

# Programming for Chemical and Life Science Informatics

I573 - Week 7  
(Statistical Programming with R)

Rajarshi Guha

26<sup>th</sup> February, 2009

- The simplest type is a scatter plot of two vectors

```
> x <- 1:10  
> y <- x^2  
> plot(x,y)
```

- You can set various features of the plot such as axis labels, axis limits and so on

```
> plot(x, y, xlab='X variable', ylab='Y variable')  
> plot(x, y, xlim=c(1,5), ylim=c(0,20), type='l')
```

# Saving Figures

- R can save to JPEG, PNG, PDF, EPS formats
- On Windows you can also save in WMF format

```
> plot(x ,y)
> dev.copy(png, 'pic.png', width=600, height=600)
> dev.off()
```

- `dev.off()` is important otherwise you won't see any output on the screen
- PDF format allows you to generate a multipage PDF, with each page containing multiple plots

```
> pdf('multi.pdf')
> plot(1:10)
> plot(100:200)
> dev.off()
```

## Adding Points to a Plot

- After calling `plot()` it is possible to replot on the same figure

```
> plot(1:10)
> points( (1:10)^0.5, type='b', col='red')
```

- You can call `points()` multiple times to add more points to the plot
- The initial ranges of the X and Y values in `plot()` will define the extent of the plot

```
> plot( 1:10, col='blue' )
> points( 50:100, col='red' )
```

- You won't see any red points on the plot

# Vector Operations

- In most languages, if you want to perform an operation on each element of the array, you have to loop over the array
- You can do the same thing in R - but it's very slow
- Many R functions automatically work on an array

```
> x <- 1:10
> x^2
> x^2
 [1]  1  4  9 16 25 36 49 64 81 100
> x + 2
 [1] 3 4 5 6 7 8 9 10 11 12
```

# Vector Operations

- You can directly add vectors

```
> x <- 1:4
> y <- 10:13
> x + y
[1] 11 13 15 17
```

- Similarly for subtraction, division and multiplication
- What happens if you add vectors of different lengths?

```
> x <- 1:4
> y <- 1:5
> x + y
[1] 2 4 6 8 6
```

- R will *recycle* the shorter vector

# Vector Operations

X = 

1	2	3	4	5	6	7
---	---	---	---	---	---	---

Y = 

1	2	3	4
---	---	---	---

X = 

1	2	3	4	5	6	7
---	---	---	---	---	---	---

Y = 

1	2	3	4	1	2	3
---	---	---	---	---	---	---

- Basically the shorter vector (Y) is extended to the size of the longer vector (X)
- The new elements of Y are filled starting from Y[1]

# Vector Operations - An Example

- Evaluate the Euclidean distance between two vectors

$$D = \sqrt{\sum_i^n (x_i - y_i)^2}$$

- Get the squared differences

```
> x <- 1:4  
> y <- 5:8  
> z <- (x - y)^2
```

- Evaluate the sum

```
> z <- sum(z)
```

- Take the sqrt

```
> sqrt(z)
```

# Matrix Operations

- As with vector operations, many R functions work on each element of a matrix

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

- Sometimes you'd like to operate on columns or rows
- What is the mean of each column? Of each row?
- You could write a loop - slow for large matrices

# Matrix Operations

- Two shortcuts for row and column means (or sums)

```
> colSums(x)
[1]  6 15 24
> colMeans(x)
[1]  2  5  8
```

- But in many scenarios we'd like to compute something more complex based on the whole column
- Say we want to autoscale the columns of a matrix
- First consider how we would autoscale a vector,  $x$

$$x_i = \frac{x_i - \bar{x}}{\sigma}$$

# Matrix Operations

- We write a function, that takes a numeric vector and returns the autoscaled version

```
autoscale <- function(x) {  
  tmp <- x - mean(x)  
  return( tmp / sd(x) )  
}
```

- We can then test this out

```
> autoscale( 1:3 )  
[1] -1  0  1
```

- If you evaluate the SD of the return vector, it will be 1 and it's mean is 0

# Matrix Operations

- Each column of a matrix is a vector
- `apply` allows us to loop over columns of a matrix and at each iteration pass that column to a function

```
> apply(x, 2, autoscale)
      [,1] [,2] [,3]
[1,]  -1  -1  -1
[2,]   0   0   0
[3,]   1   1   1
```

- The 2 indicates that the function is applied over columns. Using 1 would apply it over rows

# Matrix Operations

- Say we want the minimum and maximum values from a column
- First write a function that does it on a single vector

```
get.range <- function(x) {  
  maxval <- max(x)  
  minval <- min(x)  
  return( c(minval, maxval) )  
}
```

- We can test it out by

```
> get.range( 1:5 )  
[1] 1 5
```

# Matrix Operations

- It is then easy to apply it to the columns of a matrix

```
> x <- matrix(1:9, nrow=3)
> apply(x, 2, get.range)
      [,1] [,2] [,3]
[1,]    1    4    7
[2,]    3    6    9
```

- Sometimes you need some extra arguments in the function you're applying

# Matrix Operations

- Remember that the function you pass to `apply` should expect a vector of values as its first argument
- If your function returns a vector, the result is the vectors generated from each column, bound together columnwise

# Matrix Operations

- Say we want to shift the range of a column by an amount  $s$
- We can modify the original function easily

```
get.range <- function(x, s) {  
  maxval <- max(x) + s  
  minval <- min(x) + s  
  return( c(minval, maxval) )  
}
```

- Testing this gives

```
> get.range( 1:5, 3 )  
[1] 4 8
```

# Matrix Operations

- If we use `get.range` with `apply`, it will receive the column as the first argument
- To send it the second argument we specify it explicitly

```
> x <- matrix(1:9, nrow=3)
> apply(x, 2, get.range, s=3)
      [,1] [,2] [,3]
[1,]    4    7   10
[2,]    6    9   12
```

- Note that the value of `s` is the same for each column
- You can supply functions with as many arguments as you want
  - ▶ The first argument is always a vector representing a column (or row)
  - ▶ Remaining arguments are specified by name in `apply`

# List Operations

- list objects are very useful, since they can hold arbitrary objects
- In many cases, we'd like to loop over the elements of a list and perform some function on each element
- lapply is what we need
- Consider a function that returns the class of its argument

```
get.class <- function(x) {  
  return( class(x) )  
}
```

# List Operations

- Next consider the list

```
> x <- list(a='hello', b=12, c=c(1,2,3))
```

- We can then apply our function to each element of the list

```
> lapply(x, get.class)
$a
[1] "character"

$b
[1] "numeric"

$c
[1] "numeric"
```

# List Operations

- Note that the return value of the function for each element of the list are collected in a list
- So your function can return any type of value (scalar, vector, matrix etc)
- In many cases, a function will return a single (say numeric) value and you'd like those values as a simple vector rather than a list

```
> unlist( lapply(x, get.class) )  
          a          b          c  
"character" "numeric" "numeric"
```

# List Operations

- Consider a list whose elements are vectors of different lengths
- Obviously we can't convert it to a matrix
- We want the length of each vector

```
x <- list(c(1,2,3),  
          c(4,5,6,7),  
          c(6,7,8,9,1))  
> lapply(x, length)  
[[1]]  
[1] 3  
  
[[2]]  
[1] 4  
  
[[3]]  
[1] 5
```