey폰트는 컴퓨터 스크린에서 `text`를 회전시키거나 작은 `text`를 사용할 때 특히 효과적입니다.

◆ Hershey 폰트는 일관 폰트에서는 제공하지 않는 몇몇 심볼들을 제공합니다. 특히, 조디악 기호 (zodiac sign)나 지도 제작용 (cartographic) 심볼 그리고 천문학(astronomical) 심볼 등을 제공합니다.

◆ Hershey 폰트는 키릴(Cyrillic)과 일본어(Kana와 Kanji 포함) 문자들을 제공합니다.

Hershey 문자표를 포함한 좀 더 많은 정보를 찾기 위해 R에서 사용할 수 있는 명령문은 다음과 같습니다:

```
> help(Hershey)
```



```
> demo(Hershey)
> help(Japanese)
> demo(Japanese)
```

### Node 123. Interacting with graphics

R에서는 마우스를 사용하여 플롯에 관련된 정보를 빼거나 더하는 작업을 하는 것도 가능합니다. 이런 방식으로 작동하는 함수 중 가장 간단한 것은 `locator()` 함수입니다:

`locator(n, type)`

마우스 왼쪽 버튼을 사용하여 현재 플롯에 관련된 위치에 관련된 값들을 선택할 수 있도록 기다리는 함수입니다. 이 함수는 `n` (디폴트는 512)개의 점들이 모두 선택될 까지 또는 다음 번 마우스 버튼을 누를 때까지 계속 작동합니다. `Type` 인수는 선택된 포인트들을 어떤 플롯으로 그릴지 결정하며 `high-level` 그래픽 함수에서와 같은 기능으로 사용됩니다; 디폴트로 플롯을 표시하지 않는 것이 지정되어 있고, `locator()`는 선택된 포인트들의 위치들이 두 개의 `x`와 `y`의 성분으로 이루어진 하나의 리스트 값을 출력합니다.

하지만 `locator()` 함수는 인수들을 입력하지 않은 채 그대로 사용되는 경우가 더 많습니다. 이 함수는 범례나 레이블 같은 그래픽 요소들의 위치를 정할 때 `interactive`하게 사용되는데, 특히 이러한 요소들을 사전에 해당 그래픽의 어느 곳에 배치해야 할지 결정하기 어려운 경우에 유용하게 사용될 수 있습니다. 예를 들면, `Outlier`와 같은 포인트에 해당 포인트에 대한 정보를 입력하기 위해 다음 명령문을 사용할 수 있습니다.

```
> text(locator(1), "Outlier", adj=0)
```

(`locator()` 함수는 현재 디바이스가 `postscript`와 같이 `interactive pointing` 기능을 제공하지 않는 경우에는 그냥 무시되어 버립니다.)

`identify(x, y, labels)`

`x`와 `y`에 의해 정의된 어떤 포인트라도 (마우스의 왼쪽 버튼을 사용해서) 그 근처에 해당 포인트에 대응되는 레이블을 표기함으로써 그 포인트를 확인할 수 있도록 하는 함수입니다. (레이블 정보가 없는 경우, 해당 포인트의 인덱스 번호를 사용) 버튼을 한 번 더 누르면 선택된 포인트들의 인덱스들을 출력합니다.

때때로 우리는 플롯에서 특별한 몇 개의 포인트들의 위치를 보기 보다는, 해당 포인트에 대한 정보를 확인하고 싶어집니다. 예를 들면, 그래픽으로 표현된 데이터 중 관심 있는 몇 개의 관측치만 선택해서 그 관측치들만 따로 처리하고 싶은 경우가 있습니다. 포인트들을 두 개의 숫자 벡터인 `x`와 `y`에 의해 표현된 `(x, y)` 좌표 쌍들로 표현하면, `identify()`를 다음과 같이 사용할 수 있습니다:

```
> plot(x, y)
```

```
> identify(x, y)
```

`identify()` 함수는 그 자체로 플롯을 그리지는 않지만, 사용자들이 마우스 포인터를 움직이다 어떤 포인트에서는 마우스 왼쪽 버튼을 클릭할 수 있는 기능을 제공합니다. 어떤 포인트 근처에 마우스 포인터가 있는 경우, 해당 포인트는 인덱스 번호로 표시될 것 입니다. (이것은, 그 포인트의 위치가 `x/y vector`로 되어 있음을 의미하는 것입니다.) 또한, `identify()` 함수 내에 `labels` 인수를 사용하여 포인트들이 좀 더 `informative`하게(해당 케이스의 이름 등) 표현되도록 할 수도 있으며, `plot = FALSE` 인수를 사용하면 포인트를 선택해도 아무런 표시가 남지 않도록 할 수도 있습니다.

다. 해당 프로세스가 끝나고 나면(위 참조), `identify()`는 선택되었던 포인트들의 인덱스를 출력합니다; 이러한 인덱스 정보를 바탕으로 원래 `x`, `y`의 벡터 형태로 되어 있는 선택된 포인트들의 정보를 따로 추출해내는 것도 가능할 것 입니다.

#### Node 124. Using graphics parameters

그래픽을 만들어 낼 때, 특히 그것이 프레젠테이션이나 다른 공공 목적을 위한 것이라면, R의 디폴트만으로는 항상 원하는 형태를 얻을 수 없을 것입니다. 그렇지만, 그래픽 모수(graphics parameters)를 이용하여 거의 모든 표현 방식을 사용자 임의대로 정의할 수 있습니다. R에는 선의 형태, 색, 그림 배열 그리고 문자 정의 등에 이르기까지 굉장히 다양한 그래픽 모수들이 포함되어 있습니다. 모든 그래픽 모수는 이름(e.g. 색을 결정하는 `col`)과 값(특정색을 의미하는 숫자)으로 구성 됩니다.

각각의 디바이스를 위해 그래픽 모수의 리스트들을 따로 정의하는 것이 가능하고, 각 디바이스들은 모수들의 디폴트 값을 가진 채 초기화 됩니다. 그래픽 모수들은 크게 두 가지 방법으로 사용될 수 있습니다: 하나는 현재 디바이스를 사용할 때마다 모든 그래픽 함수들에 영향을 주는 『영구적』 방법이고 다른 방법은 오직 하나의 그래픽 함수를 사용할 때에만 영향을 주는 『임시적』 방법입니다.

#### Node 125. Permanent changes: The `par()` function

`par()`함수는 현재의 그래픽 디바이스의 그래픽 모수 리스트를 조정하기 위해 사용됩니다.

`par()`

아무런 인수가 없는 경우, 현재 디바이스에 저장되어 있는 모든 그래픽 모수와 그 모수들 해당 값의 리스트를 출력합니다.

`par(c("col", "lty"))`

문자 벡터 인수를 사용하면, 오직 따로 지정된 그래픽 모수들에 대해서만 (이 경우도 리스트의 형태로) 출력합니다.

`par(col=4, lty=2)`

그래픽 모수와 해당 모수의 값들을 따로 인수(하나의 인수도 가능)로 지정하면, 지정된 값이 리스트로 출력됩니다. (?)

`par()`함수를 사용해서 그래픽 모수를 지정하면 해당 모수와 그 값은 『영구적』으로 바뀝니다. 즉, 미래에 해당 그래픽 함수를 (현재 디바이스에서) 사용하면 새로 지정된 값이 반영되어 출력되는 것입니다. 따라서 이 방법을 사용해서 그래픽 모수를 지정하는 것은 일종의 『디폴트』 값을 지정하는 것과 같다고 생각할 수 있을 겁니다. 즉, 다른 값이 다시 지정되지 않는 한, 모든 그래픽 함수가 이 지정 값에 영향을 받게 되는 것입니다.

`par()`를 사용하면, 항상, 심지어 `par()`가 다른 함수 안에서 사용된 경우라도 전체적으로 모든 그래픽 모수들의 값이 영향을 받게 된다는 점을 주의하시기 바랍니다. 이것은 대부분의 경우 그다지 바람직하지 않다고 볼 수 있습니다. 우리가 몇 개의 그래픽 모수를 따로 지정하는 것은 대체로 몇 개의 플롯을 그린 후, 현재의 R 세션이 이 값에 영향 받기 전 상태인 원래의 모수 값으로 돌아가고 싶기 때문입니다. 특별히 모수를 바꾸는 작업이 필요할 때 `par()`의 결과를 따로 저장하고 플롯 작업이 끝난 후에는 초기 값을 다시 사용함으로써 초기값을 계속 유지하는 것이 가능합니다.

```
> oldpar <- par(col=4, lty=2)
```

... plotting commands ...

```
> par(oldpar)
```

조정 가능한 모든 그래픽 모수값을 저장해서 다시 사용하기 위해서는

```
> oldpar <- par(no.readonly=TRUE)
```

... plotting commands ...

```
> par(oldpar)
```

## Node 126. Temporary changes: Arguments to graphics functions

그래픽 모수들은 (거의 모든) 그래픽 함수에서 인수처럼 사용될 수도 있습니다. 이러한 방법은 인수로 사용되었을 때의 변화가 오직 해당 함수가 사용되는 동안만 지속된다는 점을 제외하고는, `par()` 함수에 인수들을 사용하는 것과 같은 효과를 줍니다. 예를 들면:

```
> plot(x, y, pch="+")
```

이 경우, 더하기 표시를 플롯 문자로 사용한 산점도를 그리게 되는데, 이 때 사용된 플롯 문자는 앞으로 사용하게 될 플롯의 디폴트에는 영향을 주지 않습니다.

안타깝지만, 항상 이와 같은 형태로만 실행되는 것이 아니므로 경우에 따라 `par()`를 사용해서 그래픽 모수들을 지정하고 다시 되돌리는 작업이 필요하기도 합니다.

## Node 127. Graphics parameters list

다음 섹션들은 자주 사용되는 다수의 graphical parameters에 대해 자세히 설명 할 것입니다. R은 더욱 간결한 summary를 제공하는 `par()` function을 documentation하는 것을 도와 줍니다; 이것은 약간 더 자세한 다른 방법처럼 제공 될 것입니다.

Graphics parameters는 다음과 같이 표현 될 것입니다:

The following sections detail many of the commonly-used graphical parameters. The R help's documentation for the `par()` function provides a more concise summary; this is provided as a somewhat more detailed alternative.

Graphics parameters will be presented in the following form:

name=value

parameter's effect에 대한 설명입니다. Name은 `par()`이나 graphics function을 부르기 위해 사용하는 argument name인 parameter의 이름입니다. Value는 parameter를 세팅할 때 사용할 수 있는 전형적인 value입니다.

Axes는 graphics parameter이 아니고 몇몇의 plot methods의 argument라는 것을 유의하세요: `xaxt`와 `yaxt`를 참고 하세요.

A description of the parameter's effect. name is the name of the parameter, that is, the argument name to use in calls to `par()` or a graphics function. value is a typical value you might use when setting the parameter.

Note that axes is not a graphics parameter but an argument to a few plot methods: see `xaxt` and `yaxt`.

## Node 128. Graphical elements

R plots는 points, lines, text 그리고 polygons(filled regions)로 구성되어 있습니다.

Graphical parameters는 다음과 같이 graphical elements가 어떻게 그려졌는지를 통제하며 존재합니다:

R plots are made up of points, lines, text and polygons (filled regions.) Graphical parameters exist which control how these graphical elements are drawn, as follows:

`pch="+"`

points들을 표시하기 위한 character. Default는 graphics drivers에 따라 다양하지만, 보통 원형입니다. 표시된 points들은, centered points를 생산하는 plotting character로 "."을 사용하지 않으면 적합한 position보다 약간 위쪽이나 아래쪽에 나타나는 특성이 있습니다.

Character to be used for plotting points. The default varies with graphics drivers, but it is usually a circle. Plotted points tend to appear slightly above or below the appropriate position unless you use "." as the plotting character, which produces centered points.

`pch=4`

pch가 0과 25를 포함하는 정수 사이에 존재할 때는, 전문적인 plotting symbol이 생산됩니다. Symbols가 무엇인지 보기 위해서는, 다음의 command를 사용합니다.

When pch is given as an integer between 0 and 25 inclusive, a specialized plotting symbol is produced. To see what the symbols are, use the command

`> legend(locator(1), as.character(0:25), pch = 0:25)`

21에서 25의 정수들은 이전의 symbols들과 겹치게 나타날 수도 있지만, 다른 색으로 나타낼 수 있습니다: [points and its examples](#)에 관한 help를 참조하세요.

Those from 21 to 25 may appear to duplicate earlier symbols, but can be coloured in different ways: see the help on points and its examples.

또한, pch는 현재의 font 안의 character를 나타내는 32:255 범위내의 character가 될 수도 있고 number가 될 수도 있습니다.

In addition, pch can be a character or a number in the range 32:255 representing a character in the current font.

`lty=2`



line 유형. 다른 line styles는 모든 graphics devices에서 지원하지는 않지만(그리고 지원하는 것들도 아주 다양합니다), line type 1은 항상 solid line이고, line type 0은 항상 보이지 않으며, line types 2와 onwards는 점이나 dashed lines, 아니면 어떤 것은 두 개를 합합니다.

Line types. Alternative line styles are not supported on all graphics devices (and vary on those that do) but line type 1 is always a solid line, line type 0 is always invisible, and line types 2 and onwards are dotted or dashed lines, or some combination of both.

`lwd=2`

line의 넓이. 원하는 lines의 넓이, 여러 개의 standard line의 넓이.

Axis lines와 lines()와 같이 그려진 lines와 다른 것들에 영향을 줍니다. 모든 devices가 이를 지원하지는 않으며 어떤 것은 사용될 수 있는 widths에 대한 제약이 있습니다.

Line widths. Desired width of lines, in multiples of the  standard  line width. Affects axis lines as well as lines drawn with lines(), etc. Not all devices support this,

and some have restrictions on the widths that can be used.

col=2

points, lines, text, filled regions와 images에 색이 사용될 수 있습니다. 현재의 palette(palette를 참조하세요)에서부터의 숫자나 이름이 주어진 colour.

Colors to be used for points, lines, text, filled regions and images. A number from the current palette (see ?palette) or a named colour.

col.axis

col.lab

col.main

col.sub

색은 순서대로 axis annotation, x와 y labes, 제목이나 부제들에 사용됩니다.

The color to be used for axis annotation, x and y labels, main and sub-titles, respectively.

font=2

text에 사용되는 font를 명시하는 정수. 가능하다면, plain text에 상응하는 1, bold face에 2, italic에 3, bold italic에 4, 그리고 symbol font(Greek letters를 포함하는)가 5가 되도록 device drivers를 배열 하세요.

An integer which specifies which font to use for text. If possible, device drivers arrange so that 1 corresponds to plain text, 2 to bold face, 3 to italic, 4 to bold italic and 5 to a symbol font (which include Greek letters).

font.axis

font.lab

font.main

font.sub

font

Font는 순서대로 axis annotation, x와 y labels, 제목과 부제에 사용됩니다.

The font to be used for axis annotation, x and y labels, main and sub-titles, respectively.

adj=-0.1

plorrinf position에 관련된 text의 조정. 0은 왼쪽 조정, 1은 오른쪽 조정, 그리고 0.5는 가운데 plotting position에 대해 수평으로 가운데 조정을 의미합니다. Text proportion의 actual value는 plotting position의 왼쪽에 나타나고 -0.1 value는 text 와 plotting position 사이의 txt width의 10%의 gap을 남깁니다.

Justification of text relative to the plotting position. 0 means left justify, 1 means right justify and 0.5 means to center horizontally about the plotting position. The actual value is the proportion of text that appears to the left of the plotting position, so a value of -0.1 leaves a gap of 10% of the text width between the text and the plotting position.

cex=1.5

확장 character. Value는 default test size에 상대적인 text characters(plotting characters

를 포함하는)의 원하는 사이즈입니다.

Character expansion. The value is the desired size of text characters (including plotting characters) relative to the default text size.

cex.axis

cex.lab

cex.main

cex.sub

확장 character는 순서대로 axis annotation, x와 y labels, 제목과 부제에 사용됩니다.

The character expansion to be used for axis annotation, x and y labels, main and sub-titles, respectively.

### Node 129. **Axes and tick marks**

R의 high-level plots의 많은 것들이 axes를 가지고 있고 low-level axis() graphics function을 이용해 axes를 구성할 수 있습니다. Axes는 세 개의 주 구성 요소를 가지고 있습니다: axis line(graphics parameter에 의해 조정되는 line style), the tick marks(axis line을 따라 unit divisions를 표시하는), 그리고 the tick labels(units를 표시하는)입니다. 이 구성 요소들은 다음의 graphics parameters에 의해 customized 될 수 있습니다.

Many of R's high-level plots have axes, and you can construct axes yourself with the low-level axis() graphics function. Axes have three main components: the axis line (line style controlled by the lty graphics parameter), the tick marks (which mark off unit divisions along the axis line) and the tick labels (which mark the units.) These components can be customized with the following graphics parameters.

lab=c(5, 7, 12)

첫 번째 두 숫자들은 순서대로 x와 y axes 위의 tick intervals의 희망 숫자들입니다. 세 번째 숫자는 characters안의 axis labels의 희망 길이입니다(소수점도 포함). 이 parameter에서 너무 작은 value를 선택한다면, 모든 tick labels가 같은 숫자로 rounded되는 결과를 얻을지도 모릅니다!

The first two numbers are the desired number of tick intervals on the x and y axes respectively. The third number is the desired length of axis labels, in characters (including the decimal point.) Choosing a too-small value for this parameter may result in all tick labels being rounded to the same number!

las=1

axis labels의 orientation. 0은 항상 axis에 평행함을 뜻하고, 1은 항상 수평, 그리고 2는 항상 axis에 수직임을 뜻합니다.

Orientation of axis labels. 0 means always parallel to axis, 1 means always horizontal, and 2 means always perpendicular to the axis.

mgp=c(3, 1, 0)

axis components의 자리. 첫 번째 구성요소는 text lines안의 axis label에서 axis position까지의 거리입니다. 두 번째 구성요소는 tick labels까지의 거리, 그리고 마지막 구성요소는 axis position에서 axis line(보통 0)까지의 거리입니다. 양수는 plot region 바깥쪽을 측정하고, 음수



는 안쪽을 측정합니다.

Positions of axis components. The first component is the distance from the axis label to the axis position, in text lines. The second component is the distance to the tick labels, and the final component is the distance from the axis position to the axis line (usually zero). Positive numbers measure outside the plot region, negative numbers inside.

```
tck=0.01
```

plotting region의 size의 fraction처럼 tick marks의 길이. Tick이 작을 때(0.5보다 작을 경우), x와 y axes위의 tick marks와 같은 사이즈가 되도록 강요됩니다. 1의 value는 grid lines를 제공합니다. 음의 values는 plotting region의 바깥쪽 tick marks를 줍니다. 안쪽의 tick marks에는 tck=0.01 와 mgp=c(1,-1.5,0)를 사용 하세요.

Length of tick marks, as a fraction of the size of the plotting region. When tck is small (less than 0.5) the tick marks on the x and y axes are forced to be the same size. A value of 1 gives grid lines. Negative values give tick marks outside the plotting region. Use tck=0.01 and mgp=c(1,-1.5,0) for internal tick marks.

```
xaxs="r"
```

```
yaxs="i"
```

순서대로 x와 y axes의 axis styles. Styles "i"(안)와 "r"(default) tick marks는 항상 data의 범의 안에 떨어집니다. 하지만 style "r"은 모서리에 작은 space를 남겨둡니다. (S는 R에 실행하지 않는 다른 style을 가지고 있습니다.

Axis styles for the x and y axes, respectively. With styles "i" (internal) and "r" (the default) tick marks always fall within the range of the data, however style "r" leaves a small amount of space at the edges. (S has other styles not implemented in R.)

### Node 130. **Figure margins**

R안의 single pot은 figure이라고 알려져 있고 margin(axis labes, titles 등을 포함할 수 있습니다)으로 둘러싸인 plot region으로 구성되어있고 (보통) axes 자체에 의해 제한되어 있습니다.

Graphics parameters controlling figure layout은 아래를 포함합니다:

A single plot in R is known as a figure and comprises a plot region surrounded by margins (possibly containing axis labels, titles, etc.) and (usually) bounded by the axes themselves.

Graphics parameters controlling figure layout include:

```
mai=c(1, 0.5, 0.5, 0)
```

순서대로 바닥, 왼쪽, 위쪽, 그리고 오른쪽 margin의 inches로 재어진 넓이

Widths of the bottom, left, top and right margins, respectively, measured in inches.

```
mar=c(4, 2, 2, 1)
```

mai와 비슷하지만 측정 단위가 다른 text lines입니다.

Mar과 mai는 value를 다른 것으로 바꾸는 setting이라는 점에서 같습니다. 이 parameter에 의해 선정된 default values는 가끔 너무 큼니다: 오른쪽 margin은 거의 필요하지 않고 위쪽 margin도 제목이 없다면 그다지 필요하지 않습니다. 밑쪽과 왼쪽 margins는 axis와 tick labels

를 수용하기에 충분히 커야만 합니다. 더 나아가, default는 device surface 사이즈에 상관없이 선택 됩니다: 예를들어, height=4 argument를 가진 postscript() driver를 이용하면 mar이나 mai가 명시적으로 배치되지 않았다면 50%정도의 margin의 plot 결과를 줄 것입니다. Multiple figures가 사용 중일 때(아래를 참조하세요), margins가 줄어들지만, 이 방법은 많은 figure들이 같은 page를 공유하고 있을 때는 충분하지 않습니다.

Similar to mai, except the measurement unit is text lines.

mar and mai are equivalent in the sense that setting one changes the value of the other. The default values chosen for this parameter are often too large; the right-hand margin is rarely needed, and neither is the top margin if no title is being used. The bottom and left margins must be large enough to accommodate the axis and tick labels. Furthermore, the default is chosen without regard to the size of the device surface: for example, using the postscript() driver with the height=4 argument will result in a plot which is about 50% margin unless mar or mai are set explicitly. When multiple figures are in use (see below) the margins are reduced, however this may not be enough when many figures share the same page.

### Node 131. Multiple figure environment

R은 사용자가 하나의 page에 n by m array figure를 만들 수 있도록 합니다. 각각의 figure은 각각의 margin이 있으며 figures의 array는 아래와 같이 바깥쪽 margin에 의해 임의로 둘러싸여 있습니다.

Graphical parameters는 다음과 같이 multiple figures에 연관되어 있습니다:

R allows you to create an n by m array of figures on a single page. Each figure has its own margins, and the array of figures is optionally surrounded by an outer margin, as shown in the following figure.

The graphical parameters relating to multiple figures are as follows:

```
mfc col=c(3, 2)
```

```
mfc row=c(2, 4)
```

multiple figure array의 사이즈를 설정. 첫 번째 value는 row의 개수; 두 번째는 columns의 개수 입니다. 이 두 개의 parameters의 유일한 다른 점은 mfc col 설정은 figures가 column에 의해 차고 mfc row는 row에 의해 찬다는 것입니다.

figure안의 layout은 mfc row=(3,2)의 설정에 의해 만들어질 수 있습니다; 그 figure은 4개의 plots가 그려진 후에 page를 보여줍니다.

Set the size of a multiple figure array. The first value is the number of rows; the second is the number of columns. The only difference between these two parameters is that setting mfc col causes figures to be filled by column; mfc row fills by rows.

The layout in the Figure could have been created by setting mfc row=c(3,2); the figure shows the page after four plots have been drawn.

이들 중 어느 쪽을 설정하던 symbols의 베이스 사이즈와 text(par("cex")와 pointsize of the device를 조정하는)를 줄일 수 있습니다. 정확히 2개의 rows와 columns가 있는 layout 안에, base size는 0.83의 약수로 줄어듭니다: 만약 세 개 이상의 rows나 column가 있다면, 줄어드는



factor는 0.66입니다.

Setting either of these can reduce the base size of symbols and text (controlled by `par("cex")` and the pointsizes of the device). In a layout with exactly two rows and columns the base size is reduced by a factor of 0.83: if there are three or more of either rows or columns, the reduction factor is 0.66.

```
mfg=c(2, 2, 3, 2)
```

**multiple figure 환경 안의 현재 figure의 위치.** 처음 두 개의 숫자는 현재의 figure의 row와 column입니다; 마지막 두 개의 숫자는 multiple figure array안의 rows와 column의 숫자입니다. array안의 figures사이를 jump하기 위해 이 parameter를 설정 하세요. 마지막 두 개의 숫자들에 대해 같은 page에 있는 unequally-sized figures에 대한 true values가 아닌 다른 values를 이용 할 수도 있습니다.

Position of the current figure in a multiple figure environment. The first two numbers are the row and column of the current figure; the last two are the number of rows and columns in the multiple figure array. Set this parameter to jump between figures in the array. You can even use different values for the last two numbers than the true values for unequally-sized figures on the same page.

```
fig=c(4, 9, 1, 4)/10
```

**page에 있는 현재 figure의 위치.** Values는 pages의 확률이 왼쪽 밑에서부터 재어진 것 처럼 순서대로 왼쪽, 오른쪽, 밑, 그리고 위쪽의 모서리입니다. 예제의 value는 페이지의 오른쪽 밑에 속한 figure을 위한 것일 것입니다. 현재 페이지에 figure을 더하고 싶다면 new=TRUE를 사용하세요(S와는 다르게).

Position of the current figure on the page. Values are the positions of the left, right, bottom and top edges respectively, as a percentage of the page measured from the bottom left corner. The example value would be for a figure in the bottom right of the page. Set this parameter for arbitrary positioning of figures within a page. If you want to add a figure to a current page, use new=TRUE as well (unlike S).

```
oma=c(2, 0, 3, 0)
```

```
omi=c(0, 0, 0.8, 0)
```

바깥쪽 margins의 크기. Mar과 mai와 같이, inches로 된 text lines의 첫 번째 치수와 두 번째 치수는 밑 margin에서 시작하여 시계 방향으로 작동합니다.

**바깥쪽 margins는 page-wise 제목 등에 특히 유용합니다.** Text는 argument outer=TRUE를 가지고 있는 mtext () function을 가진 바깥쪽 margin에 더해질 수 있습니다. Default에 의한 바깥쪽 margins는 없지만 그들을 oma나 omi를 사용해서 명백하게 만들어야만 합니다.

Size of outer margins. Like mar and mai, the first measures in text lines and the second in inches, starting with the bottom margin and working clockwise.

Outer margins are particularly useful for page-wise titles, etc. Text can be added to the outer margins with the mtext() function with argument outer=TRUE. There are no outer margins by default, however, so you must create them explicitly using oma or omi.

**Multiple figures의 더 복잡한 배합은 split.screen()과 layout() functions, 그리고 grid and**

lattice packages에 의해 생산될 수 있습니다.

More complicated arrangements of multiple figures can be produced by the `split.screen()` and `layout()` functions, as well as by the grid and lattice packages.

## Node 132. Device drivers

R은 거의 모든 디스플레이 타입에 (다른 quality levels의) 그래픽을 생성할 수 있습니다. 하지만 이것을 시작하기 전에, R은 어떤 type의 device를 취급하는지를 알고 있어야 합니다. 이것은 device driver를 시작하면 됩니다. Device driver의 목적은 R에서 특정 device가 이해되도록 graphical instructions를 바꾸는 것입니다.

R can generate graphics (of varying levels of quality) on almost any type of display or printing device. Before this can begin, however, R needs to be informed what type of device it is dealing with. This is done by starting a device driver. The purpose of a device driver is to convert graphical instructions from R (◆draw a line,◆ for example) into a form that the particular device can understand.

Device driver는 device driver function을 부름으로써 시작됩니다. 여기에는 각각의 device driver에 대해 한 개의 function이 존재합니다: 그것들을 전부 나열하려면 `help(Devices)`를 입력하세요. 예를 들어, 명령을 내리는 것은

Device drivers are started by calling a device driver function. There is one such function for every device driver: type `help(Devices)` for a list of them all. For example, issuing the command

```
> postscript()
```

모든 차후의 PostScript format으로 프린터에 보내질 graphics output에 영향을 줍니다. 몇몇의 자주 사용되는 device drivers들은:

causes all future graphics output to be sent to the printer in PostScript format. Some commonly-used device drivers are:

`X11()`

Unix와 비슷한 X11 윈도우 시스템과 사용하기 위함

For use with the X11 window system on Unix-alikes  
`windows()`

윈도우에서 사용하기 위함

For use on Windows

`quartz()`

Mac OS X에서 사용하기 위함

For use on Mac OS X

`postscript()`

PostScript printers에 프린트 하거나 PostScript 그래픽 파일들을 만들기 위함

For printing on PostScript printers, or creating PostScript graphics files.  
`pdf()`

PDF 파일에 포함될 수 있는 PDF 파일을 생산함.

Produces a PDF file, which can also be included into PDF files.

png()

비트맵 PNG 파일을 생산함. (항상 쓸 수 있지는 않음: help 페이지를 참조)

Produces a bitmap PNG file. (Not always available: see its help page.)

jpeg()

image plots에 가장 바람직하게 사용되는 비트맵 JPEG 파일을 생산함. (항상 쓸 수 있지는 않음: help 페이지를 참조)

Produces a bitmap JPEG file, best used for image plots. (Not always available: see its help page.)

Device를 끝냈으면, command를 내림으로써 device driver를 끝내는 것을 잊지 마세요.

When you have finished with a device, be sure to terminate the device driver by issuing the command

```
> dev.off()
```

이것은 device가 깨끗하게 끝났음을 확실하게 합니다; 예를 들어 hardcopy devices의 경우 이것은 모든 페이지가 완료 되었고 프린터로 보내졌음을 확실히 합니다. (이는 normal 세션 마지막 부분에 자동적으로 될 것입니다.)

This ensures that the device finishes cleanly; for example in the case of hardcopy devices this ensures that every page is completed and has been sent to the printer. (This will happen automatically at the normal end of a session.)

Node 133.

### PostScript diagrams for typeset documents

파일 명령문을 postscript()라는 device driver함수로 보냄으로써, 그래픽들을 당신이 원하는 파일 안에 PostScript 포맷의 graphics를 저장 할 수도 있습니다. Plot은 horizontal=FALSE argument이 주어지지 않은 이상 landscape의 형태를 지니고, 넓이와 높이 arguments를 가지고 graphic의 크기를 조정할 수 있을 것입니다. (plot은 이들의 크기에 적합하게 재어질 것입니다.) 예를 들어, 명령

By passing the file argument to the postscript() device driver function, you may store the graphics in PostScript format in a file of your choice. The plot will be in landscape orientation unless the horizontal=FALSE argument is given, and you can control the size of the graphic with the width and height arguments (the plot will be scaled as appropriate to fit these dimensions.) For example, the command

```
> postscript("file.ps", horizontal=FALSE, height=5, pointsize=10)
```

은 5인치 높이의 figure을 위한 PostScript 코드를 가지고 있는 파일을 생성할 것이고, 문서에 포함 될 지도 모릅니다. 이는 중요한 요점으로 만약 파일이 이미 존재하는 명령 안에 지어진다면, 덮어 쓰여진 것이 됩니다. 이런 케이스는 같은 R 세션에서 먼저 만들어 진 경우에도 해당됩니다.

will produce a file containing PostScript code for a figure five inches high, perhaps for inclusion in a document. It is important to note that if the file named in the command already exists, it will be overwritten. This is the case even if the file was only created earlier in the same R session.

PostScript output의 많은 용도는 다른 문서의 figure을 포함하고 있는 것 입니다. 이는 요약된

PostScript가 생산되었을 때 가장 잘 working합니다: R은 항상 conformant output을 생산하지만, 단지 onefile=FALSE argument가 제공 되었을 때의 output만 표시합니다. 이 예외적인 notation stems는 S compatibility에서 나온 것 입니다: 이것은 output이 single page에 있을 것이라는 뜻입니다(EPSF specification의 일부인). 그러므로 inclusion을 위한 plot을 생산하기 위해서는 다음과 같이,

Many usages of PostScript output will be to incorporate the figure in another document. This works best when encapsulated PostScript is produced: R always produces conformant output, but only marks the output as such when the onefile=FALSE argument is supplied. This unusual notation stems from S-compatibility: it really means that the output will be a single page (which is part of the EPSF specification). Thus to produce a plot for inclusion use something like

```
> postscript("plot1.eps", horizontal=FALSE, onefile=FALSE,  
height=8, width=6, pointsize=10)
```

#### Node 134. **Multiple graphics devices**

좀더 advanced된 R의 사용은 가끔 여러 개의 graphics devices가 같은 시간에 사용되고 있을 때 유용합니다. 당연히 한 개의 graphics device는 아무 시간이나 graphics commands를 수락할 수 있고, 이는 current device라고 알려져 있습니다. Multiple devices가 열렸을 때, 이는 어떤 위치에서나 device 종류에 이름을 주는 숫자 sequence를 생성합니다.

In advanced use of R it is often useful to have several graphics devices in use at the same time. Of course only one graphics device can accept graphics commands at any one time, and this is known as the current device. When multiple devices are open, they form a numbered sequence with names giving the kind of device at any position.

Main commands는 multiple devices와 함께 operate되는 것으로 사용되고, 그 뜻들은 다음과 같습니다:

The main commands used for operating with multiple devices, and their meanings are as follows:

```
X11()  
[UNIX]  
windows()  
win.printer()  
win.metafile()  
[Windows]  
quartz()  
[Mac OS X]  
postscript()  
pdf()  
png()  
jpeg()  
tiff()
```

bitmap()

...

각각의 device driver function으로의 새로운 call들은 새로운 graphics device를 열고, 그림으로써 device list를 한 개 더 늘립니다. 이 device는 graphics output이 보내질 현재 device가 됩니다.

Each new call to a device driver function opens a new graphics device, thus extending by one the device list. This device becomes the current device, to which graphics output will be sent.

dev.list()

모든 active devices의 숫자와 이름을 돌려줌. List에 있는 position 1의 device는 항상 graphics commands를 절대 받지 않는 null device입니다.

Returns the number and name of all active devices. The device at position 1 on the list is always the null device which does not accept graphics commands at all.

dev.next()

dev.prev()

순서대로 현재의 device 다음 또는 전의 graphics device의 숫자와 이름을 돌려줌.

Returns the number and name of the graphics device next to, or previous to the current device, respectively.

dev.set(which=k)

device list의 position k의 현재 graphic device를 바꾸는 데 사용할 수 있음. Device의 숫자와 label을 돌려줌.

Can be used to change the current graphics device to the one at position k of the device list. Returns the number and label of the device.

dev.off(k)

device list의 point k에 있는 graphics device를 종결시킴. 몇몇의 devices, postscript devices 같은 것,에서는 device가 어떻게 initiated 되었는가에 따라 파일을 바로 프린트를 하거나 나중에 할 프린팅에 완전히 끝낸 파일을 프린트 할 것입니다.

Terminate the graphics device at point k of the device list. For some devices, such as postscript devices, this will either print the file immediately or correctly complete the file for later printing, depending on how the device was initiated.

dev.copy(device, ..., which=k)

dev.print(device, ..., which=k)

device k의 카피를 만듦. 여기에서 device는 extra arguments를 가진 postscript같은 device function입니다. 만약 필요하다면 ♦...♦. dev.print 를 명시해 놓은 것은 비슷하지만 hard copies 같은 것을 프린트하는 마지막 actions가 바로 실행 되도록 복사된 device는 바로 닫아 집니다.


Make a copy of the device k. Here device is a device function, such as postscript, with extra arguments, if needed, specified by ♦...♦. dev.print is similar, but the copied device is immediately closed, so that end actions, such as printing hardcopies, are immediately performed.

graphics.off()

null device를 제외한 list에 있는 모든 graphics devices을 종료함.

Terminate all graphics devices on the list, except the null device.

### Node 135. **Dynamic graphics**

Dynamic 또는 interactive한 그래픽 built-in기능을 제공하지 않습니다. (e.g. rotating point clouds or to  brushing  (interactively highlighting) points) 그렇지만, Swayne, Cook과 Buja에 의해 개발된 GGobi 시스템 (<http://www.ggobi.org>)을 통해 충분한 dynamic 그래픽 기능들을 사용하는 것이 가능합니다. 이러한 GGobi 기능들을 이용하기 위해서는 R의 패키지 중의 하나인 rggobi를 사용해야 합니다. (사용법: <http://www.ggobi.org/rggobi>)

또한, rgl 패키지는 3D plot을 사용하여 이러한 plot의 표면(surface) 등에서 interactive하게 작업할 수 있도록 해 줍니다.

### Node 140. **A sample session**

다음의 세션은 R을 사용하면서 R environment의 몇가지 features를 설명하기 위함입니다. 많은 시스템 features는 처음에는 친숙하지 않고 당황스럽겠지만, 이 당황스러움은 곧 사라질 것입니다.

The following session is intended to introduce to you some features of the R environment by using them. Many features of the system will be unfamiliar and puzzling at first, but this puzzlement will soon disappear.

\$ R

로그인, 윈도우 시스템을 시작.

R을 적합한 platform으로 시작.

배너와 함께 R 프로그램이 시작됨.

(R에서, 왼쪽의 prompt는 혼란을 피하기 위해 보이지 않을 것입니다.)

Login, start your windowing system.

Start R as appropriate for your platform.

The R program begins, with a banner

(Within R, the prompt on the left hand side will not be shown to avoid confusion.)

help.start()

온라인 help를 위해 (사용자의 기계에 사용 가능한 웹 브라우저를 사용하여) HTML 인터페이스를 시작. 마우스를 사용해 이 facility의 features를 간단하게 탐색할 수 있어야 합니다. Help window를 아이콘화 하고 다음 파트로 넘어가세요.

Start the HTML interface to on-line help (using a web browser available at your machine). You should briefly explore the features of this facility with the mouse. Iconify the help window and move on to the next part.

x <- rnorm(50)

y <- rnorm(x)

두개의 x 와 y 좌표의 two pseudo-random normal vectors를 생성.

Generate two pseudo-random normal vectors of x- and y-coordinates.

```
plot(x, y)
```

평면에 points를 표시. Graphics window는 자동으로 나타날 것입니다.

Plot the points in the plane. A graphics window will appear automatically.

```
ls()
```

현재 R workspace에 어떤 R object가 있는지 봄.

See which R objects are now in the R workspace.

```
rm(x, y)
```

더 이상 필요하지 않은 objects들을 지움.

Remove objects no longer needed. (Clean up).

```
x <- 1:20
```

Make  $x = (1, 2, \dots, 20)$ .

```
w <- 1 + sqrt(x)/2
```

standard deviations의 'weight' vector

A 'weight' vector of standard deviations

```
dummy <- data.frame(x=x, y= x + rnorm(x)*w)
```

두개의 columns와 x와 y의 data frame을 만들고 봄.

Make a data frame of two columns, x and y, and look at it.

```
fm <- lm(y ~ x, data=dummy)
```

```
summary(fm)
```

simple linear regression을 맞추고 analysis를 봄. Tilde의 왼쪽에 있는 y로, x에 대해 dependent인 y를 모델링 합니다.

Fit a simple linear regression and look at the analysis. With y to the left of the tilde, we are modelling y dependent on x.

```
fm1 <- lm(y ~ x, data=dummy, weight=1/w^2)
```

```
summary(fm1)
```

standard deviations를 알기 때문에 우리는 weighted regression을 할 수 있습니다.

Since we know the standard deviations, we can do a weighted regression

```
attach(dummy)
```

data frame 안에 columns를 variables처럼 보이도록 만듦.

Make the columns in the data frame visible as variables

```
lrf <- lowess(x, y)
```

nonparametric local regression function을 만듦.

Make a nonparametric local regression function

```
plot(x, y)
```

Standard point plot.

```
lines(x, lrf$y)
```

local regression을 더함.

Add in the local regression.

```
abline(0, 1, lty=3)
```

The true regression line: (intercept 0, slope 1).

```
abline(coef(fm))
```

Unweighted regression line.

```
abline(coef(fm1), col = "red")
```

Weighted regression line.

```
detach()
```

search path에서부터 data frame을 지움.

Remove data frame from the search path.

```
plot(fitted(fm), resid(fm),
```

```
xlab="Fitted values",
```

```
ylab="Residuals",
```

```
main="Residuals vs Fitted")
```

heteroscedasticity를 체크하기 위한 standard regression diagnostic plot. 보이시나요?

A standard regression diagnostic plot to check for heteroscedasticity. Can you see it?

```
qqnorm(resid(fm), main="Residuals Rankit Plot")
```

skewness, kurtosis와 outliers를 체크하기 위한 normal scores plot. (여기서는 별로 유용하지 않습니다.)

A normal scores plot to check for skewness, kurtosis and outliers. (Not very useful here.)

```
rm(fm, fm1, lrf, x, dummy)
```

다시 지움.

Clean up again.

다음 섹션에서는 Machaelson의 classical experiment와 빛의 속도를 재기 위한 Morley의 data를 볼 것 입니다. 이 dataset은 morley object에서 찾아볼 수 있지만 우리는 read.table function을 설명하기 위해 읽을 것 입니다.

The next section will look at data from the classical experiment of Michaelson and Morley to measure the speed of light. This dataset is available in the morley object, but we will read it to illustrate the read.table function.

```
filepath <- system.file("data", "morley.tab" , package="datasets")
```

data file에 path를 얻음.

Get the path to the data file.

```
mm <- read.table(filepath)
```

Michaelson과 Morley data를 data frame처럼 읽고 봄. 다섯개의 experiments( column Expt)이 있고 각각에는 20개의 runs( column Run)이 있으며 sl은 알맞게 coded된 기록된 빛의 속도 입니다.

Read in the Michaelson and Morley data as a data frame, and look at it. There are five experiments (column Expt) and each has 20 runs (column Run) and sl is the recorded speed of light, suitably coded.

```
mm$Expt <- factor(mm$Expt)
```

```
mm$Run <- factor(mm$Run)
```

Expt와 Run을 factos로 바꿈.



Change Expt and Run into factors

```
attach(mm)
```

position 3(default)에서 data frame을 보이게끔 만듦.

Make the data frame visible at position 3 (the default).

```
plot(Expt, Speed, main="Speed of Light Data", xlab="Experiment No.")
```

simple boxplots를 이용해 다섯 개의 experiments를 비교.

Compare the five experiments with simple boxplots

```
fm <- aov(Speed ~ Run + Expt, data=mm)
```

```
summary(fm)
```

'runs'와 'experiments'를 factor처럼 하여 randomized block처럼 분석.

Analyze as a randomized block, with 'runs' and 'experiments' as factors.

```
fm0 <- update(fm, . ~ . - Run)
```

```
anova(fm0, fm)
```

'runs'를 없애면서 sub-model을 맞추고 이전의 analysis of variance를 이용하여 비교.

Fit the sub-model omitting 'runs', and compare using a formal analysis of variance

```
detach()
```

```
rm(fm, fm0)
```

다음으로 가기 전에 clean up.

Clean up before moving on.

이제 조금 더 많은 graphical features를 알아 봅시다: contour과 image plots.

We now look at some more graphical features: contour and image plots.

```
x <- seq(-pi, pi, len=50)
```

```
y <- x
```

x는  $-\pi \leq x \leq \pi$  안의 50 개의 똑같이 spaced된 vector이고 y도 같습니다.

x is a vector of 50 equally spaced values in  $-\pi \leq x \leq \pi$ . y is the same

```
f <- outer(x, y, function(x, y) cos(y)/(1 + x^2))
```

f는 순서대로 x와 y에 의해 indexed 된 rows와 column로 된 function values의 square matrix입니다.

f is a square matrix, with rows and columns indexed by x and y respectively, of values of the function  $\cos(y) = (1 + x^2)$ .

```
oldpar <- par(no.readonly = TRUE)
```

```
par(pty="s")
```

plotting parameters를 저장하고 plotting region을 "square"로 설정합니다.

Save the plotting parameters and set the plotting region to "square".

```
contour(x, y, f)
```

```
contour(x, y, f, nlevels=15, add=TRUE)
```

f의 contour map을 만듦; 더 자세한 사항을 위해서는 lines를 더하세요.

Make a contour map of  $f$ ; add in more lines for more detail.

```
fa <- (f-t(f))/2
```

$fa$ 는  $f$ 의 "asymmetric part"입니다. ( $t()$ 는 transpose)

$fa$  is the "asymmetric part" of  $f$ . ( $t()$  is transpose).

```
contour(x, y, fa, nlevels=15)
```

contour plot을 만들고,

Make a contour plot,

```
par(oldpar)
```

전의 graphics parameters를 복구하세요.

. . . and restore the old graphics parameters

```
image(x, y, f)
```

```
image(x, y, fa)
```

몇 개의 고밀도의 (원한다면 hardcopies를 얻을 수 있는) image plots를 만들고,

Make some high density image plots, (of which you can get hardcopies if you wish  
objects()); `rm(x, y, f, fa)`

다음으로 가기 전에 clean up하세요.

R은 복잡한 산수도 할 수 있습니다.

. . . and clean up before moving on.

R can do complex arithmetic, also.

```
th <- seq(-pi, pi, len=100)
```

```
z <- exp(1i*th)
```

$1i$ 는 복소수  $i$ 에 사용됩니다.

$1i$  is used for the complex number  $i$ .

```
par(pty="s")
```

```
plot(z, type="l")
```

complex arguments를 plot한다는 것은 imaginary 대비 real parts를 plot하는 것 입니다.

이것은 원형이어야 합니다.

Plotting complex arguments means plot imaginary versus real parts. This should be a circle.

```
w <- rnorm(100) + rnorm(100)*1i
```

unit circle안에 sample points를 원한다고 가정합시다. 한가지 방법은 standard normal real과 imaginary parts로 복소수를 take하는 것이고,

Suppose we want to sample points within the unit circle. One method would be to take complex numbers with standard normal real and imaginary parts . . .

```
w <- ifelse(Mod(w) > 1, 1/w, w)
```

그들의 상호관계에 원 바깥쪽을 그리는 것입니다.

. . . and to map any outside the circle onto their reciprocal.

```
plot(w, xlim=c(-1,1), ylim=c(-1,1), pch="+", xlab="x", ylab="y")
```

```
lines(z)
```

모든 points는 unit circle안에 있지만, distribution은 일정하지 않습니다.

All points are inside the unit circle, but the distribution is not uniform

```
w <- sqrt(runif(100))*exp(2*pi*runif(100)*1i)
```

```
plot(w, xlim=c(-1,1), ylim=c(-1,1), pch="+", xlab="x", ylab="y")
```

```
lines(z)
```

두번째 방법은 일정한 distribution을 사용하는 것입니다. Points는 이제 disc에 더욱 고르게 간격을 두고 있는 것을 볼 것입니다.

The second method uses the uniform distribution. The points should now look more evenly spaced over the disc.

```
rm(th, w, z)
```

다시 clean up하세요.

Clean up again.

```
q()
```

R 프로그램을 종료하세요. R workspace를 저장할 것이냐고 물어볼텐데, 이런 시험적인 세션은 저장하지 않아도 좋습니다.

Quit the R program. You will be asked if you want to save the R workspace, and for an exploratory session like this, you probably do not want to save it.

#### Node 142. Invoking R from the command line

UNIX나 Windows의 command line에서 작업할 때, 명령어 'R'은 form R [options] [<infile] [>outfile] 형태의 main R 프로그램을 시작하거나 "직접적으로"라고 불려지려고 하지 않는 여러가지 R tools의 wrapper처럼 R CMD interface를 통해 사용 될 수 있습니다(예로, R 문서 format이나 add-on 패키지를 다루는 processing files). Environment variable TMPDIR이 설정되지 않았거나 그것이 임시 파일이나 directories를 만들기위한 적합한 장소를 가리키고있는지 확인해야 합니다. 대부분의 옵션들은 R 세션의 시작과 끝에 어떤일이 벌어지는지를 조정합니다. Startup mechanism은 다음과 같습니다. (좀 더 자세한 사항에 대해서는 'Startup' 주제의 온라인 help를 참조하시고, 아래의 세션은 윈도우를 위한 몇가지 세부사항입니다.)

When working in UNIX or at a command line in Windows, the command 'R' can be used both for starting the main R program in the form R [options] [<infile] [>outfile], or, via the R CMD interface, as a wrapper to various R tools (e.g., for processing files in R documentation format or manipulating add-on packages) which are not intended to be called "directly". You need to ensure that either the environment variable TMPDIR is unset or it points to a valid place to create temporary files and directories. Most options control what happens at the beginning and at the end of an R session. The startup mechanism is as follows (see also the on-line help for topic 'Startup' for more information, and the section below for some Windows-specific details).

- '-no-environ'이 주어지지 않은 이상, R은 environment variables를 설정하기 위해서 사용자와 site file들을 찾습니다. Site file의 이름은 environment variable R\_ENVIRON에 의해 지적됩니다; 만약 이것이 설정되어있지 않다면, 'R\_HOME/etc/Renviron.site' 이 (만약 존재 한다면) 사용됩니다. User file은 만약 설정되어 있다면 the environment variable R\_ENVIRON\_USER에 의해 지적됩니다; 그렇지 않으면, 현재나 사용자의

home directory에 있는 ``.Renviron`` 파일을 찾을 것 입니다. 이 파일들은 ``name=value`` 형태의 line을 가지고 있어야만 합니다. (더 정확한 설명을 위해서는 `help("Startup")`를 참조하세요.) 당신이 설정하고자하는 variables는 `R_ PAPERSIZE` (the default 페이퍼 사이즈), `R_PRINTCMD` (the default 프린트 명령), 그리고 `R_LIBS` (specifies the list of R library trees searched for add-on packages)입니다.

- Unless ``-no-environ`` was given, R searches for user and site files to process for setting environment variables. The name of the site file is the one pointed to by the environment variable `R_ENVIRON`; if this is unset, ``R_HOME/etc/Renviron.site`` is used (if it exists). The user file is the one pointed to by the environment variable `R_ENVIRON_USER` if this is set; otherwise, files ``.Renviron`` in the current or in the user's home directory (in that order) are searched for. These files should contain lines of the form ``name=value``. (See `help("Startup")` for a precise description.) Variables you might want to set include `R_ PAPERSIZE` (the default paper size), `R_PRINTCMD` (the default print command) and `R_LIBS` (specifies the list of R library trees searched for add-on packages).
- 그러면 R은 옵션 command line ``-no-site-file``이 주어지지 않은 이상은 사이트 전체의 startup을 검색할 것입니다. 이 파일들의 이름은 `R_PROFILE` environment variable의 value에서부터 얻어졌습니다. 만약 variable들이 설정되지 않았을 경우, default ``R_HOME/etc/Rprofile.site``가 존재한다면 이것이 사용됩니다.
- Then R searches for the site-wide startup profile unless the command line option ``-no-site-file`` was given. The name of this file is taken from the value of the `R_PROFILE` environment variable. If that variable is unset, the default ``R_HOME/etc/Rprofile.site`` is used if this exists.
- 그리고, ``-no-init-file``가 주어지지 않은 이상, R은 사용자 프로필을 검색하고 출처를 밝힙니다. 이 파일들의 이름은 environment variable `R_PROFILE_USER`에서 얻어졌습니다; 만약 설정되지 않았다면, 파일은 현재 directory나 사용자의 home directory의 ``.Rprofile``이라는 파일이 찾을 것입니다.
- Then, unless ``-no-init-file`` was given, R searches for a user profile and sources it. The name of this file is taken from the environment variable `R_PROFILE_USER`; if unset, a file called ``.Rprofile`` in the current directory or in the user's home directory (in that order) is searched for.
- 이는 ( ``-no-restore`` 나 ``-no-restore-data``가 지정되어있지 않은 이상) ``.RData``가 존재한다면 이것에서 부터의 저장된 이미지도 불러올 것입니다.
- It also loads a saved image from ``.RData`` if there is one (unless ``-no-restore`` or ``-no-restore-data`` was specified).
- 마지막으로, 만약 function `.First`가 존재한다면, 이는 executed될 것입니다. 이 function은 (R 세션의 마지막에 executed되는 `.Last`도 마찬가지입니다.) 적합한 startup 프로필들을 정의 할 수도 있고 ``.RData``안에 존재할 수도 있습니다.
- Finally, if a function `.First` exists, it is executed. This function (as well as `.Last`

which is executed at the end of the R session) can be defined in the appropriate startup profiles, or reside in ``.RData``.

또한, R 프로세스에서 사용가능한 메모리를 조정하는 옵션이 있습니다. (좀 더 많은 정보를 얻고 싶다면, 온라인 help 주제 'Memory'를 참조하세요.) 사용자는 보통 R에 의해 사용된 메모리 용량을 제한하려고 하지만 않는다면 이것들을 사용하지 않아도 됩니다. R은 다음의 command-line 옵션을 허용합니다.

In addition, there are options for controlling the memory available to the R process (see the on-line help for topic 'Memory' for more information). Users will not normally need to use these unless they are trying to limit the amount of memory used by R. R accepts the following command-line options.

`--help`

`-h`

Standard output에 짧은 help 메시지를 프린트하고 성공적으로 exit합니다.

Print short help message to standard output and exit successfully

`--version`

Standard output의 버전 정보를 프린트하고 성공적으로 exit합니다.

Print version information to standard output and exit successfully.

`--encoding=enc`

console이나 stdin에서부터의 input을 가정하기 위해 인코딩을 지정하세요. 이는 iconv라고 알려져 있는 encoding이어야만 합니다: help 페이지를 참조하세요.

Specify the encoding to be assumed for input from the console or stdin. This needs to be an encoding known to iconv: see its help page.

`RHOME`

Standard output의 R "home directory"의 path를 프린트하고 성공적으로 exit하세요. front-end shell script와 main page와 따로, R installation은 모든 것(executables, 패키지 등)을 이 directory에 넣습니다.

Print the path to the R "home directory" to standard output and exit successfully. Apart from the front-end shell script and the main page, R installation puts everything (executables, packages, etc.) into this directory.

`--save`

`--no-save`

Data sets가 R세션 마지막에 저장되어야 하는지 마는지를 조정하세요. 만약 어떤 interactive session도 주어지지 않았다면, 사용자는 q()가 있는 세션을 종료할 때 원하는 behavior을 질문 받을 것입니다; non-interactive use에서는 한 개의 다음의 것들이 지정되어 있거나 다른 옵션에 의해 암시되어야 합니다. (아래를 보세요)

Control whether data sets should be saved or not at the end of the R session. If neither is given in an interactive session, the user is asked for the desired behavior when ending the session with q(); in non-interactive use one of these must be specified or implied by some other option (see below).

`--no-enviro`

Environment variable를 설정하기위해 어떠한 사용자 파일도 읽지마세요.

Do not read any user file to set environment variables

'--no-site-file'

Startup때 사이트 전체의 프로필을 읽지마세요.

Do not read the site-wide profile at startup.

'--no-init-file'

Startup때 사용자 프로필을 읽지마세요.

Do not read the user's profile at startup

'--restore'

'--no-restore'

'--no-restore-data'

저장된 이미지(R이 시작된 directory에 있는 '.RData' 파일)가 start때 복구 되어야 하는지 마는지를 조정. Default는 복구 됩니다. ('-no-restore'는 모든 특정한 '-no-restore-\*' options를 암시합니다.)

Control whether saved images (file '.RData' in the directory where R was started) should be restored at startup or not. The default is to restore. ('-no-restore' implies all the specific '-no-restore-\*' options.)

'--no-restore-history'

History 파일(보통 R이 시작된 directory안의 '.Rhistory' 파일이지만, the environment variable R\_HISTFILE에의한 set일 수도 있습니다.)이 startup때 복구 되어야 하는지 마는지를 조정. Default는 복구됩니다.

Control whether the history file (normally file '.Rhistory' in the directory where R was started, but can be set by the environment variable R\_HISTFILE) should be restored at startup or not. The default is to restore.

'--no-Rconsole'

(윈도우일 경우에만) startup 때 'Rconsole'가 loading되는걸 방지.

(Windows only) Prevent loading the 'Rconsole' file at startup.

'--vanilla'

윈도우 내의 '-no-save', '-no-envron', '-no-site-file', '-no-init-file' 그리고 '-no-restore'를 병합하고 이는 '-no-Rconsole'를 포함합니다.

Combine '-no-save', '-no-envron', '-no-site-file', '-no-init-file' and '-no-restore'. Under Windows, this also includes '-no-Rconsole'.

'-f file'

'--file=file'

파일에서 input을 take함: '-'는 stdin을 의미합니다. '-save'가 설정되지 않은 이상 '-no-save'를 뜻합니다.

Take input from file: '-' means stdin. Implies '-no-save' unless '-save' has been set.

'-e expression'

Input line처럼 expression을 사용하세요. 한 개 이상의 '-e' 옵션이 사용될 수 있으나, '-f' 나 '-file'과 같이 사용하는 사용할 수 없습니다. '-save'가 설정되지 않은 이상 '-no-save'을 뜻합니다. (이

방법으로 씌여지는 expressions의 총 길이는 10,000 바이트의 제한이 있습니다.)

Use expression as an input line. One or more '-e' options can be used, but not together with '-f' or '-file'. Implies '-no-save' unless '-save' has been set. (There is a limit of 10,000 bytes on the total length of expressions used in this way.)

'--no-readline'

(UNIX에서만 사용가능) readline을 통해 command-line을 고치는 것을 끄. 이는 ESS ("Emacs Speaks Statistics") 패키지를 이용하여 Emacs안에서 R을 가동할 때 유용합니다. 더 자세한 정보는 페이지 87쪽의 Appendix C [The command-line editor]를 참조하세요.

(UNIX only) Turn off command-line editing via readline. This is useful when running R from within Emacs using the ESS ("Emacs Speaks Statistics") package. See Appendix C [The command-line editor], page 87, for more information.

'--ess'

(Windows에서만 사용가능) ESS에서 R-inferior-mode를 사용함으로써 Rterm up을 설정.

(Windows only) Set Rterm up for use by R-inferior-mode in ESS.

'--min-vsize=N'

'--max-vsize=N'

여러가지 사이즈의 objects들의 메모리의 최소값이나 최대값을 "vector heap" 사이즈를 N바이트로 바꾸는 설정을 하여 확정하세요. 여기서, N은 정수이거나 'Giga' ( $2^{30}$ ), 'Mega' ( $2^{20}$ ), (computer) 'Kilo' ( $2^{10}$ ), regular 'kilo' (1000)를 뜻하는 'G', 'M', 'K', 'k'로 끝나는 정수들이어야만 합니다.

Specify the minimum or maximum amount of memory used for variable size objects by setting the "vector heap" size to N bytes. Here, N must either be an integer or an integer ending with 'G', 'M', 'K', or 'k', meaning 'Giga' ( $2^{30}$ ), 'Mega' ( $2^{20}$ ), (computer) 'Kilo' ( $2^{10}$ ), or regular 'kilo' (1000).

'--min-nspace=N'

'--max-nspace=N'

"cons cells"의 개수를 N으로 설정함으로써 고정된 크기의 objects의 memory 값을 확정. 좀 더 자세한 사항을 위해서는 N에 관한 이전의 옵션을 보세요. Cons cell은 21-bit 기계에서는 28 바이트를 차지하고, 64-bit 기계에서는 보통 56 바이트를 차지합니다.

Specify the amount of memory used for fixed size objects by setting the number of "cons cells" to N. See the previous option for details on N. A cons cell takes 28 bytes on a 32-bit machine, and usually 56 bytes on a 64-bit machine.

'--max-ppsize=N'

N locations에 따라 Pointer protection stack의 사이즈를 최대로 확정. 이는 10,000으로 default이지만 크고 복잡한 계산을 하기위해서는 늘릴 수도 있습니다. 현재로써 허용되는 최대값은 100,000입니다.

Specify the maximum size of the pointer protection stack as N locations. This defaults to 10000, but can be increased to allow large and complicated calculations to be done. Currently the maximum value accepted is 100000.

'--max-mem-size=N'



(Windows에서만 사용 가능) R objects와 working areas를 위해 사용될 memory의 양을 제한. 이는 1.5Gb의 작은..

(Windows only) Specify a limit for the amount of memory to be used both for R objects and working areas. This is set by default to the smaller of 1.5Gb and the amount of physical RAM in the machine, and must be between 32Mb and the maximum allowed on that version of Windows.

'--quiet'

'--silent'

'-q'

initial copyright과 welcome messages는 프린트하지마세요.

Do not print out the initial copyright and welcome messages.

'--slave'

R을 최대한 조용히 만듦. 이 옵션은 결과를 산출하기 위해 R을 사용하는 support programs를 위한 것 입니다. 이는 '-quiet'과 '-no-save'을 뜻합니다.

Make R run as quietly as possible. This option is intended to support programs which use R to compute results for them. It implies '-quiet' and '-no-save'.

'--interactive'

(UNIX만 사용 가능) Assert that R really is being run interactively even if input has been redirected: 만약 input이 FIFO나 pipe에서 나오고 interactive program에서 fed이면 사용하세요.

(UNIX only) Assert that R really is being run interactively even if input has been redirected: use if input is from a FIFO or pipe and fed from an interactive program.

'--verbose'

프로그램에 대해 더 많은 정보를 프린트하고 특히 R의 옵션 verbose를 TRUE로 설정 합니다. R 코드는 이 옵션을 진단 메시지의 프린트를 컨트롤 하기위해 사용합니다.

Print more information about progress, and in particular set R's option verbose to TRUE. R code uses this option to control the printing of diagnostic messages.

'--debugger=name'

'-d name'

(UNIX에서만 사용가능) R을 debugger name으로 가동. 대부분의 debuggers의 경우(예외는 valgrind와 최신 버전의 gdb입니다), 한층 더해진 command line 옵션들은 무시되고, debugger안에서부터 R executable이 시작될 때 대신 주어져야 합니다.

(UNIX only) Run R through debugger name. For most debuggers (the exceptions are valgrind and recent versions of gdb), further command line options are disregarded, and should instead be given when starting the R executable from inside the debugger.

'--gui=type'

'-g type'

(UNIX에서만 사용가능) type을 graphical 사용자 interface로 사용(이는 interactive graphics도 포함한다는 것을 유의하세요). 현재, type의 가능한 values는 'X11'(디폴트)와 'Tcl/Tk'가 지원된다는 전제 하에서는 'Tk'입니다. (back-compatibility를 위해서는 'x11' 와 'tk'

가 허용됩니다.)

(UNIX only) Use type as graphical user interface (note that this also includes interactive graphics). Currently, possible values for type are 'X11' (the default) and, provided that 'Tcl/Tk' support is available, 'Tk'. (For back-compatibility, 'x11' and 'tk' are accepted.)

'--args'

이 flag는 나머지 command line들을 스킵하게 하는 것 빼고는 아무것도 하는게 없습니다: 이는 commandArgs(TRUE)로 flag에서 values를 복구하는데 유용할 수 있습니다.

Input과 output은 ('<' 과 '>'를 사용하여) 보통 방법으로 다시 돌릴 수 있으나, 4095 바이트의 line 길이 제한은 계속 적용됩니다. 경고와 에러 메시지는 error channel(stderr)로 보내집니다.

This flag does nothing except cause the rest of the command line to be skipped: this can be useful to retrieve values from it with commandArgs(TRUE).

Note that input and output can be redirected in the usual way (using '<' and '>'), but the line length limit of 4095 bytes still applies. Warning and error messages are sent to the error channel (stderr).

명령 R CMD는 R의 conjunction에 유용한 여러가지 tools의 invocation을 허용하지만, 일부러 "직접적으로" 불려지지는 않습니다. 일반적인 형태는

The command R CMD allows the invocation of various tools which are useful in conjunction with R, but not intended to be called "directly". The general form is

R CMD command args

명령이 tool의 이름인 곳과 args the arguments가 그로 전해진 곳입니다. 현재로써는 다음과 같은 tools가 사용 가능합니다.

where command is the name of the tool and args the arguments passed on to it. Currently, the following tools are available.

BATCH Run R in batch mode. (batch mode에서 작동)

COMPILE (UNIX only) Compile files for use with R.  
((UNIX에서만 사용가능) R을 사용하기 위해 파일들을 모음.)

SHLIB Build shared library for dynamic loading.  
(dynamic loading을 위해 공유 library를 만듦.)

INSTALL Install add-on packages. (부가적인 패키지들을 설치)

REMOVE Remove add-on packages. (부가적인 패키지들을 지움)

build Build (that is, package) add-on packages. (부가적인 패키지들을 만듦.)

check Check add-on packages. (부가적인 패키지들을 체크함)

LINK (UNIX only) Front-end for creating executable programs.  
((UNIX에서만 사용가능) executable 프로그램을 만들기위한 기초준비)

Rprof Post-process R profiling files. (R profiling 파일들을 미리 수행)

Rdconv Convert Rd format to various other formats, including HTML, Nroff, LATEX, plain text, and S documentation format.

(HTML, Nroff, LATEX, plain text,와 S 문서 format을 포함한 Rd format을 다른 여러가지 format으로 바꿈)

Rd2dvi Convert Rd format to DVI/PDF.(Rd format을 DVI/PDF로 바꿈)

Rd2txt Convert Rd format to text.(Rd format을 텍스트로 바꿈)

Sd2Rd Convert S documentation to Rd format. (S 문서를 Rd format으로 바꿈)

config Obtain configuration information. (configuration정보를 얻음)

Use

R CMD command -help

각각의 tools accessible의 사용 정보를 R CMD interface를 통해 얻기위해 사용.

to obtain usage information for each of the tools accessible via the R CMD interface.

### Node 143. Invoking R under Windows

Window에서 R을 실행하는 두가지 방법이 있습니다. Terminal window(예를들어, cmd.exe 나 command.com나 더 가능한 shell)안에서, 전 섹션에서 설명되었던 방법들이 R.exe나 더 직접적으로는 Rterm.exe에 의해 불러져 사용될지도 모릅니다. (이들은 원래 batch use를 위한 것들입니다.) interactive 사용을 위해서는, console-based GUI (Rgui.exe)가 있습니다. Windows에서의 startup 순서는 UNIX와 매우 비슷하지만, 'home directory'로의 references가 Windows에서는 항상 명확하지는 않기 때문에 명확히 나타내 주어야 합니다. 만약 environment variable R\_USER이 정의되었다면, 이는 home directory를 산출합니다. 다음으로, 만약 environment variable HOME이 정의되었다면, 이도 home directory를 산출합니다. 이 두가지의 사용자가 조정 가능한 설정들 후에, R은 home directory가 정의된 system을 찾으려고 할 것입니다. 이는 첫번째로 Windows의 "개인적인" directory를 사용하려고 시도 할 것입니다(보통 Windows XP의 C:Documents와 SettingsusernameMy Documents). 만약 이것이 실패하고 environment variables HOMEDRIVE와 HOMEPATH가 정의되어 있으면(보통 정의되어 있습니다.), 이는 home directory를 정의 할 것입니다. 이 모든 것이 실패한다면 home directory는 starting directory가 됩니다. variables TMPDIR, TMP와 TEMP들중 하나가 설정해제 되어있던지 아니면 이들중 하나가 temporary files와 directories를 만들기위한 적합한 장소를 가리키고 있는지 확인해야만 합니다. Environment variable는 command line에 'name=value' pair로 제공될 수 있습니다. 만일 '.RData'로 끝나는 데이터가 있다면(어떠한 경우라도) 이는 workspare가 복구되도록하는 path로 바뀔 것입니다: 이는 '-restore'를 의미하고 working kirectory를 parent of the named file로 설정합니다. (이 작용은 끌어서 당기기와 RGui.exe에 결합된 파일에 사용되지만, Rterm.exe에도 사용될 수 있습니다. 만약 named file이 존재하지 않는다면 parent directory가 존재할 경우 working directory를 설정합니다.) invoking RGui.exe일 때 다음의 부가적인 command-line옵션이 사용 가능 합니다.

There are two ways to run R under Windows. Within a terminal window (e.g. cmd.exe or command.com or a more capable shell), the methods described in the

previous section may be used, invoking by R.exe or more directly by Rterm.exe. (These are principally intended for batch use.) For interactive use, there is a console-based GUI (Rgui.exe). The startup procedure under Windows is very similar to that under UNIX, but references to the 'home directory' need to be clarified, as this is not always defined on Windows. If the environment variable R\_USER is defined, that gives the home directory. Next, if the environment variable HOME is defined, that gives the home directory. After those two user-controllable settings, R tries to find system defined home directories. It first tries to use the Windows "personal" directory (typically C:\Documents and Settings\username\My Documents in Windows XP). If that fails, and environment variables HOMEDRIVE and HOMEPATH are defined (and they normally are) these define the home directory. Failing all those, the home directory is taken to be the starting directory. You need to ensure that either the environment variables TMPDIR, TMP and TEMP are either unset or one of them points to a valid place to create temporary files and directories. Environment variables can be supplied as 'name=value' pairs on the command line. If there is an argument ending '.RData' (in any case) it is interpreted as the path to the workspace to be restored: it implies '-restore' and sets the working directory to the parent of the named file. (This mechanism is used for drag-and-drop and file association with RGui.exe, but also works for Rterm.exe. If the named file does not exist it sets the working directory if the parent directory exists.) The following additional command-line options are available when invoking RGui.exe.

'--mdi'

'--sdi'

'--no-mdi'

Rgui가 MDI 프로그램(한 개의 주 window안에 여러 개의 child windows가 있는것)처럼 작동할 지 아니면 SDI 어플리케이션(console, graphics와 pager o 르 위한 여러 개의 top-level windows)처럼 작동할지 조정. Command-line 설정은 사용자의 'Rconsole' 파일에 덮어씌웁니다.

Control whether Rgui will operate as an MDI program (with multiple child windows within one main window) or an SDI application (with multiple top-level windows for the console, graphics and pager). The command-line setting overrides the setting in the user's 'Rconsole' file.

'--debug'

Rgui에서 "Break to debugger" 메뉴 아이템을 사용가능하게 하고, command line이 진행되고 있는 동안 debugger에 break를 유발.

Enable the "Break to debugger" menu item in Rgui, and trigger a break to the debugger during command line processing.

R CMD를 가지고있는 Windows에서, 원래있는 commands들중 한 개를 대신해서 사용자의 '\*.bat' 나 '\*.exe'를 특징짓는 것도 좋은 생각일 수 있습니다. 이는 R\_HOME, R\_VERSION, R\_CMD, R\_OSTYPE, PATH, PERL5LIB, 그리고 TEXINPUTS를 포함한 똑바로 설정된 여러 개

의 environment variables과 함께 작동할 것입니다. 예를들어, 만약 path에 'latex.exe'를 이미 가지고 있다면,

In Windows with R CMD you may also specify your own '\*.bat' or '\*.exe' file instead of one of the built-in commands. It will be run with several environment variables set appropriately, including R\_HOME, R\_VERSION, R\_CMD, R\_OSTYPE, PATH, PERL5LIB, and TEXINPUTS. For example, if you already have 'latex.exe' on your path, then

R CMD latex.exe mydoc

TEXINPUTS에 추가된 R의 'share/texmf'의 path와 함께 'mydoc.tex'위의 LATEX를 작동할 것입니다.

will run LATEX on 'mydoc.tex', with the path to R's 'share/texmf' macros appended to TEXINPUTS.

#### Node 144. Invoking R under Mac OS X

Mac OS X서 R을 작동하는 두가지의 방법이 있습니다. R을 invoking함으로 인한 Terminal.app window안에서, 전의 섹션에서 설명한 방법들이 적용됩니다. Default에 의한 console-based GUI (R.app)도있는데 이는 시스템의 applications 폴더에 설치되어있습니다. 이는 standard doubleclickable Mac OS X application입니다. Mac OS X의 startup 순서는 UNIX에서와 비슷합니다. 'home directory'는 R.framework안에 있는것이지만, startup과 현재의 working directory는 GUI안에서 Preferences window accessible안에 다른 startup directory가 주어지지 않은 이상 사용자의 home directory로 설정됩니다.

There are two ways to run R under Mac OS X. Within a Terminal.app window by invoking R, the methods described in the previous sections apply. There is also console-based GUI (R.app) that by default is installed in the Applications folder on your system. It is a standard doubleclickable Mac OS X application. The startup procedure under Mac OS X is very similar to that under UNIX. The 'home directory' is the one inside the R.framework, but the startup and current working directory are set as the user's home directory unless a different startup directory is given in the Preferences window accessible from within the GUI.

#### Node 145. Scripting in R

만약 R 명령어의 'foo.R' 파일을 작동하고 싶다면, 추천하는 방법은 R CMD BATCH foo.R을 사용하는 것입니다. 만약 이것을 background나 batch job처럼 작동하고싶다면 OS-specific facilities를 사용하세요: 예를들어 대부분의 shells는 Unix-alike OSes R CMD BATCH foo.R & background job을 실행합니다. Command line위에있는 부가적인 arguments를 통해 Parameters를 scrips로 넘길 수도 있습니다: 예를들어

If you just want to run a file 'foo.R' of R commands, the recommended way is to use R CMD BATCH foo.R. If you want to run this in the background or as a batch job use OS-specific facilities to do so: for example in most shells on Unix-alike OSes R CMD BATCH foo.R & runs a background job. You can pass parameters to scripts via additional arguments on the command line: for example

R CMD BATCH --args arg1 arg2 foo.R &

는 arguments를 script로 보냅니다. 이는 다음의 것에 의해 character vector처럼 복귀될 수 있습니다.

```
will pass arguments to a script which can be retrieved as a character vector by  
args <- commandArgs(TRUE)
```

이는 다음의 것에 의해 invoked할 수 있는 다른 front-end Rscript에 의해 더 간단하게 만들어지고, This is made simpler by the alternative front-end Rscript, which can be invoked by Rscript foo.R arg1 arg2

이는 또한 다음과 같은 (적어도 Unix같은, 그리고 몇몇의 Window shells) executable script 파일들을 쓸 수도 있는데,

and this can also be used to write executable script files like (at least on Unix-alikes, and in some Windows shells)

```
#!/path/to/Rscript  
args <- commandArgs(TRUE)
```

...

```
q(status=<exit status code>)
```

마약 이것이 'runfoo' 텍스트 파일로 입력되고 (chmod 755 runfoo에 의해) executable로 만들어 졌다면, 이는 다음의 것에 의해 다른 arguments에 invoked할 수 있습니다.

If this is entered into a text file 'runfoo' and this is made executable (by chmod 755 runfoo), it can be invoked for different arguments by

```
runfoo arg1 arg2
```

더 많은 옵션들에 대해서는 help("Rscript")를 참조하세요. 이는 R output을 'stdout'과 'stderr'로 쓰고 command를 실행하는 shell을 보통 방법으로 되돌릴 수도 있습니다. 만약 Rscript를 hard code the path로 하지않고 당신의 path에 가지고싶다면(보통 Windows를 제외한 R 설치하는 경우),

For further options see help("Rscript"). This writes R output to 'stdout' and 'stderr', and this can be redirected in the usual way for the shell running the command. If you do not wish to hardcode the path to Rscript but have it in your path (which is normally the case for an installed R except on Windows), use

```
#!/usr/bin/env Rscript
```

...

를 사용하세요.

적어도 Bourne과 bash shells에서는, #! Mechanism은 #! /usr/bin/env Rscript -vanilla 과 같은 여분의 arguments를 허용하지 않습니다. 한가지 생각해야 할 것은 stdin()이 무엇을 참고하느냐 입니다. 이는 다음과 같은 segments들을 R scripts를 쓰는 보통 장소이고,

At least in Bourne and bash shells, the #! mechanism does not allow extra arguments like #! /usr/bin/env Rscript -vanilla. One thing to consider is what stdin() refers to. It is commonplace to write R scripts with segments like

```
chem <- scan(n=24)
```

```
2.90 3.10 3.40 3.40 3.70 3.70 2.80 2.50 2.40 2.40 2.70 2.20
```

5.28 3.37 3.03 3.03 28.95 3.77 3.40 2.20 3.50 3.60 3.70 3.70

`stdin()`은 traditional usage를 허용하기 위한 script 파일을 뜻합니다. 만약 프로세스의 `'stdin'`을 참조하고 싶다면, `"stdin"`을 파일 connection처럼 사용하세요, 예를들어, `scan("stdin", ...)`. executable script 파일들(François Pinard에 의해 제안된)을 쓰는 다른 방법은 다음과 같은 here document를 쓰는 것입니다.

and `stdin()` refers to the script file to allow such traditional usage. If you want to refer to the process's `'stdin'`, use `"stdin"` as a file connection, e.g. `scan("stdin", ...)`. Another way to write executable script files (suggested by François Pinard) is to use a here document like

```
#!/bin/sh
```

```
[environment variables can be set here]
```

```
R --slave [other options] <<EOF
```

```
R program goes here...
```

```
EOF
```

하지만 here `stdin()`은 프로그램 source를 참조하고 `"stdin"`은 사용할 수 없을 것입니다. 아주 짧은 scripts는 `'-e'` flag를 통해 command-line에 있는 R script로 보내질 수 있습니다.

but here `stdin()` refers to the program source and `"stdin"` will not be usable. Very short scripts can be passed to `Rscript` on the command-line via the `'-e'` flag.

## Node 147. Preliminaries

R이 UNIX 안에서 편집을 위해 배열될 때 GNU readline library가 사용가능 할 때, inbuilt command line editor는 전 commands에 사용된 recall, editing, 그리고 resubmission을 허용합니다. 다른 버전의 readline이 존재하고 inbuilt command line editor에 의해 사용될 수 있다는 것을 유의하세요: 이는 Mac OS X에서 발생할 수 있습니다. 이것은 `'-no-readline'`이라는 startup 옵션을 사용해 기능을 억제할 수도 있습니다(ESS1의 사용에 유용합니다). Windows 버전의 R은 어느정도 더 간단한 command-line editing을 가지고 있습니다: GUI의 `'Help'` 메뉴에 있는 `'Console'`과 `Rterm.exe`에 있는 command-line editing에 대한 `'README.Rterm'` 파일을 참조하세요. Readline을 할 수 있는 R을 사용할 때, 다음에 설명된 functions들이 사용 가능합니다. 이들 많은 것들이 Control이나 Meta characters를 사용합니다. Control-m과 같은 control characters는 M키를 누른채로 CTRL을 누르면 얻어지고 C-m below라고 씁니다. Meta-b와 같은 Meta characters는 META2와 B를 누르면 M-b로 씌여집니다. 만약 당신의 terminal이 META key를 가지고 있지 않으면, ESC로 시작하는 두개의 character sequences를 사용하여 Meta characters를 type할 수 있습니다. 그러므로, M-b를 쓰기 위해서는, ESCB를 타입할 수 있습니다. ESC character sequences는 진짜 Meta keys와 함께 terminals에서 허용됩니다. 이 경우는 Meta characters의 경우에 매우 의미를 가집니다.

When the GNU readline library is available at the time R is configured for compilation under UNIX, an inbuilt command line editor allowing recall, editing and re-submission of prior commands is used. Note that other versions of readline exist and may be used by the inbuilt command line editor: this may happen on Mac OS X. It can be disabled (useful for usage with ESS1) using the startup option `'-no-readline'`.



Windows versions of R have somewhat simpler command-line editing: see 'Console' under the 'Help' menu of the GUI, and the file 'README.Rterm' for command-line editing under Rterm.exe. When using R with readline capabilities, the functions described below are available. Many of these use either Control or Meta characters. Control characters, such as Control-m, are obtained by holding the CTRL down while you press the M key, and are written as C-m below. Meta characters, such as Meta-b, are typed by holding down META2 and pressing B, and written as M-b in the following. If your terminal does not have a META key, you can still type Meta characters using two-character sequences starting with ESC. Thus, to enter M-b, you could type ESCB. The ESC character sequences are also allowed on terminals with real Meta keys. Note that case is significant for Meta characters.

#### Node 148. Editing actions

R 프로그램은 당신이 입력한 erroneous line을 포함한 command line의 히스토리를 간직하고, 그 히스토리에 있는 commands는 다시 부를수도 있고, 필요하다면 바꿀 수도 있으며, 새로운 commands처럼 다시 제출하기도 합니다. 이 고치는 단계에서 사용자가 하는 Emacs-style command-line editing에서의 아무 straight typing은 당신이 고치고있는 characters가 command안에 삽입되는 원인이되고 커서 오른쪽의 아무 characters의 위치를 바꿉니다. Vi mode character insertion mode에서는 M-i나 M-a에 의해 시작되는데, characters는 입력되고 insertion mode는 ESC를 입력함으로써 끝납니다. 어떤 상황에서는 RET command를 누르면 다시 제출하는 명령을 내리게 됩니다. 다른 editing actions는 다음 테이블에 요약되어 있습니다.

The R program keeps a history of the command lines you type, including the erroneous lines, and commands in your history may be recalled, changed if necessary, and re-submitted as new commands. In Emacs-style command-line editing any straight typing you do while in this editing phase causes the characters to be inserted in the command you are editing, displacing any characters to the right of the cursor. In vi mode character insertion mode is started by M-i or M-a, characters are typed and insertion mode is finished by typing a further ESC. Pressing the RET command at any time causes the command to be re-submitted. Other editing actions are summarized in the following table.

#### Node 150. **Command recall and vertical motion**

C-p Go to the previous command (backwards in the history).

[전 명령으로 가세요(히스토리에서 뒤쪽으로)]

C-n Go to the next command (forwards in the history).

[다음 명령으로 가세요(히스토리에서 앞쪽으로)]

C-r text Find the last command with the text string in it.

대부분의 터미널에서, 순서대로 C-p와 C-n 대신 위쪽과 아래쪽 화살표를 쓸 수도 있습니다.

On most terminals, you can also use the up and down arrow keys instead of C-p and C-n, respectively.

#### Node 151. **Horizontal motion of the cursor**

C-a Go to the beginning of the command.

[명령의 처음부분으로 감]

C-e Go to the end of the line.

[줄의 끝으로 감]

M-b Go back one word.

[한 글자 뒤로 감]

M-f Go forward one word.

[한 글자 앞으로 감]

C-b Go back one character.

[한 character 뒤로 감]

C-f Go forward one character.

[한 character 앞으로 감]

대부분의 terminals에서, 순서대로 C-b와 C-f 대신에 왼쪽과 오른쪽 화살표 키를 사용 할 수도 있습니다.

On most terminals, you can also use the left and right arrow keys instead of C-b and C-f, respectively.

#### Node 152. **Editing and re-submission**

text Insert text at the cursor.

[커서가 있는 곳에 텍스트를 삽입]

C-f text Append text after the cursor.

[커서 다음에 글자 Append를 씀]

DEL Delete the previous character (left of the cursor).

[(커서 왼쪽에 있는) 전 character를 지움]

C-d Delete the character under the cursor.

[커서 아래있는 character를 지움]

M-d Delete the rest of the word under the cursor, and "save" it.

[커서 아래에 있는 나머지 단어들을 지우고, 저장]

C-k Delete from cursor to end of command, and "save" it.

[커서에서부터 command의 끝까지 지우고, 저장]

C-y Insert (yank) the last "saved" text here.

[여기에 (yank)를 삽입하고 "saved"라고 씀]

C-t Transpose the character under the cursor with the next.

[다음과 함께 커서 아래있는 character를 transpose]

M-l Change the rest of the word to lower case.

[나머지 단어들을 소문자로 바꿈]

M-c Change the rest of the word to upper case.

[나머지 단어들을 대문자로 바꿈]

RET Re-submit the command to R.

[R에 command를 다시 제출]

마지막 RET는 command line editing sequence를 종료합니다.

The final RET terminates the command line editing sequence