Additional Matrix and Vector Operations and Graphics 101

Song Liu (song.liu@bristol.ac.uk)
GA 18, Fry Building,
Microsoft Teams (search "song liu").

These are tricks not frequently used, but making good use of them (or avoid abusing them) may help you write good code.

Index Elements Except ...

- If you want to index elements in your vectors except specific elements, you can do

```
a <- c(1,2,3,4)
a[-3] # all elements in a without the 3rd one

[1] 1 2 4
```

Index Rows/Cols Except ...

- You can also use it to index rows in a matrix except ...

```
A <- matrix(1:9, nrow = 3)
A[-3,] # all rows in A except the third row
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
# all rows in A except the 2nd and 3rd row
A[c(-2,-3),]
[1] 1 4 7
```

Concatenate Vectors

- You can concatenate two vectors to create a bigger vector.

```
a <- 1:4
b <- 5:8
ab <- c(a, b)
ab
[1] 1 2 3 4 5 6 7 8
```

Concatenate Matrices

- You can concatenate two matrices in R, by rows or by columns.
- `cbind`, concatenate two matrices by columns

```
a <- matrix(1:4, nrow = 2)
b <- matrix(5:8, nrow = 2)
a
     [,1] [,2]
[1,]    1    3
[2,]    2    4
b
     [,1] [,2]
[1,]    5    7
[2,]    6    8
ab <- cbind(a, b)
ab
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
```

Concatenate Matrices

- `rbind`, concatenate two matrices by rows

```
a <- matrix(1:4, nrow = 2)
b <- matrix(5:8, nrow = 2)
a
     [,1] [,2]
[1,]    1    3
[2,]    2    4
b
     [,1] [,2]
[1,]    5    7
[2,]    6    8
ab <- rbind(a, b)
ab
     [,1] [,2]
[1,]    1    3
[2,]    2    4
[3,]    5    7
[4,]    6    8
```

Insert New Elements in a Vector

```
a <- 1:4
# inserting a new element 2.5 after 2.
a <- c(a[1:2], 2.5 ,a[3:4])
a
[1] 1.0 2.0 2.5 3.0 4.0
```

- Once a vector is created by R, you cannot change its size.
- The above code does NOT modify the original vector `a`. Instead, it created a new vector by calling `c(a[1:2], 2.5 ,a[3:4])`, and reassigned it to the original variable `a`.
- This operation is called "insertion by reassignment", requires **additional memory allocations**, thus is time-consuming when carried out repeatedly.

Delete Elements in a Vector

```
a <- 1:4
# delete the third and fourth element.
a <- a[c(-3,-4)]
a
[1] 1 2
```

Similar to vector insertion, you are not modifying the original vector. You are simply creating a new vector by `a[c(-3,-4)]` and assigned it to `a`.

Delete Rows/Columns in a Matrix

```
A <- matrix(1:9, nrows = 3)
A
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
# delete the second row.
A <- A[-2, ]
A
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    3    6    9
```

Tetris

- Eliminate the row(s) that are filled with 1s.

```
A <- matrix(c(0, 0, 0, 0,
+               1, 0, 1, 0,
+               1, 0, 1, 1,
+               1, 1, 1, 1), nrow = 4, byrow = T)
A
     [,1] [,2] [,3] [,4]
[1,]    0    0    0    0
[2,]    1    0    1    0
[3,]    1    0    1    1
[4,]    1    1    1    1
A <- A[rowSums(A) != 4,]
     [,1] [,2] [,3] [,4]
[1,]    0    0    0    0
[2,]    1    0    1    0
[3,]    1    0    1    1

A
```

Recycling

- In vector ops, R requires two vectors to be **the same length**.
- If two vectors do not have the same length, R automatically repeats the shorter vector to match the length of the longer one.

```
a <- 1:4
b <- 1:8

a + b #a is repeated twice
# the same as c(1:4,1:4) + 1:8
[1]  2  4  6  8  6  8 10 12
```

Recycling

- If the longer vector is not a multiple of the shorter vector, the shorter vector will repeat until it is long enough.

```
a <- 1:3
b <- 1:4
a + b
# the same as c(1:3, 1) + 1:4
[1] 2 4 6 5
Warning message: In a + b : longer object length
is not a multiple of shorter object length
```

Recycling

- Recycling does NOT works on matrices

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

A
B

A+B
Error in A + B : non-conformable arrays
```

- In NumPy or MATLAB, there is a feature called "broadcasting", which allows recycling in matrices.

Recycling

- However, the following code works

```
A <- matrix(1:4, nrow = 2)
B <- matrix(1:2, nrow = 2, ncol = 2)
A
     [,1] [,2]
[1,]    1    3
[2,]    2    4
B
     [,1] [,2]
[1,]    1    1
[2,]    2    2

A+B
         [,1] [,2]
[1,]    2    4
[2,]    4    6
```

The matrix function automatically repeat the vector `1:2` twice to create a 2 by 2 matrix.

Recycling

In other words,

```
B <- matrix(1:2, nrow = 2, ncol = 2)
```

is the same as

```
B <- matrix(c(1:2, 1:2), nrow = 2, ncol = 2)
```

When creating a matrix, if not enough elements are provided to fill the matrix, the data is automatically recycled.

`apply` Function to Rows/Cols of Matrix

- If you have a function, and want to apply this function to all rows/columns to a matrix, you can use `apply` function.

```
# create a 100 by 2 random matrix, filled with samples
# from the standard normal distribution.
# Each row of x is a random point in 2D space.
x <- matrix(rnorm(2*100), nrow = 100)
# checking if a point vector is in the unit circle.
is_in_circle <- function(p){
    if( sqrt(sum(p^2)) <1 ){
        return(T)
    }else{
        return(F)
    }
}
```

- How do I apply `is_in_circle` to all the rows in `x`?

`apply` Function to Rows/Cols of Matrix

`apply(m,dim,f)`: applies `f` to rows (when `dim = 1`) or columns (when `dim = 2`) to matrix `m`.

```
apply(x,1,is_in_circle)
[1] FALSE TRUE TRUE FALSE TRUE FALSE FALSE ...
```

`apply` Function to Rows/Cols of Matrix

- Your function can also output a vector:

```
# project a point p on the circle.
project_on_circle <- function(p){
return(p / sqrt(sum(p^2)))
}
```

- `apply` will run your function on rows/columns of `m` and stack the outcome vectors **by column**.

```
px <- apply(x,1,project_on_circle)
px
          [,1]       [,2]      [,3]       [,4]      [,5]
[1,]  0.1483527 -0.9574072 0.9174074 -0.9469062 0.6920671
[2,] -0.9889345 -0.2887412 0.3979494 -0.3215100 0.7218332
...
```

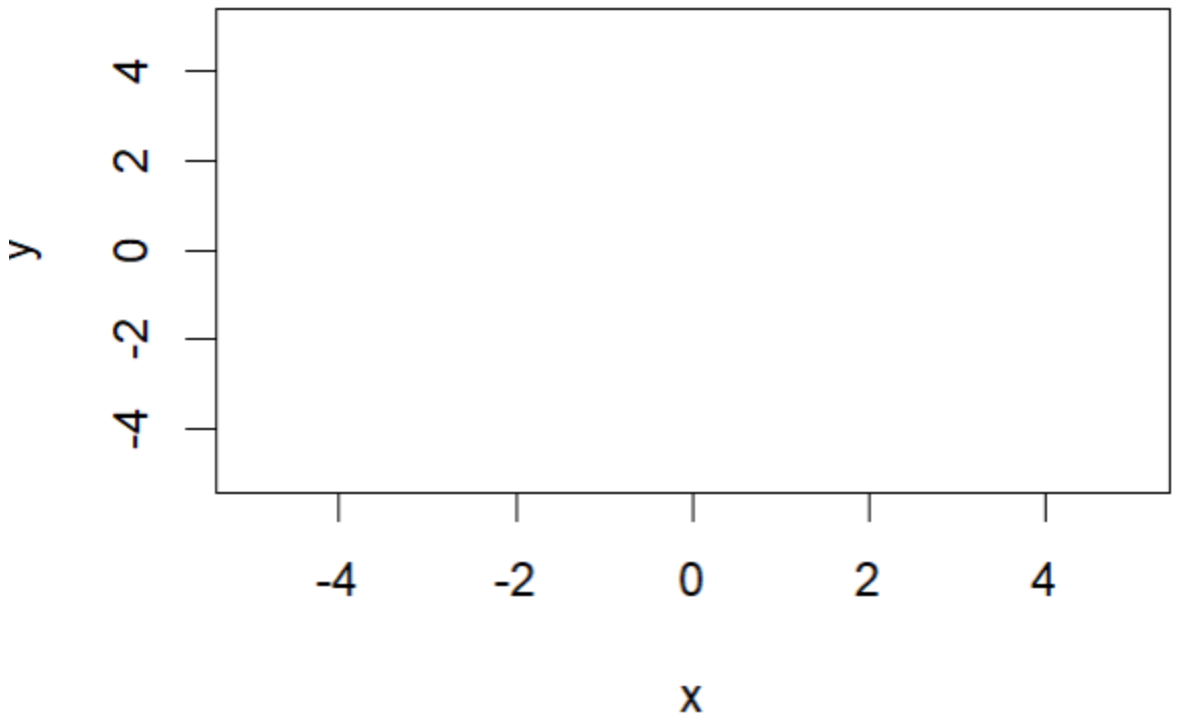
Graphics 101 in R

- R has a powerful graphics features, which allows users to visualize data easily.
    - Hint: You can type `demo("graphics")` in the command line to see demo code/plots.
- You may have already seen some of the graphics function, so I will be brief.

Create a plot: `plot`

`plot` function is usually used to create an empty canvas, ready for further plotting actions.

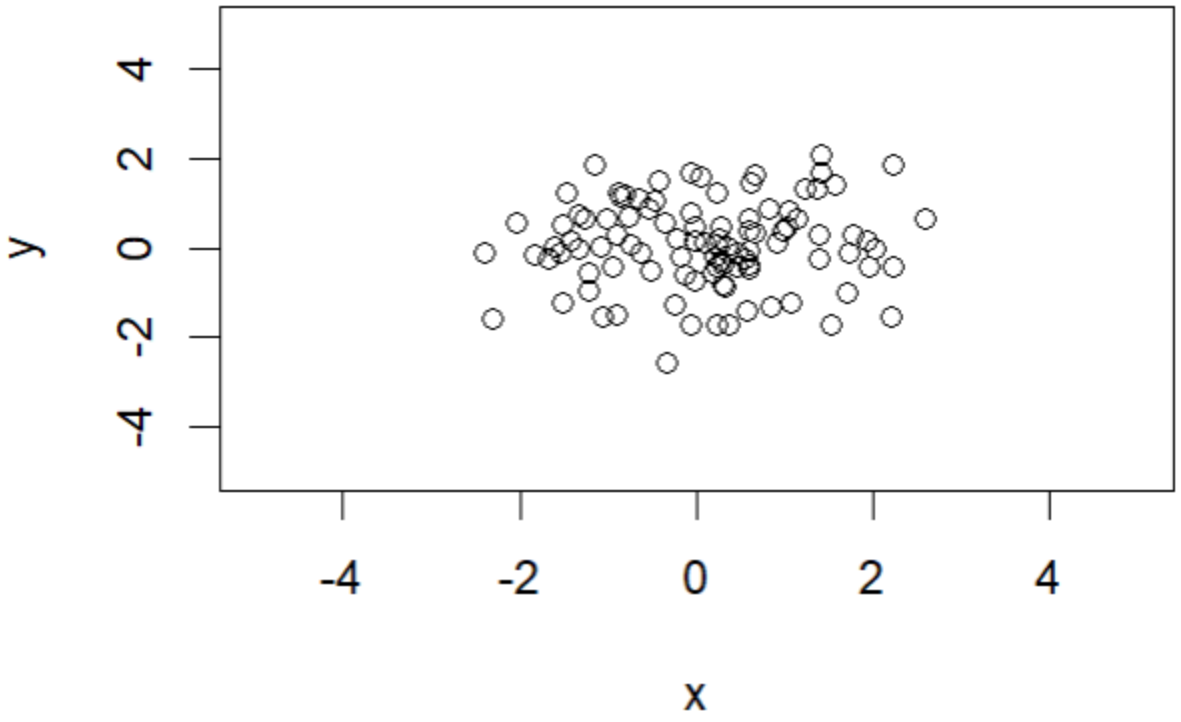
```
# create an empty plot, with x axis ranges from -5 to 5
# y axis ranges from -5 to 5.
plot(c(-5,5),c(-5,5), type = "n", xlab = "x", ylab = "y")
```



Visualize Data Points

`points` can be used to draw data points on a 2D plot.

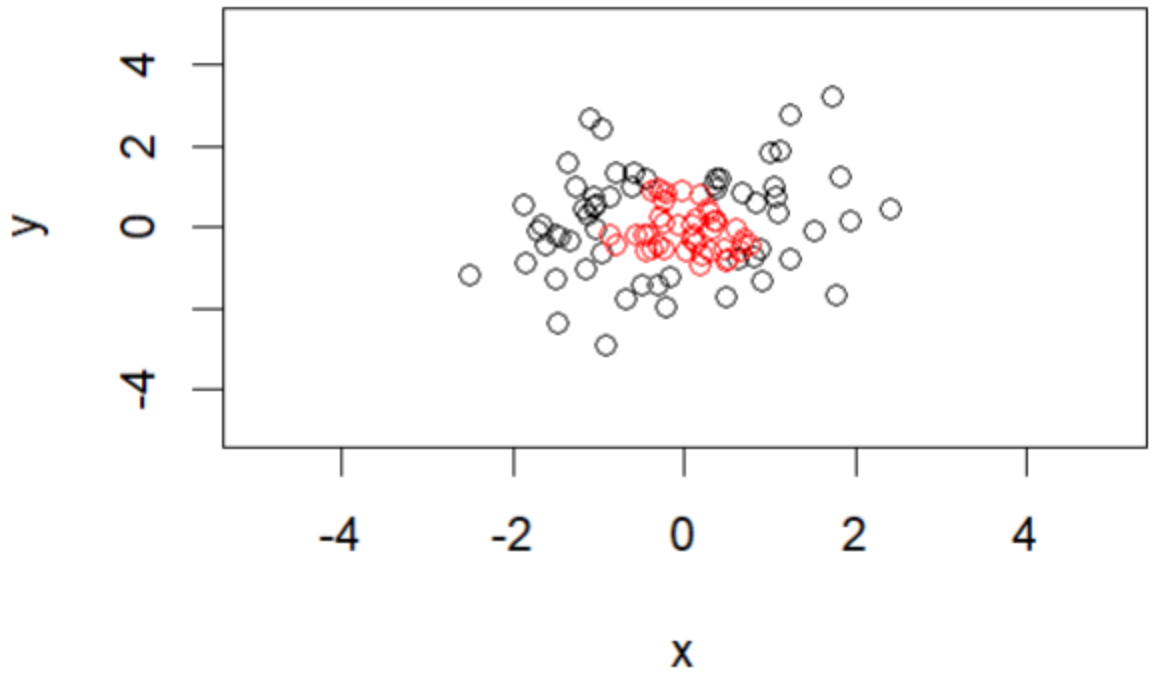
```
x <- rnorm(100)
y <- rnorm(100)
# draw points on the current plot. The first dimension
# of points are stored in a vector x and the second
# dimension are stored in a vector y.
points(x,y)
```



Visualize Data Points

You can change the the color of plotted points

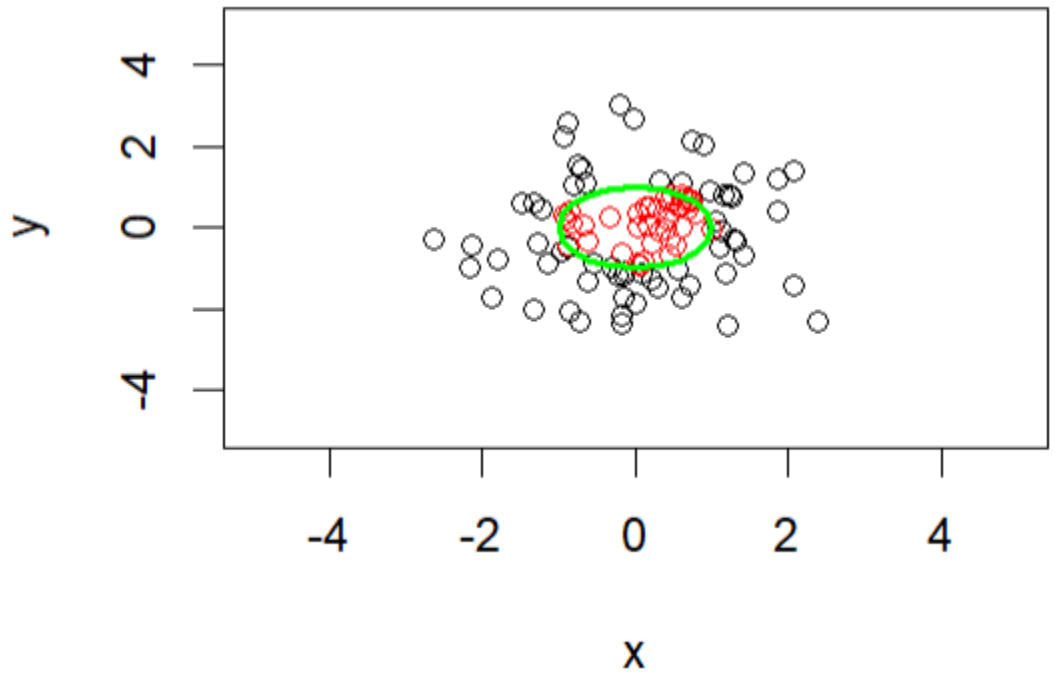
```
p <- cbind(x,y)
in_circ <- t(apply(p,1, is_in_circle))
# plot all points in the unit circle in red
points(p[in_circ,1],p[in_circ,2],col="red")
```



Draw Lines

You can draw line in a 2D plot using `lines` function:

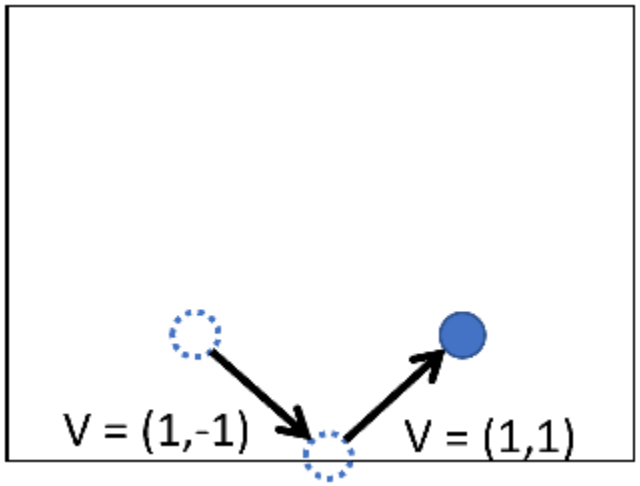
```
# generate points on a circle
circ <- cbind(cos(0:20*pi/10), sin(0:20*pi/10))
# draw lines by connecting these points
lines(circ[,1],circ[,2], col = "green", lwd = 2)
```



Homework

0. Read lecture slides on visualizations carefully.
1. Create an empty plot, whose x-axis and y-axis range from -5 to 5.
2. Initialize two vectors:
    - $x = (0, 0)$ and $v = (.23456, .12345)$.
    - Draw the vector x as a **red** dot in the plot.
    - Add plot title: "point simulation, press ESC to stop."

Homework

3. Now let us write a function that updates the point `x`'s location based on the velocity vector `v`.
    - If the position of `x` is slightly out of the box, the point will be **bounced back**.
    - 

Homework

3. Write a function `update(x,v)` that takes two vectors `x` and `v` as inputs and returns the updated `x` and `v`.
    - First, the function checks if x is outside of the boundary.
        - If so, it updates the velocity vector by "reflecting it". See the picture in the previous slide.
    - Then, update `x <- x + v`
    - Finally, `return(list(x,v))`

- Hint: `x[2] > 5 || x[2] < -5` is true if x is above the ceiling or is below the floor.

Homework (submit)

4. Your code should look like this:

```
update <- function(x,v){
    # TODO: Check the boundary collision, and update v.

    x <- x + v #update x
    return(list(x,v))
}

x <- c(0,0) #the initial position of x
v <- c(.23456,.12345) #the initial velocity of x

while(T){
    #TODO: create a new plot and draw x's current location

    xv <- update(x,v) # update x and v
    # getting new x and v from the returned list.
    # list indexing uses double brackets [[]]
    x<- xv[[1]]
    v<- xv[[2]]

    # wait a bit to allow RStudio plot the picture.
    Sys.sleep(.1)
}
```

Homework (submit)

- Your plot should look like this. https://github.com/anewgithubname/MATH10017/raw/main/labs/lab14_1.mp4

Homework (Challenge)

5. Now, let us add more points to the simulation.
    - Copy your previous code to a new R file.
    - Replacing the old vectors `x` and `v` with two 50×2 matrices `x` and `v`.
        - Elements in `x` are samples drawn from a uniform distribution $U(-1, 1)$.
        - Elements in `v` are samples drawn from a uniform distribution $U(-.5, .5)$.
    - Now `x` contains 50 points in a 2D space while the corresponding rows of `v` are velocity vectors for `x`.
6. Rewrite `update` function **using vectorized code** so that it can update 50 points' locations at once.

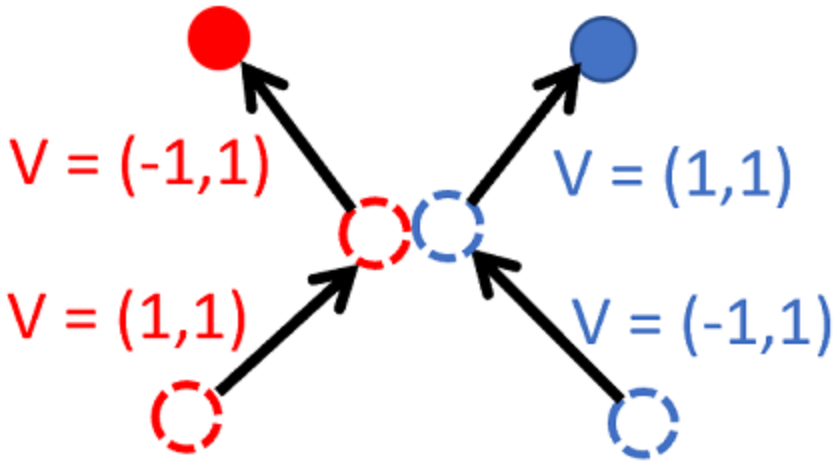
Homework (Challenge)

- Hint: `(x[,2] > 5 | x[,2] < -5)` will produce a **logical vector** that is `TRUE` for points which are above the ceiling or below the floor.

Homework (Challenge)

- Your plot should look like this: https://github.com/anewgithubname/MATH10017/raw/main/labs/lab14_2.mp4

Homework (difficult Challenge)

- Now consider each point is a radius 0.2 tennis ball.
- Rewrite your `update` function so it checks for the collision of balls.
  - If two balls collide, they will exchange momentum.
  - 

  - Consider using the `pdist` functions we wrote last week for checking ball collisions.