02 Data Structures and Data Types

Vectors

- A vector is an ordered collection of objects of the same type
- The function `c(...)` concatenates arguments to form vector
- To create a patterned vector
  - `:` : Sequence integers
  - `seq()` : General sequence
  - `rep()` : Vector of replicated elements

```
> v1 <- c(2.5, 4, 7.3, 0.1)
> v1
[1] 2.5 4.0 7.3 0.1
> v2 <- c("A", "B", "C", "D")
> v2
[1] "A" "B" "C" "D"
> v3 <- -3:3
> v3
[1] -3 -2 -1 0 1 2 3
```

```
> seq(0, 5)
[1] 0 1 2 3 4 5
> seq(0, 5, by=0.5)
[1] 0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
> seq(0, 5, length.out=5)
[1] 0.00 1.25 2.50 3.75 5.00
> seq(0, 5, length.out=6)
[1] 0 1 2 3 4 5
> rep(1, 15)
[1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
> rep(1:5, 3)
[1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
> rep(1:5, times=3)
[1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
> rep(1:5, each=3)
[1] 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5
```

Vectors

- R is a vectorized language
- A vector is the most basic datatype for data storage in R.
- It stores one or more values of the same type.

```
> x <- 10:20
> x <- seq(10, 20)
> x <- c(10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20)
```

- Compute the sum of natural numbers from 1 to 1000

```
> x <- 1:10^3
> sum(x)
```

- Compute the sum of odd numbers form 1 to 1000

```
> x <- seq(1, 10^3, 2)
> sum(x)
```

Exercise

- Compute the following values with $N = 100$

1. $\displaystyle\sum_{k=1}^{N} \frac{1}{k}$

2. $\displaystyle\sum_{n=0}^{N} 2^{-n}$

3. $\displaystyle\sum_{n=0}^{N} \frac{1}{n!}$

4. $\displaystyle\sum_{n=0}^{N} \frac{(-1)^n}{(2n+1)!} \left(\frac{\pi}{3}\right)^{2n+1}$

Reference Elements of a Vector

- Brackets [ ] are used for subsetting, returns a vector with potentially a subset of elements
- Use a minus sign to remove elements

```
> x <- c(4, 7, 2, 10, 1, 0)
> x[4]
[1] 10
> x[1:3]
[1] 4 7 2
> x[c(2,5,6)]
[1] 7 1 0
> x[-3]
[1] 4 7 10 1 0
```

```
> x[-c(4,5)]
[1] 4 7 2 0
> x[x > 4]
[1] 7 10
> x[x < 4]
[1] 2 1 0
> x[3] <- 99
> x
[1] 4 7 99 10 1 0
```

```
> x <- 22

# x is a vector of length 1
> x

# single subsetting still returns a vector of length 1
> x[1]

# double subsetting also returns a vector of length 1
> x[1][1]

# it never ends....
> x[1][1][1]
```

Location Functions of Vector

- Additional functions that will return the indices of a vector.
  - `which()` : Indices of a logical vector where the condition is `TRUE`
  - `which.max()` : Location of the (first) maximum element of a vector
  - `which.min()` : Location of the (first) minimum element of a vector
  - `match()` : First position of an element in a vector

```
> x <- c(4, 7, 2, 10, 1)
> x >= 4
[1] TRUE TRUE FALSE TRUE FALSE
> which(x >= 4)
[1] 1 2 4
```

```
> which.max(x)
[1] 4
> which(x==max(x))
[1] 4
> x[which.max(x)]
[1] 10
> max(x)
[1] 10
> y <- rep(1:5, times=5:1)
> y
[1] 1 1 1 1 1 2 2 2 2 3 3 3 4 4 5
> match(1:5, y)
[1] 1 6 10 13 15
> unique(y)
[1] 1 2 3 4 5
> match(unique(y), y)
[1] 1 6 10 13 15
```

Vector Operations

- When vectors are used in math expressions, the operations are performed element by element.

```
> x <- c(4,7,2,10,1,0)
> y <- x^2 + 1
> y
[1] 17 50 5 101 2 1
> x*y
[1] 68 350 10 1010 2 0
> round(x/y, 5)
[1] 0.23529 0.14000 0.40000 0.09901 0.50000
0.00000
> round(x/sum(y), 5)
[1] 0.02273 0.03977 0.01136 0.05682 0.00568
0.00000
> sum(x)/max(y)
[1] 0.2376238
```

Useful Vector Functions

`sum(x)`	`prod(x)`	Sum/prod of the elements of `x`
`cumsum(x)`	`cumprod(x)`	Cumulative sum/prod of the elements of `x`
`mean(x)`	`median(x)`	Mean/median of `x`
`var(x)`	`sd(x)`	Variance/standard deviation of `x`
`cov(x,y)`	`cor(x,y)`	Covariance/correlation of `x`
`range(x)`		Range of `x`
`quantile(x)`		Quantiles of `x` for the given proabilities
`length(x)`		Numbers of elements in `x`
`unique(x)`		Unique elements of `x`
`rev(x)`		Reverse the the elements of `x`
`sort(x)`		Sort the the elements of `x`
`union(x,y)`		Union of `x` and `y`
`intersect(x,y)`		Intersection of `x` and `y`
`setdiff(x,y)`		Elements of `x` that are not in `y`

Exercise

- Suppose that we have

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2},$$

where x starts from -4 to 4 increased by 0.01.

1. Compute the following statistics of $f(x)$.

   min, max, mean, median, variance and standard deviation

2. Which x can maximize or minimize $f(x)$?
3. Scale $f(x)$ such that

$$f'(x) = \frac{f(x)}{\sum_{i=1}^{n} f(x)}$$

   Compute the cumulative sum of $\sum_{x \leq c} f'(x)$, for $c = 1.64$ and 1.96

Matrices

- A matrix is just a two-dimensional generalization of a vector
- To create a matrix,
  `matrix(data=NA, nrow=1, ncol=1, byrow = FALSE, dimnames = NULL)`
  - `data` : a vector that gives data to fill the matrix; if data does not have enough elements to fill the matrix, then the elements are recycled.
  - `nrow` : desired number of rows
  - `ncol` : desired number of columns
  - `byrow` : if FALSE (default) matrix is filled by columns, otherwise by rows
  - `dimnames` : (optional) list of length 2 giving the row and column names respectively, list names will be used as names for the dimensions
- Reference matrix elements using the `[ ]` just like with vectors, but now with 2-dimensions

Creating Matrices

- A matrix can be created as follows

```
> matrix(1:6, nrow=3, ncol=2)
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

- An optional argument `byrow=TRUE` can be used to arrange the vector of values by row.

```
> matrix(1:6, nrow=3, ncol=2, byrow=TRUE)
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
```

Matrix Elements and Recycling

- The first argument to the matrix function contains the elements which are to form the matrix. If there are not enough elements in the argument to create the matrix, the recycling rule is applied to obtain more.
- A 2×3 matrix filled with 1 and 2 can be obtained as follows

```
> matrix(1:2, nrow=3, ncol=4)
     [,1] [,2] [,3] [,4]
[1,]    1    2    1    2
[2,]    2    1    2    1
[3,]    1    2    1    2
```

Optional Dimension Specifications

- Often R can work out the number of rows in a matrix given the number of columns and the elements, or the the number of columns in a matrix given the number of rows and the elements. In such cases it is not necessary to specify both the number of rows and the number of columns.

```
> matrix(1:15, nrow=5)
     [,1] [,2] [,3]
[1,]    1    6   11
[2,]    2    7   12
[3,]    3    8   13
[4,]    4    9   14
[5,]    5   10   15
```

Determining Matrix Dimensions

- The number of rows and columns or a matrix can be obtained with the functions `nrow` and `ncol`. Or obtained together with the function `dim` which returns a vector containing the number of rows as the first element can the number of columns as its second.

```
> x <- matrix(1:30, 5, 6)
> nrow(x)
[1] 5
> ncol(x)
[1] 6
> dim(x)
[1] 5 6
```

Creating Matrices from Rows and Columns

- Matrices can be created by gluing together rows with `rbind` of gluing together columns with `cbind`.

```
> cbind(1:3, 4:6)
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> rbind(1:3, 4:6)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

Binding Rows and Columns; Recycling

- The arguments to `rbind` and `cbind` are not required to be the same length. When they are not, a matrix is created which is big enough to accommodate the largest argument, and the others have the recycling rule applied to them to supply additional arguments. Apparent mismatches produce warnings.

```
> rbind(1:2, 1:3)
     [,1] [,2] [,3]
[1,]    1    2    1
[2,]    1    2    3
Warning message:
In rbind(1:2, 1:3) :
  number of columns of result is not a multiple of
  vector length (arg 1)
```

Matrices and Naming

- It is possible to attach row and column labels to matrices. Row names can be attached with `rownames`, column names with `colnames`, and both can be attached simultaneously with `dimnames`.

```
> x <- matrix(1:6, nrow=2)
> dimnames(x) <- list(c("First", "Second"),
                      c("A", "B", "C"))
> x
```

```
       A B C
First  1 3 5
Second 2 4 6
```

Extracting Names

- Names can also be extracted with `dimnames`, `rownames` and `colnames`.

```
> dimnames(x)
[[1]]
[1] "First"  "Second"

[[2]]
[1] "A" "B" "C"
> rownames(x)
[1] "First" "Second"
> colnames(x)
[1] "A" "B" "C"
```

Matrix Subsets

- The ij-th element of a matrix `x` can be extracted with the expression `x[i, j]`. It is used to extract more general subsets of the elements of a matrix by specifying vector subscripts.

```
> x <- matrix(1:12, nrow=3, ncol=4)
> x
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> x[3,1]
[1] 3
> x[1:2, c(2, 4)]
     [,1] [,2]
[1,]    4   10
[2,]    5   11
```

Assigning to Matrix Subsets

- It is also possible to assign to subsets of matrices.

```
> x[1:2, c(2, 4)] <- 21:24
> x
     [,1] [,2] [,3] [,4]
[1,]    1   21    7   23
[2,]    2   22    8   24
[3,]    3    6    9   12
> x[2:1, c(2, 4)] <- 21:24
> x
     [,1] [,2] [,3] [,4]
[1,]    1   22    7   24
[2,]    2   21    8   23
[3,]    3    6    9   12
```

Specifying Entire Rows and Columns

- When a subscript is omitted, it is taken to correspond to all possible values. This works for extracting value from matrices and for assigning to them.

```
> x <- matrix(1:12, nrow=3, ncol=4)
> x[1,] <- 100
> x
     [,1] [,2] [,3] [,4]
[1,]  100  100  100  100
[2,]    2    5    8   11
[3,]    3    6    9   12
```

Subsetting Matrices as Vectors

- Because matrices are just vectors with additional dimensioning information they can be treated as vectors.

```
> x <- matrix(1:6, nrow=2, ncol=3)
> x[7]
[1] NA
```

- The functions `row` and `col` return matrices indicating the row and column of each element. This can be used to extract or change submatrices.

```
> x[row(x) < col(x)] <- 0
> x
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    2    4    0
```

Tridiagonal Matrix

- Here is how to create a 4×4 tridiagonal matrix with diagonal values being 2 and the off-diagonal elements being 1.

```
> x <- matrix(0, nrow=4, ncol=4)
> x[row(x)==col(x)] <- 2
> x[abs(row(x)-col(x))==1] <- 1
> x
     [,1] [,2] [,3] [,4]
[1,]    2    1    0    0
[2,]    1    2    1    0
[3,]    0    1    2    1
[4,]    0    0    1    2
```

Reference Elements of Matrix

- We can reference parts of matrix by using row/column names
- Reference matrix elements using the [ ] but now use the column/row name, with quotations

```
> N <- matrix(c(5,8,3,0,4,1), 2, 3, byrow=TRUE)
> colnames(N) <- c("c.1", "c.2", "c.3")
> N
     c.1 c.2 c.3
[1,]   5   8   3
[2,]   0   4   1
> N[,"c.2"]
[1] 8 4
> N[,c("c.1", "c.3")]
     c.1 c.3
[1,]   5   3
[2,]   0   1
```

Matrix Operations

- When matrices are used in math expressions, the operations are performed element by element.
- For matrix multiplication use the `%*%` operator
- If a vector is used in matrix multiplication, it will be coerced to either a row or column matrix to make the arguments conformable. Using `%*%` on two vectors will return the inner product as a matrix and not a scalar. Use either `c()` or `as.vector()` to convert to a scalar.

```
> A <- matrix(1:4, nrow=2)
> B <- matrix(2, nrow=2, ncol=2)
> A * B
```

```
     [,1] [,2]
[1,]    2    6
[2,]    4    8
```

```
> A %*% B

     [,1] [,2]
[1,]    8    8
[2,]   12   12

> y <- 1:3
> y %*% y

     [,1]
[1,]   14

> A * (y %*% y)
Error in A * (y %*% y) :  non-conformable arrays
> A * c(y %*% y)

     [,1] [,2]
[1,]   14   42
[2,]   28   56
```

Combining Vectors and Matrices

- When a vector is added to a matrix, the recycling rule is used to expand the vector so that it matches the number of elements in the matrix.

```
> x <- matrix(1:4, nrow=2, ncol=2)
> x
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> x + 1
     [,1] [,2]
[1,]    2    4
[2,]    3    5
> x + 1:2
     [,1] [,2]
[1,]    2    4
[2,]    4    6
```

Combining Vectors and Matrices

- Some checks are carried out to try to make sure that operations are sensible. This can produce warnings.

```
> x + 1:3
     [,1] [,2]
[1,]    2    6
[2,]    4    5
Warning message:
In x + 1:3 :
  longer object length is not a multiple of
  shorter object length
```

- Note that it is an error to try to combine a matrix with a vector which has more elements than the matrix.

Useful Matrix Functions

`t(A)`	Transpose of `A`
`det(A)`	Determinant of `A`
`solve(A, b)`	Solves the equation of `Ax=b` for `x`
`solve(A)`	Matrix inverse of `A`
`eigen(A)`	Eigenvalues and eigenvectors of `A`
`chol(A)`	Choleski decomposition of `A`
`diag(n)`	Create a `n` $\times$ `n` identity matrix
`diag(A)`	Returns the diagonal elements of a matrix `A`
`diag(x)`	Create a diagonal matrix from a vector `x`
`lower.tri(x)`	Matrix of logicals indicating lower triangular matrix
`upper.tri(x)`	Matrix of logicals indicating upper triangular matrix
`rbind(...)`	Combines arguments by rows
`cbind(...)`	Combines arguments by columns
`dim(A)`	Dimensions of `A`
`nrow(A),ncol(A)`	Number of rows/columns of `A`

Exercise

- Suppose that we have

$$A = \begin{bmatrix} 1 & 0.2 & 0.5 \\ 0 & 1 & 0.3 \\ 0.5 & 0.7 & 1 \end{bmatrix} \qquad\qquad B = (0.6 \quad 0.5 \quad 0.3)^{\mathrm{T}}$$

Compute the followings:

1. cA and cB, where $c = \sqrt{2}$
2. $A^{\mathrm{T}}A$
3. AB and $B^{\mathrm{T}}A$
4. BB^{T} and $B^{\mathrm{T}}B$
5. $A^{-1}A$
6. $(A^{\mathrm{T}}A)^{-1}AB$

The Function `apply()`

- The `apply()` function is used for applying functions to the margins of a matrix, array, or dataframes.

```
apply(X, MARGIN, FUN, ...)
```

  - `X` : A matrix, array or dataframe
  - `MARGIN` : Vector of subscripts indicating which margins to apply the function to `1 = rows`, `2 = columns`
  - `FUN` : Function to be applied
  - `...` : Optional arguments for FUN
- You can also use your own function (more on this later)

```
> x <- matrix(rep(1:4, 3), 3, 4)
> x
     [,1] [,2] [,3] [,4]
[1,]    1    4    3    2
[2,]    2    1    4    3
[3,]    3    2    1    4

> apply(x, 1, sum)
[1] 10 10 10
> apply(x, 2, mean)
[1] 2.000000 2.333333 2.666667 3.000000
> apply(x, 2, range)
     [,1] [,2] [,3] [,4]
[1,]    1    1    1    2
[2,]    3    4    4    4
```

```
> apply(x, 1, sort)

     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    2    2    2
[3,]    3    3    3
[4,]    4    4    4

> apply(x, 2, sort)

     [,1] [,2] [,3] [,4]
[1,]    1    1    1    2
[2,]    2    2    3    3
[3,]    3    4    4    4

> apply(x, 2, function(t) sum(t!=1))
[1] 2 2 2 3
```

Arrays

- An array is a multi-dimensional generalization of a vector
- To create an array,

  `array(data=NA, dim=length(data), dimnames=NULL)`

  - `data` : A vector that gives data to fill the array
  - `dim` : Dimension of the array, a vector of length one or more giving the maximum indices in each dimension
  - `dimnames` : Name of the dimensions, list with one component for each dimension.
- Values are entered by columns
- Like with vectors and matrices, when arrays are used in math expressions the operations are performed element by element.
- Also like vectors and matrices, the elements of an array must all be of the same type (numeric, character, logical, etc.)

```
> # Sample 2 × 3 × 2 array
> w <- array(1:12, dim=c(2,3,2), dimnames=
+ list(c("A","B"), c("X","Y","Z"), c("N","M")))
> w
 , , N

  X Y Z
A 1 3 5
B 2 4 6

, , M

  X  Y  Z
A 7  9 11
B 8 10 12
```

```
> w[2,3,1]   # Row 2, Column 3, Matrix 1
[1] 6
> w[,"Y",]   # Column named "Y"
  N  M
A 3  9
B 4 10

> w[1,,]   # Row 1
  N  M
X 1  7
Y 3  9
Z 5 11

> w[1:2,,"M"]   # Rows 1 and 2, Matrix "M"
  X  Y  Z
A 7  9 11
B 8 10 12
```

Function apply()

- For a 3-dimensional array there are now three margins to apply the function to: 1 = `rows`, 2 = `columns`, and 3 = `matrices`.

```
> apply(w, 2, sum)    # Column sums
 X  Y  Z
18 26 34
> apply(w, 3, sum)    # Matrix sums
 N  M
21 57
> apply(w, c(1,3), sum)    # Row and matrix sums
   N  M
A  9 27
B 12 30
```

Lists

- A `list` is a general form of a vector, where the elements don't need to be of the same type or dimension.
- The function `list(...)` creates a list of the arguments
- Arguments have the form `name=value`. Arguments can be specified with and without names.
- Elements of a list can be referenced using
  - [ ], [[ ]] or $
- To apply functions for each list, `lapply()` can be used.

```
> x <- list(num=c(1,2,3), "Nick", identity=diag(2))
> x
$num
[1] 1 2 3

[[2]]
[1] "Nick"

$identity
     [,1] [,2]
[1,]    1    0
[2,]    0    1

> x[[2]]   # Second element of x
[1] "Nick"
> x[["num"]]   # Element named "num"
[1] 1 2 3
```

```
> x$identity   # Element named "identity"
     [,1] [,2]
[1,]    1    0
[2,]    0    1

> x[[3]][1,]   # First row of the third element
[1] 1 0
> x[1:2]   # A sublist of the first two elements
$num
[1] 1 2 3

[[2]]
[1] "Nick"
```

```
> A <- list(X=seq(101, 55, -5), Y=seq(205, 299, 8))
> A

$X
 [1] 101  96  91  86  81  76  71  66  61  56

$Y
 [1] 205 213 221 229 237 245 253 261 269 277 285 293

> lapply(A, sum)

$X
[1] 785

$Y
[1] 2988
```

Generating Factor

- For plotting and testing of hypotheses, we need to generate yet another type of a sequence, called a factor.
- For example, when for each of three experimental conditions there are measurements from five patients, the corresponding factor can be generated as follows:

```
> gl(3, 5)
[1] 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3
Levels:  1 2 3
```

- This is clearly different from a numerical sequence.

```
> rep(1:3, each=5)
[1] 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3
> class(gl(3,5))
[1] "factor"
> class(rep(1:3, each=5))
[1] "integer"
```

Generating Factor

- But, we can convert the numerical sequence to a factor.

```
> factor(rep(1:3, each=5))
[1] 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3
Levels:  1 2 3
> A <- factor(rep(1:3, each=5))
> class(A)
[1] "factor"
```

- The three conditions are often called levels of a factor.
- Each of these levels has five repeats corresponding to the number of observations within each level.

```
> table(A)

A
1 2 3
5 5 5
```

```
> fac <- gl(3, 5)
> num <- rep(1:3, each=5)
> summary(fac)
1 2 3
5 5 5

> summary(num)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
      1       1       2       2       3       3

> class(fac)
[1] "factor"
> class(num)
[1] "integer"
```

Classes of Different Vectors

```
> x <- c("Anna", "Peter", "Calvin")
> class(x)
[1] "character"
> x <- c(1.2, 1.5)
> class(x)
[1] "numeric"
> x <- 1:4
> class(x)
[1] "integer"
> x <- factor(1:4)
> class(x)
[1] "factor"
> x <- x > 12
> class(x)
[1] "logical"
```

Logical Operations

- Logical values are represented by the reserved words `TRUE` and `FALSE` in all caps or simply `T` and `F`.

`!x`	NOT x
`x & y`	x AND y elementwise, returns a vector
`x && y`	x AND y, returns a single value
`x | y`	x OR y elementwise, returns a vector
`x || y`	x OR y, returns a single value
`x %in y`	x IN y
`x < y`	$x < y$
`x > y`	$x > y$
`x <= y`	$x \leq y$
`x >= y`	$x \geq y$
`x == y`	$x = y$
`x != y`	$x \neq y$

```
> x <- 1:8
> x > 5
[1] FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE
> x %% 2 == 0
[1] FALSE TRUE FALSE TRUE FALSE TRUE FALSE TRUE
> # What elements of x are even or greater than 5?
> (x %% 2 == 0) | (x > 5)
[1] FALSE TRUE FALSE TRUE FALSE TRUE TRUE TRUE
> x[(x %% 2== 0) | (x > 5)]
[1] 2 4 6 7 8
> (x %% 2 == 0) & (x > 5)
[1] FALSE FALSE FALSE FALSE FALSE TRUE FALSE TRUE
> x[(x %% 2== 0) & (x > 5)]
[1] 6 8
```

```
> x
[1] 1 2 3 4 5 6 7 8
> y <- 5:12
> y
[1] 5 6 7 8 9 10 11 12
> x %in% y
[1] FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE
> y %in% x
[1] TRUE TRUE TRUE TRUE FALSE FALSE FALSE FALSE
> rev(x)
[1] 8 7 6 5 4 3 2 1
> sum(y/rev(x) >= 1)
[1] 6
> sum(y %% x != 0)
[1] 5
```

```
> x <- sqrt(2)
> x
[1] 1.414214
> x^2
[1] 2
> x^2 == 2
[1] FALSE
> x^2 != 2
[1] TRUE
> format(x^2, nsmall=20)
[1] "2.00000000000000044409"
> format(2, nsmall=20)
[1] "2.00000000000000000000"
> all.equal(x^2, 2)
[1] TRUE
> all.equal(x^2, 1)
[1] "Mean relative difference:  0.5"
```

Exercise

- Consider the following inequality

$$\frac{e^x + e^{-x}}{2^{|x|}} > 1 + \frac{\sqrt{|x|(1-|x|)}}{x+\sqrt{2}} \quad \text{for} \quad |x| \leq 1$$

1. Is this true or false?
2. Specify the range of x such that the inequality is always false.

Constructing a data frame

- A `data.frame` is used for storing data tables (like a matrix), but where different columns can contain different data types.
- `data.frame` are more flexible than matrices.
- A `data.frame` consists of a list of column vectors of equal length, where data in each column must be of the same type.

```
> data <- data.frame(
    patientID=c("101", "102", "103", "104"),
    treatment=c("drug","placebo","drug","placebo"),
    age=c(20, 30, 24, 22))
> data
> nrow(data)
> ncol(data)
> dim(data)
> str(data)
```

Examples of data.frame

- A vector in a `data.frame` can be referenced in multiple ways.

```
> data[[2]]
[1] "drug" "placebo" "drug" "placebo"
> data[["treatment"]]
> data$treatment
> data[,2]
```

- A row slice or subset of a `data.frame` returns a `data.frame` with a subset of the rows:

```
> data[c(1,3), ]
> treatmentIsDrug <- data$treatment=="drug"
> treatmentIsDrug
> data[treatmentIsDrug, ]
> subset(data, treatment=="drug")
```

```
> row.names(data) <- c("Anna", "Karl", "Esther", "Robert")
> data

> data[c("Karl", "Robert"),]
> data[2]
> data["treatment"]
> data[c("treatment", "age")]

> data[data$age > 23,]
> data[data$treatment=="drug",]
> gender <- c("F", "M", "F", "M")
> data.frame(data, gender=gender)
```

Missing Data

- R denotes data that is not available by NA
- How a function handles missing data depends on the function. For example, sample mean only ignores NAs if the argument na.rm = TRUE, whereas which always ignores missing data.

```
> x <- c(4, 7, 2, 0, 1, NA)
> mean(x)
[1] NA
> mean(x, na.rm=TRUE)
[1] 2.8
> which(x > 4)
[1] 2
> round(log(x), 4)
[1] 1.3863 1.9459 0.6931 -Inf 0.0000 NA
```

Missing Data

- Quantities that are not a number, such as 0/0, are denoted by `NaN`. In R `NaN` implies `NA` (`NaN` refers to unavailable numeric data and `NA` refers to any type of unavailable data)
- Undefined or null objects are denoted in R by `NULL`. This is useful when we do not want to add row labels to a matrix.
- Logical values are converted to numbers by setting
  - `TRUE` as `1`
  - `FALSE` as `0`.

```
> y <- NULL
> sum(y)
[1] 0
> seq(8) > 5
[1] FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE
> as.numeric(seq(8) > 5)
[1] 0 0 0 0 0 1 1 1
```

Detecting Missing Data

- Functions used for detecting missing data,
  - `is.na(x)` : Tests for `NA` or `NaN` data in `x`
  - `is.nan(x)` : Tests for `NaN` data in `x`
  - `is.null(x)` : Tests if `x` is `NULL`

```
> x <- c(4, 7, 2, 0, 1, NA)
> x == NA
[1] NA NA NA NA NA NA
> is.na(x)
[1] FALSE FALSE FALSE FALSE FALSE TRUE
> sum(is.na(x))
[1] 1
> (y <- x/0)
[1] Inf Inf Inf NaN Inf NA
> is.nan(y)
[1] FALSE FALSE FALSE TRUE FALSE FALSE
> is.na(y)
[1] FALSE FALSE FALSE TRUE FALSE TRUE
```

Testing and Coercing Objects

- All objects in R have a type. We can test the type of an object using a `is.type()` function.
- We can also attempt to coerce objects of one type to another using a `as.type()` function.

Type	Testing	Coercing
Array	`is.array()`	`as.array()`
Character	`is.character()`	`as.character()`
Dataframe	`is.data.frame()`	`as.data.frame()`
Factor	`is.factor()`	`as.factor()`
List	`is.list()`	`as.list()`
Logical	`is.logical()`	`as.logical()`
Matrix	`is.matrix()`	`as.matrix()`
Numeric	`is.numeric()`	`as.numeric()`
Vector	`is.vector()`	`as.vector()`

```
> x <- 1:10
> sum(x >= 5)
[1] 6
> is.vector(x)
[1] TRUE
> is.numeric(x)
[1] TRUE
> as.list(x)
[[1]]
[1] 1

.....

[[10]]
[1] 10
> as.numeric("123")
[1] 123
```