Random Numbers

Gaston Sanchez

October 9, 2016

Random Numebrs

Random numbers have many application in science and computer propgramming, especially when there are significant uncertainties in a phenomenon of interest. The goal of this unit is to look at some practical problmes involving working with random numbers and creating simulations.

The key idea in computer simulations with random numbers is first to formulate an algorithmic description of the phenomenon we want to study. This description frequently maps directly onto a simple R script, where we use random numbers to mimix the uncertain features of the phenomenon. The script needs to perform a large number of repeated calculations, and the final answers are only approximate, but the accuracy can usually be made good enough for practical purposes.

Generating Random Numbers

R has several functions that allows you to generate random numbers following a certain distribution.

The function `sample()` takes a random sample from the input vector.

All computations of random numbers are based on deterministic algorithms, so the sequence of numbers is not truly random. However, the sequence of numbers appears to lack any systematic pattern, and we can therefore regard the numbers as random.

The Seed

Every time you use one of the random generator functions in R, the call produces different numbers. For replication and debugging purposes, it is useful to get the same sequence of random numebrs every time we run the script. This functionality is obtained by setting a **seed** before we start generating the numebrs. The seed is an integer and set by the function `set.seed()`

```
set.seed(123)
runif(4)
```

```
## [1] 0.2875775 0.7883051 0.4089769 0.8830174
```

If we set the seed to `123` again, the sequence of uniform random numbers is regenrated:

```
set.seed(123)
runif(4)
```

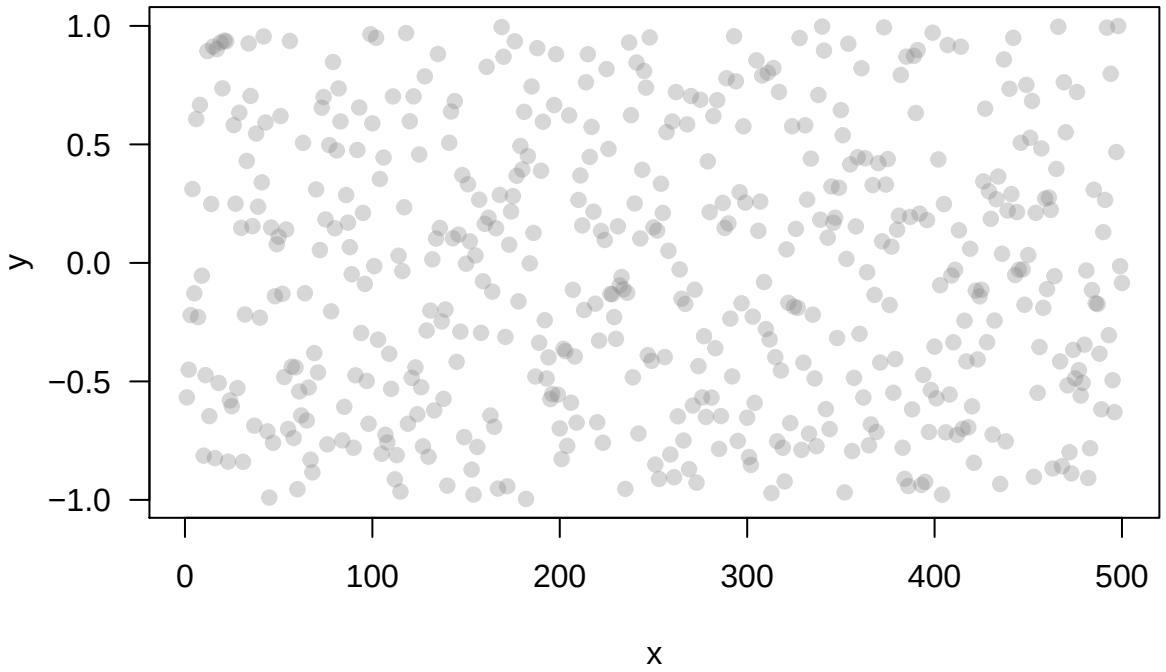
```
## [1] 0.2875775 0.7883051 0.4089769 0.8830174
```

If we don't specify a seed, the random generator functions set a seed based on the current time. That is, the seed will be different each time we run the script and consequently the sequence of random numbers will also be different.

Uniformly Distributed Random Numebrs

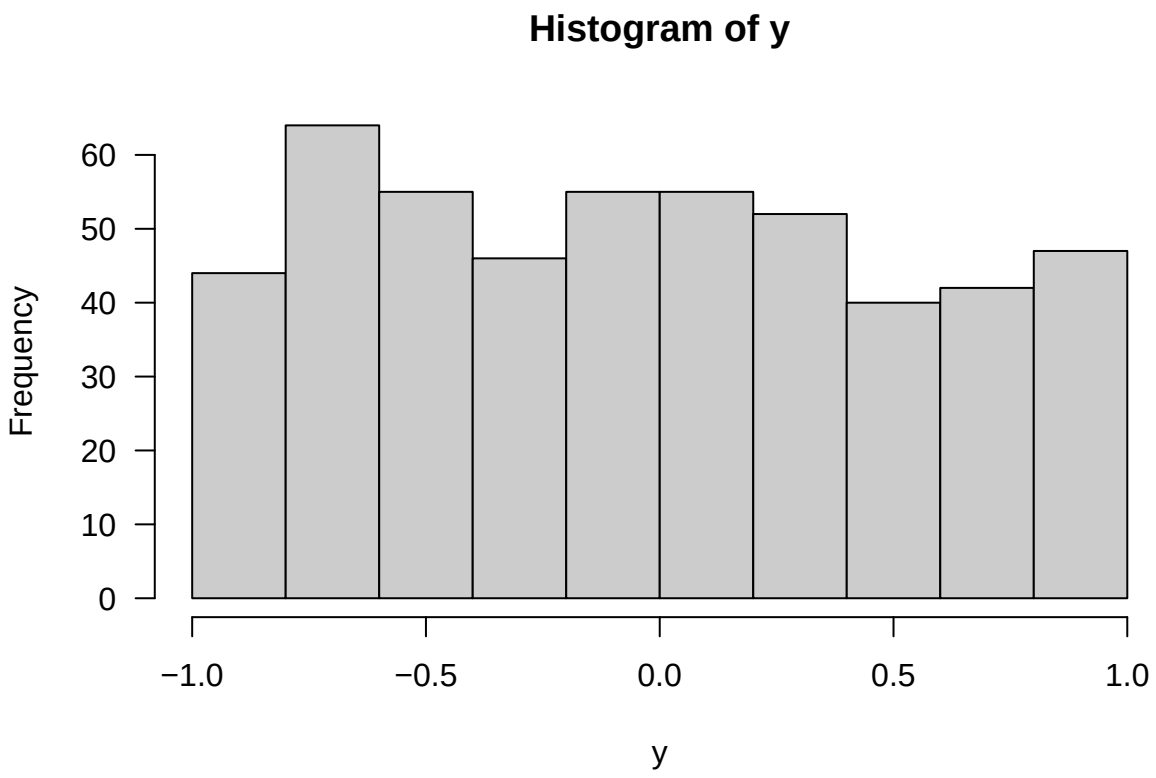
The numbers generated by `runif()` tend to be equally distributed between 0 and 1, which means that there is no part of the interval $[0, 1]$ with more random numbers than other parts.

```
set.seed(345)
n <- 500
x <- 1:n
y <- runif(n, min = -1, max = 1)

plot(x, y, las = 1, pch = 19, col = "#88888855")
```



The previous figure shows the values of 500 random uniformly distributed numbers between -1 and 1. Another interesting visualization involves looking at how the random numbers are distributed in the given interval.

```
hist(y, las = 1, col = "gray80")
```



Drawing Integers

To simulate drawing random integers, you can use the function `sample.int()`. The main argument is `n`, which represents the maximum integer to sample from: `1, 2, 3, ..., n`

```
sample.int(10)
```

```
##  [1]  7  6  1  5  8  4  9  2  3 10
```

Computing Probabilities

With the mathematical rules from probability theory we can compute the probability that a certain event happens, say the probability that you get one blue ball when drawing three balls from a box with four blue balls, five white balls, and three yellow balls. Unfortunately, theoretical calculations of probabilities may soon become hard or impossible if the problem is slightly changed. There is a simple numerical way of computing probabilities that is generally applicable to problems with uncertainty: Monte Carlo Simulation.

Principles of Monte Carlo Simulation

Assume that we perform N experiments where the outcome of each experiment is random. Suppose that some event takes place M times in these N experiments. An estimate of the probability of the event is then M/N. The estimate becomes more accurate as N increases, and the exact probability is assumed to be reached in the limit as $N \to \infty$.

Programs that run a large number of experiments and record the outcome of events are often called simulation programs. The mathematical technique of letting the computer perform lots of experiments based on drawing

random numbers is commonly called **Monte Carlo simulation**. Many probabilistic problems can be calculated exaclty by mathematics from probability theory, but very often Monte Carlo simulation is the only way to solve statistical problems.

Example: Drawing balls from a box

Suppose there are 11 balls in a box: four blue, five white, and three yellow. We want to write R code that draws three balls at random without replacement from the box, and compute the probability of drawing two or more blue balls.

```
# colored balls in a box
box <- c(rep('blue', 4), rep('white', 5), rep('yellow', 3))
box
```

```
##  [1] "blue"   "blue"   "blue"   "blue"   "white"  "white"  "white"
##  [8] "white"  "white"  "yellow" "yellow" "yellow"
```

```
# drawing three balls without replacement
sample(box, 3)
```

```
## [1] "blue" "blue" "blue"
```

In this example you can apply theprobability rules to find the probability of getting at least two blue balls from drawing three balls out of the box. But let's see how to use Monte Carlo simulation to find an approximate value for such probability. We are going to simulate 1000 repetitions of the experiment. Each repetition involves drawing three balls. To store the outputs we can use a matrix of 1000 rows and three columns:

```
# number of balls drawn, and repetitions
draw <- 3
reps <- 1000

outputs <- matrix("", nrow = reps, ncol = draw)
set.seed(123)
for (i in 1:reps) {
  outputs[i,] <- sample(box, draw)
}
head(outputs)
```

```
##      [,1]     [,2]     [,3]
## [1,] "blue"   "white"  "white"
## [2,] "yellow" "yellow" "blue"
## [3,] "white"  "yellow" "white"
## [4,] "white"  "yellow" "white"
## [5,] "white"  "white"  "blue"
## [6,] "yellow" "blue"   "blue"
```

Now that we have our 1000 repetitions, let's compute the number of blue balls for each row in `outputs`:

```
# number of blue balls in each experiment
blues <- apply(outputs, 1, function(x) sum(x == "blue"))

# number of at least two blue balls
sum(blues >= 2)
```

```
## [1] 241
```

```
# approx probability of two or more blue balls
sum(blues >= 2) / reps
```

```
## [1] 0.241
```

Monte Carlo Integration

One of the earliest applications of random numbers was numerical computation of integrals, that is, a non-random (deterministic) problem. We will consider two related methods for computing:

$$\int_a^b f(x)dx$$

Standard Monte Carlo Integration

Let $x_1, x_2, \ldots, x_n$ be uniformly distributed random numbers between a and b. Then

$$(b-1)\frac{1}{n}\sum_{i=1}^{n} f(x_i)$$

is an approximation to the integral $\int_a^b f(x)dx$.

This method is referred to as **Monte Carlo integration**. How does it work? A well-known result from calculus is that the integral of a function f over $[a, b]$ equals the mean value of f over $[a, b]$ multiplied by the length of the interval $b - a$. If we approximate the mean value of $f(x)$ by the mean of n randomly distributed function evaluations $f(x_i)$, we get the integral.

Here's a function to perform Monte Carlo integration

```
mc_integration <- function(f, a, b, n) {
  x <- runif(n, min = a, max = b)
  (b - a) * mean(f(x))
}
```

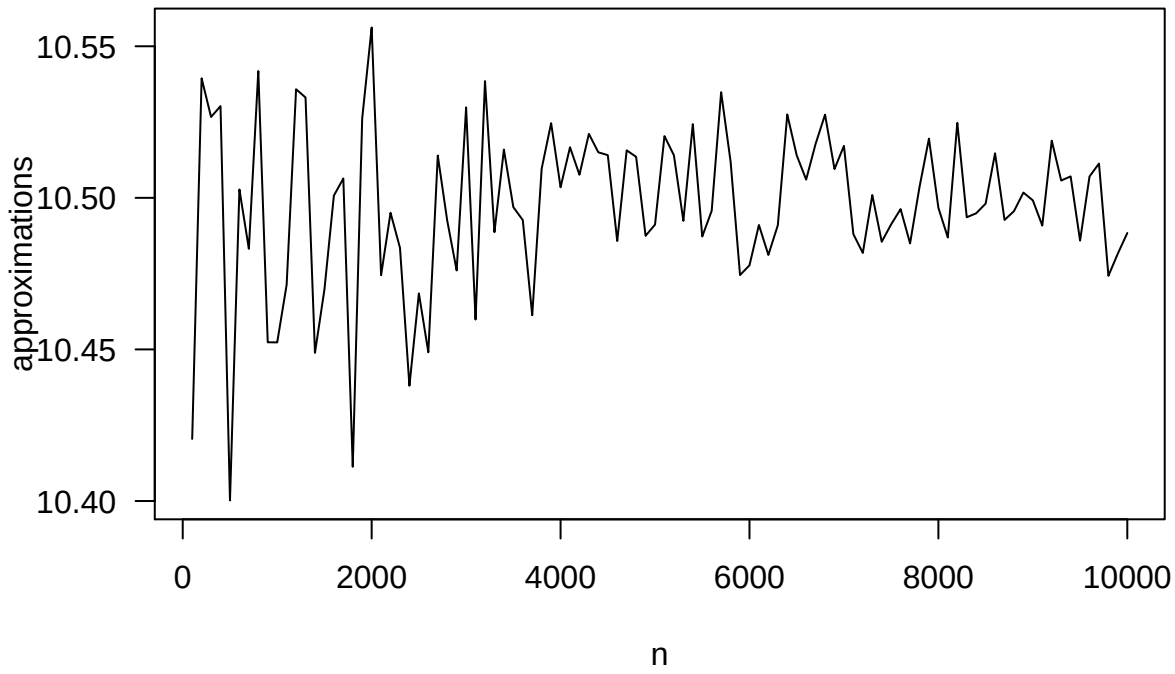
Let's try the Monte Carlo integration method on a simple linear funciton $f(x) = 3 + 5x$, integrated from 1 to 2.

```
f1 <- function(x) 3 + 5*x

mc_integration(f1, 1, 2, 100)
```

```
## [1] 10.63283
```

Most other numerical integration methods will integrate such a linear function exactly. This is not the case with Monte Carlo integration. It would be interesting to see how the quality of the Monte Carlo approximation behaves as n increases.

```r
n <- seq(100, 10000, by = 100)

approximations <- rep(0, length(n))

for (i in 1:length(n)) {
  approximations[i] <- mc_integration(f1, 1, 2, n[i])
}

plot(n, approximations, type = "l", las = 1)
```



For functions of many variables, Monte Carlo integration in high space dimension completely oytperforms methods like the Trapezoidal rule and Simpson's rule.

Area Computing by Throwing Random Points

The second method that involves Monte Carlo simulation for computing:

$$\int_a^b f(x)dx$$

is by throwing random points to some geometric region.

Consider a geometric region G in the plane and a surrounding bounding box B with geometry $[x_L, x_H]\times[y_L, y_H]$. One way of computing the area of G is to draw N random points inside B and count how many of them, M, that lie inside G. The area of G is then the fraction M/N times the area of B, $(x_H - x_L)(y_H - y_L)$. This method behaves like a dart game where you record how many hits there are inside G if every throw hits uniformly within B.

Computing the integral $\int_a^b f(x)dx$ implies computing the area under the curve $y = f(x)$ and the above axis x, between $x = 1$ and $x = b$. We introduce the reacngle B

$$B = \{(x, y)|a \leq x \leq b, 0 \leq y \leq m\}$$

where $m \leq max_{x \in [a,b]} f(x)$.

The algorithm for computing the area under the curve is to draw N random points inside B and count how many of them, M, that are above the x axis and below the $y = f(x)$ curve. The are is then estimated by:

$$\frac{M}{N} m(b-a)$$

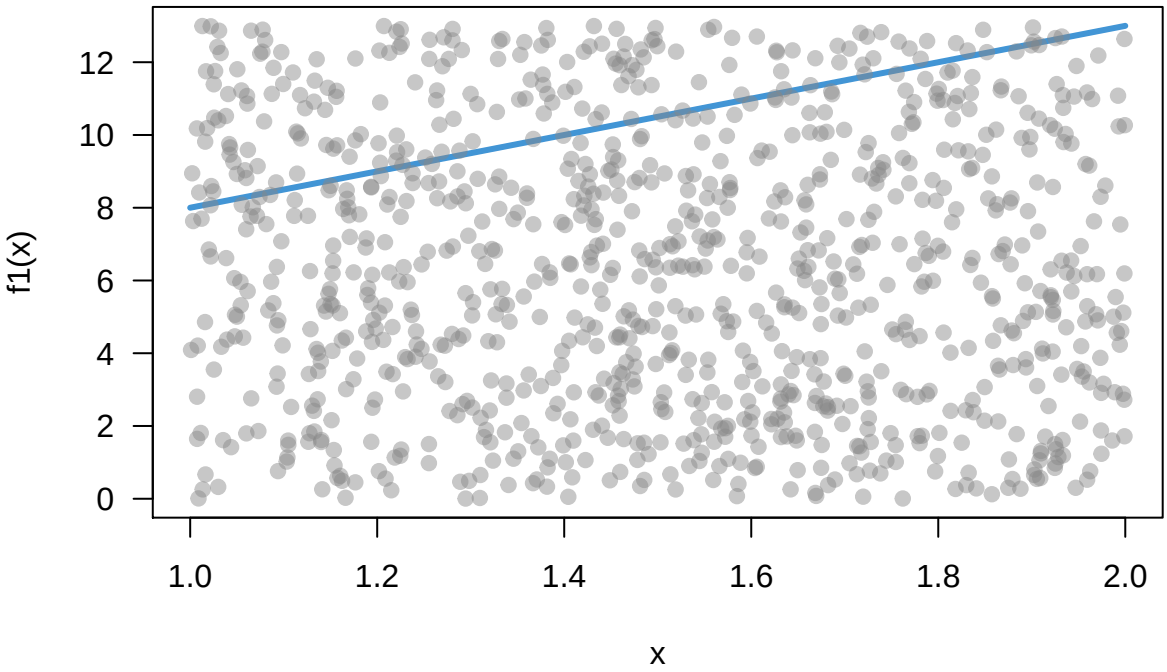
Let's consider the prefiouvs linear function $f(x) = 3 + 5x$, over the x interval from 1 to 2.

```
mc_area <- function(f, a, b, n, m) {
  below <- 0
  x <- runif(n, min = a, max = b)
  y <- runif(n, min = 0, max = m)
  below <- sum(f1(x) < y)
  area <- (b - a) * m * (below / n)
  area
}
```

Let's try the Monte Carlo dart method on $f(x) = 3 + 5x$.

```
f1 <- function(x) 3 + 5*x

mc_area(f1, 1, 2, 1000, 13)
```

```
## [1] 2.535
```



Random Walk in One Space Dimension

In this section we will simulate a collection of particles that move around in a random fashion. This type of simulations are fundamental in physics, biology, chemistry as well as other sciences and can be used to

describe many phenomena. Some application areas include molecular motion, heat conduction, quantum mechanics, polymer chains, population genetics, and pricing of financial instruments.

Imagine that we have some particles that perform random moves, either to the right of to the left. We may flip a coin to decide the movement of each particle: *heads* implies movement to the right, and *tails* implies movement to the left. Each move is one unit length.

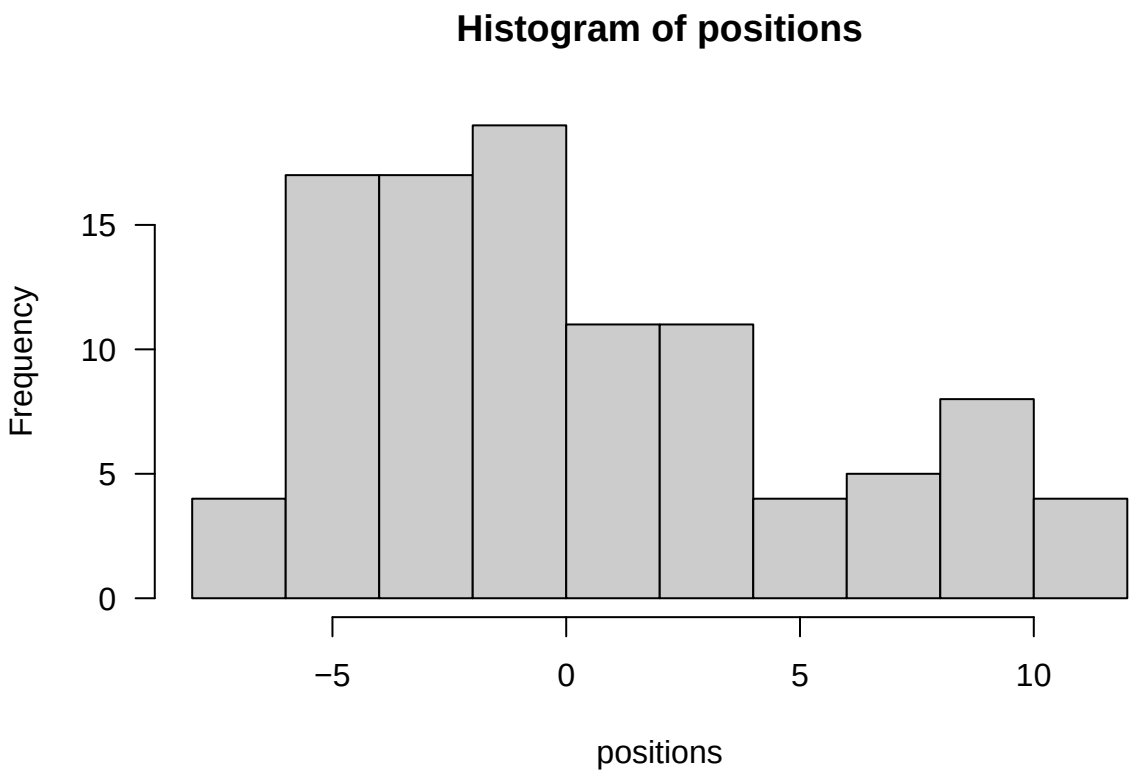
How can we implement n_s random steps of n_p particles in R? First we need a coordinate system where all movements are along the x axis. We draw numbers to simulate movement to the right (1) or to the left (-1):

```
# random walk for one particle
steps <- 100
left_or_right <- c(-1, 1)

# start in position 0
previous <- 0
positions <- rep(0, steps)

set.seed(123)
for (s in 2:steps) {
  # move to left or right?
  move <- sample(left_or_right, 1)
  positions[s] <- previous + move
  previous <- positions[s]
}

# final position after 100 steps
positions[steps]
```

```
## [1] -7
```

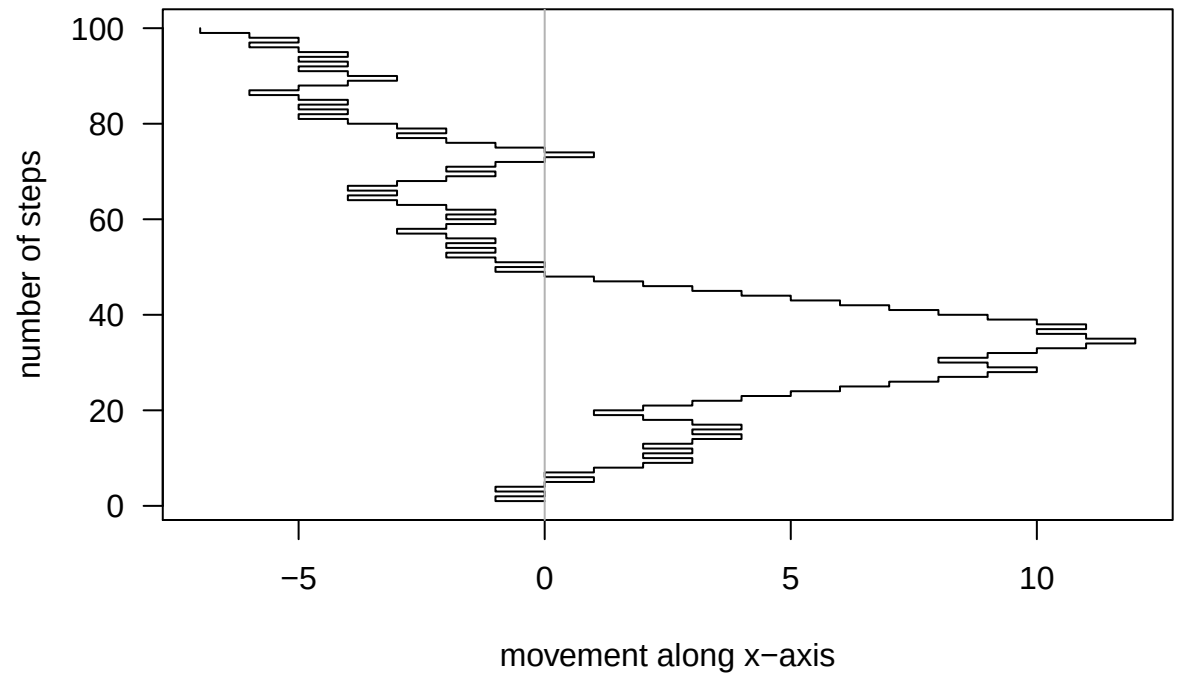
Look at the distribution:

```
hist(positions, col = "gray80", las = 1)
```



We may add some visualiztion of the movements:

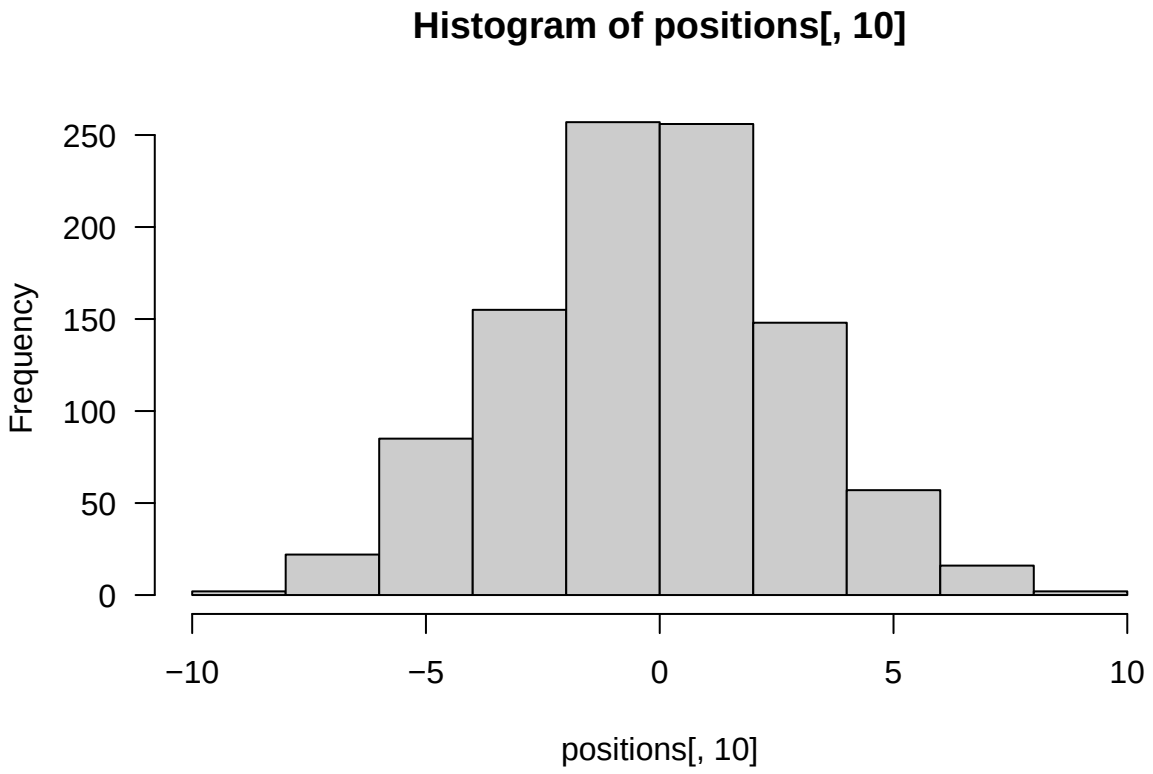
```
plot(positions, 1:steps, las = 1, type = "s",
     xlab = "movement along x-axis",
     ylab = "number of steps")
abline(v = 0, col = "gray70")
```



Now let's simulate random walks for many particles:

```r
# random walk for one particle
particles <- 1000
steps <- 30

# matrix to store positions
positions <- matrix(0, nrow = particles, ncol = steps)
left_or_right <- c(-1, 1)

set.seed(123)
for(p in 1:particles) {
  # start in position 0
  previous <- 0
  for (s in 2:steps) {
    # move to left or right?
    move <- sample(left_or_right, 1)
    current <- previous + move
    positions[p, s] <- current
    previous <- current
  }
}
```

```r
# distribution of positions of 1000 particles at step 10
hist(positions[,10], las = 1, col = "gray80")
```



Computing statistics of the particle positions

When simulating random walks, the analysis of such simulations is not really focused on the graphics of the walk, but more in the statistics of the positions of the particles at each step.A more interesting visualization

consists of looking at the distribution of the particles along the x axis, plus estimate the mean position and the standard deviation.

```
# statistics of positions
hist_positions <- function(positions, step = 1) {
  mean_positions <- colMeans(positions)
  sd_positions <- apply(positions, 2, sd)
  hist(positions[ ,step], col = "gray80", las = 1,
       main = paste("positions at step", step))
  abline(v = mean_positions[step], col = "tomato", lwd = 3)
  abline(v = sd_positions[step], col = "orange", lwd = 3)
  abline(v = -sd_positions[step], col = "orange", lwd = 3)
}

hist_positions(positions, step = 20)
```