# Laborator 3

Elemente de probabilități în R

Obiectivul acestui laborator este de a prezenta succint câteva funcții utile teoriei probabilităților din programul R, care este structura lor și cum le putem aplica. De asemenea, tot în acest laborator vom prezenta și câteva probleme ce se pot rezolva cu ajutorul algoritmilor aleatori.

## 1 Familia de funcții `apply`

Pe lângă buclele `for` și `while`, în R există și un set de funcții care permit scrierea și rularea într-o manieră mai compactă a codului dar și aplicarea de funcții unor grupuri de date.

- `lapply()`: Evaluează o funcție pentru fiecare element al unei liste
- `sapply()`: La fel ca `lapply` numai că încearcă să simplifice rezultatul
- `apply()`: Aplică o funcție după fiecare dimensiune a unui `array`
- `tapply()`: Aplică o funcție pe submulțimi ale unui vector
- `mapply()`: Varianta multivariată a funcției `lapply`
- `split`: Împarte un vector în grupuri definite de o variabilă de tip factor.

### 1.1 `lapply()`

Funcția `lapply()` efectuează următoarele operații:

1. buclează după o listă, iterând după fiecare element din acea listă
2. aplică o *funcție* fiecărui element al listei (o funcție pe care o specificăm)
3. întoarce ca rezultat tot o listă (prefixul `l` vine de la listă).

Această funcție primește următoarele trei argument: (1) o listă `X`; (2) o funcție `FUN`; (3) alte argumente via `...`. Dacă `X` nu este o listă atunci aceasta va fi transformată într-una folosind comanda `as.list()`.

Considerăm următorul exemplu în care vrem să aplicăm funcția `mean()` tuturor elementelor unei liste

```r
set.seed(222)
x <- list(a = 1:5, b = rnorm(10), c = rnorm(20, 1), d = rnorm(100, 5))
lapply(x, mean)
$a
[1] 3

$b
[1] 0.1996044

$c
[1] 0.7881026

$d
[1] 5.064188
```

Putem să folosim funcția `lapply()` pentru a evalua o funcție în moduri repetate. Mai jos avem un exemplu în care folosim funcția `runif()` (permite generarea observațiilor uniform repartizate) de patru ori, de fiecare dată generăm un număr diferit de valori aleatoare. Mai mult, argumentele $min = 0$ și $max = 3$ sunt atribuite, prin intermediul argumentului `...`, funcției `runif`.

```
x <- 1:4
lapply(x, runif, min = 0, max = 3)
[[1]]
[1] 0.03443616

[[2]]
[1] 1.267361 1.365441

[[3]]
[1] 1.8084700 2.1902665 0.4139585

[[4]]
[1] 1.5924650 0.7355067 2.1483841 1.6082945
```

## 1.2 `sapply()`

Funcția `sapply()` are un comportament similar cu `lapply()` prin faptul că funcția `sapply()` apelează intern `lapply()` pentru valorile de input, după care evaluează:

- dacă rezultatul este o listă în care fiecare element este de lungime 1, atunci întoarce un vector
- dacă rezultatul este o listă în care fiecare element este un vector de aceeași lungime (>1), se întoarce o matrice
- în caz contrar se întoarce o listă.

Considerăm exemplul de mai sus

```
set.seed(222)
x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1), d = rnorm(100, 5))
sapply(x, mean)
        a         b         c         d 
2.5000000 0.1996044 0.7881026 5.0641876 
```

## 1.3 `split()`

Funcția `split()` primește ca argument un vector sau o listă (sau un data.frame) și împarte datele în grupuri determinate de o variabilă de tip factor (sau o listă de factor).

Argumentele aceste funcții sunt

```
str(split)
function (x, f, drop = FALSE, ...)
```

unde

- `x` este un vector, o listă sau un data.frame
- `f` este un factor sau o listă de factori

Considerăm următorul exemplu în care generăm un vector de date și îl împărțim după o variabilă de tip factor creată cu ajutorul funcției `gl()` (*generate levels*).

```
x <- c(rnorm(10), runif(10), rnorm(10, 1))
f <- gl(3, 10)
split(x, f)
```

```
$`1`
 [1] -2.27414224 -0.11266780  0.61308167  0.07733545  0.57137727
 [6]  0.11672493 -0.95685256 -1.90008460 -1.48972089  0.55925676

$`2`
 [1] 0.91159086 0.03291829 0.78368939 0.11852882 0.64443831 0.78790988
 [7] 0.82451477 0.05642366 0.65075027 0.95426854

$`3`
 [1]  2.6666242  2.6634334  1.8106280 -0.7837308  1.6575684  0.1546575
 [7]  0.4930056 -0.9031544  2.4042311  1.4106863
```

Putem folosi funcția `split` și în conjuncție cu funcția `lapply` (atunci când vrem să aplicăm o funcție `FUN` pe grupuri de date).

```
lapply(split(x, f), mean)
$`1`
[1] -0.4795692

$`2`
[1] 0.5765033

$`3`
[1] 1.157395
```

## 1.4 `tapply()`

Funcția `tapply()` este folosită pentru aplicarea unei funcții `FUN` pe submulțimile unui vector și poate fi văzută ca o combinație între `split()` și `sapply()`, dar doar pentru vectori.

```
str(tapply)
function (X, INDEX, FUN = NULL, ..., default = NA, simplify = TRUE)
```

Argumentele acestei funcții sunt date de următorul tabel:

Table 1: Argumentele functiei tapply

| Argument | Descriere |
|---|---|
| `X` | un vector |
| `INDEX` | este o variabilă de tip factor sau o listă de factori |
| `FUN` | o funcție ce urmează să fie aplicată |
| `...` | argumente ce vor fi atribuite funcției `FUN` |
| `simplify` | dacă vrem să simplificăm rezultatul |

Următorul exemplu calculează media după fiecare grupă determinată de o variabilă de tip factor a unui vector numeric.

```
x <- c(rnorm(10), runif(10), rnorm(10, 1))
f <- gl(3, 10)
f
 [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3
Levels: 1 2 3
tapply(x, f, mean)
```

```
           1            2            3 
-0.0007774025  0.3736457792  0.5789436983 
```

Putem să aplicăm și funcții care întorc mai mult de un rezultat. În această situație rezultatul nu poate fi simplificat:

```
tapply(x, f, range)
$`1`
[1] -2.1904113  0.9249901

$`2`
[1] 0.004445296 0.998309704

$`3`
[1] -0.3379675  1.9327099
```

### 1.5 apply()

Funcția `apply()` este folosită cu precădere pentru a aplica o funcție liniilor și coloanelor unei matrice (care este un `array` bidimensional). Cu toate acestea poate fi folosită pe tablouri multidimensionale (`array`) în general. Folosirea funcției `apply()` nu este mai rapidă decât scrierea unei bucle `for`, dar este mai compactă.

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

Argumentele funcției `apply()` sunt

- `X` un tablou multidimensional
- `MARGIN` este un vector numeric care indică dimensiunea sau dimensiunile după care se va aplica funcția
- `FUN` este o funcție ce urmează să fie aplicată
- `...` alte argumente penru funcția`FUN`

Considerăm următorul exemplu în care calculăm media pe coloane într-o matrice

```
x <- matrix(rnorm(200), 20, 10)
apply(x, 2, mean)  ## media fiecarei coloane
 [1]  3.745002e-02  1.857656e-01 -2.413659e-01 -2.093141e-01 -2.562272e-01
 [6]  8.986712e-05  7.444137e-02 -7.460941e-03  6.275282e-02  9.801550e-02
```

precum și media după fiecare linie

```
apply(x, 1, sum)   ## media fiecarei linii
 [1]  2.76179139  2.53107681  0.87923177  1.80480589  0.98225832
 [6] -3.06148753 -1.40358820 -0.65969812 -1.63717046 -0.29330726
[11] -2.41486442 -3.15698523  2.27126822 -3.88290287 -3.15595194
[16]  5.41211963  2.32985530 -3.05330574 -0.02110926 -1.34909559
```

## 2 Repartiții și elemente aleatoare în R

R pune la disploziție majoritatea repartițiilor uzuale. Tabelul de mai jos prezintă numele și parametrii acestora:

Table 2: Numele si parametrii repartitiilor uzuale in R

| Repartiția | Nume | Parametrii | Valori prestabilite |
|---|---|---|---|
| Beta | `beta` | `shape1`, `shape2` | |
| Binomial | `binom` | `size`, `prob` | |
| Cauchy | `cauchy` | `location`, `scale` | `location = 0`, `scale = 1` |
| Chi-Squared | `chisq` | `df` | |
| Exponential | `exp` | `rate` (=1/mean) | `rate = 1` |
| Fisher | `f` | `df1`, `df2` | |
| Gamma | `gamma` | `shape`, `rate` (=1/scale) | `rate = 1` |
| Hypergeometric | `hyper` | `m`, `n`, `k` | |
| Log-Normal | `lnorm` | `mean`, `sd` | `mean = 0`, `sd = 1` |
| Logistic | `logis` | `location`, `scale` | `location = 0`, `scale = 1` |
| Normal | `norm` | `mean`, `sd` | `mean = 0`, `sd = 1` |
| Poisson | `pois` | `lambda` | |
| Student | `t` | `df` | |
| Uniform | `unif` | `min`, `max` | `min = 0`, `max = 1` |
| Weibull | `weibull` | `shape` | |

Pentru fiecare repartiție, există patru comenzi în R prefixate cu literele `d`, `p`, `q` și `r` și urmate de numele repartiției (coloana a 2-a). De exemplu `dnorm`, `pnorm`, `qnorm` și `rnorm` sunt comenzile corespunzătoare repartiției normale pe când `dunif`, `punif`, `qunif` și `runif` sunt cele corespunzătoare repartiției uniforme.

- `dname`: calculează densitatea atunci când vorbim de o variabilă continue sau funcția de masă atunci când avem o repartiție discretă ($\mathbb{P}(X = k)$)
- `pname`: calculează funcția de repartiție, i.e. $F(x) = \mathbb{P}(X \leq x)$
- `qname`: reprezintă funcția cuantilă, cu alte cuvinte valoarea pentru care funcția de repartiție are o anumită probabilitate; în cazul continuu, dacă `pname(x) = p` atunci `qname(p) = x` iar în cazul discret întoarce cel mai mic întreg $u$ pentru care $\mathbb{P}(X \leq u) \geq p$.
- `rname`: generează observații independente din repartiția dată

Avem următoarele exemple:

```
qnorm(0.975)
[1] 1.959964
pnorm(1.96)
[1] 0.9750021
rnorm(5)
[1] 0.4304737 0.8405027 1.9550682 1.6208507 2.1059503

x = seq(-1, 1, 0.25)
dnorm(x)
[1] 0.2419707 0.3011374 0.3520653 0.3866681 0.3989423 0.3866681 0.3520653
[8] 0.3011374 0.2419707
rnorm(3, 5, 0.5)
[1] 5.327249 4.728878 5.773167

dunif(x)
[1] 0 0 0 0 1 1 1 1 1
runif(3)
[1] 0.6353840 0.8470974 0.0672359
```

# 3 Exerciții pregătitoare

## 3.1 Aruncarea cu banul

În acest exemplu vrem să simulăm aruncarea unei monede (echilibrate) folosind funcția `sample()`. Această funcție permite extragerea, cu sau fără întoarcere (`replace = TRUE` sau `replace = FALSE` - aceasta este valoarea prestabilită), a unui eșantion de volum dat (`size`) dintr-o mulțime de elemente `x`.

Spre exemplu dacă vrem să simulăm 10 aruncări cu banul atunci apelăm:

```
sample(c("H", "T"), 10, replace = TRUE)
 [1] "T" "T" "T" "T" "T" "T" "T" "H" "H" "T"
```

Pentru a estima probabilitatea de apariției a stemei (`H`) repetăm aruncarea cu banul de 10000 de ori și calculăm raportul dintre numărul de apariții ale evenimentului $A = \{H\}$ și numărul total de aruncări:

```
# atunci cand moneda este echilibrata
a = sample(c("H","T"), 10000, replace = TRUE)
p = sum(a == "H")/length(a)
p
[1] 0.5073
```

și pentru cazul în care moneda nu este echilibrată

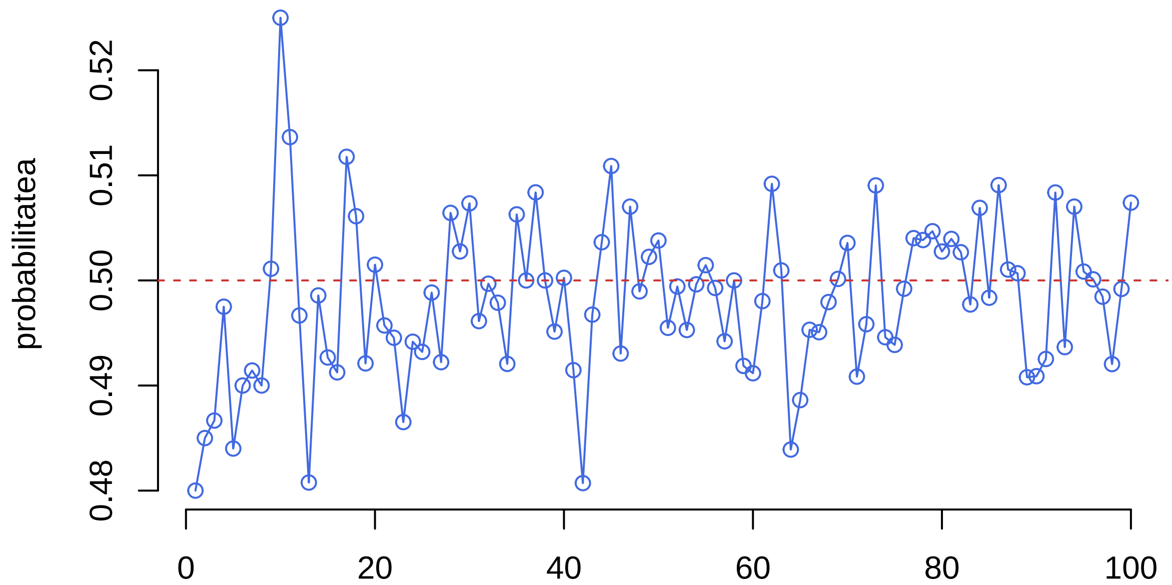
```
a = sample(c("H","T"), 10000, replace = TRUE, prob = c(0.2, 0.8))
p = sum(a == "H")/length(a)
p
[1] 0.2012
```

Putem vedea cum evoluează această probabilitatea în funcție de numărul de repetări

```
y = rep(0,100)

for (i in 1:100){
  a = sample(c("H","T"), i*100, replace = TRUE)
  y[i] = sum(a == "H")/length(a)
}

plot(1:100, y, type = "o", col = "royalblue", bty = "n",
     xlab ="", ylab = "probabilitatea")
abline(h = 0.5, lty = 2, col = "brown3")
```




### 3.2 Jocul de loto

Construiți în R o funcție care să simuleze jocul de loto 6/49. Acest joc consistă din extragerea aleatoare a 6 numere dintr-o urnă cu 49 de numere posibile, fără întoarcere. Fiecare extragere se face de manieră uniformă din numerele rămase în urnă (la a i-a extragere fiecare bilă din urnă are aceeași șansă să fie extrasă). De exemplu putem avea următorul rezultat: $10, 27, 3, 45, 12, 24$.

**Notă**: Funcția `sample()` poate face această operație, ceea ce se cere este de a crea voi o funcție care să implementeze jocul fără a folosi funcția *sample*. Binențeles că puteți folosi funcții precum: `runif` , `floor`, `choose`, etc.

Începem prin a construi o funcție care ne permite generarea unei variabile aleatoare uniform repartizate pe mulțimea $\{1, 2, \ldots, n\}$ (această funcție este cea care simulează procesul de extragere de la fiecare pas):

```
myintunif = function(n){
  # dunctia care genereaza un numar uniform intre 1 si n
  r = n*runif(1)
  u = floor(r)+1
  return(u)
}
```

Funcția care realizează extragerea fără întoarcere a $k$ numere aleatoare din $n$, este:

```
myrandsample=function(n,k){
  #
```

```
  x = 1:n
  q = rep(0,k)

  for(i in 1:k){
    l = length(x)
    u = myintunif(l)
    q[i] = x[u]
    x = x[x!=q[i]]
  }
  return(q)
}
```

Pentru a vedea ce face această funcție putem scrie:

```
n = 49
k = 6

myrandsample(n,k)
[1]  3 16 12 48 23 32
```

## 3.3 Generarea variabilelor aleatoare discrete

În acest exercițiu ne propunem să definim p funcție `rand_sample(n,x,p)` care permite generarea a $n$ observații dintr-o mulțime $x$ (vector numeric sau de caractere) cu probabilitatea $p$ pe $x$ (un vector de aceeași lungime ca $x$).

Funcția se poate construi sub forma următoare:

```
rand_sample = function(n,x,p){
  # n - numarul de observatii
  # x - multimea de valori
  # p - vectorul de probabilitati

  out = c()

  ind = 1:length(x)
  cs = cumsum(p)

  if (length(x)!=length(p)){
    return(print('Cei doi vectori ar trebui sa fie de aceeasi lungime !'))
  }

  for (i in 1:n){
    r = runif(1)

    m = min(ind[r<=cs])
    out = c(out,x[m])
  }

  return(out)
}
```

Pentru a testa această funcție să considerăm două exemple:

1. în acest caz: $n = 10$, $x = [1, 2, 3]$ și $p = [0.2, 0.3, 0.5]$

```r
rand_sample(10,c(1,2,3),c(0.2,0.3,0.5))
 [1] 2 3 3 1 3 3 1 2 3 1
```

2. în acest caz: $n = 15$, $x = [a, b, c, d]$ și $p = [0.15, 0.35, 0.15, 0.45]$

```r
rand_sample(15,c('a','b','c','d'),c(0.15,0.35,0.15,0.45))
 [1] "b" "d" "d" "d" "a" "b" "d" "a" "d" "c" "d" "d" "b" "b" "d"
```

# 4 Aplicația 1: Verificarea egalității a două polinoame

În această secțiune ne propunem să abordăm următoarea problemă:

> Având date două polinoame $F(x)$ și $G(x)$, cu $F(x) = \prod_{i=1}^{d}(x - a_i)$ și $G(x)$ dat sub forma canonică ($\sum_{i=0}^{d} c_i x^i$), vrem să verificăm dacă are loc identitatea
>
> $$F(x) \stackrel{?}{\equiv} G(x).$$

Știm că două polinoame sunt egale atunci când coeficienții lor în descompunerea canonică sunt egali. Observăm că dacă am transforma polinonum $F(x)$ la forma sa canonică prin înmulțirea consecutivă a monomului $i$ cu produsul primelor $i-1$ monoame atunci am avea nevoie de $\Theta(d^2)$ operații. În cele ce urmează presupunem că fiecare operație de înmulțire se poate face în timp constant, ceea ce nu corespunde în totalitate cu realitatea în special în cazurile în care vrem să înmulțim coeficienți mari.

Ne propunem să construim un algoritm randomizat care să verifice această egalitate într-un număr mai mic de operații ($O(d)$ operații). Să presupunem că $d$ este gradul maxim al lui $x$ (exponentul cel mai mare) în $F(x)$ și $G(x)$. Algoritmul poate fi descris astfel: alegem uniform un număr $r$ din mulțimea $\{1, 2, \dots, 100d\}$ (prin uniform înțelegem că cele $100d$ numere au aceeași șansă să fie alese) și calculăm valorile lui $F(r)$ și $G(r)$. Dacă $F(r) \neq G(r)$ atunci algoritmul întoarce că cele două polinoame nu sunt egale iar dacă $F(r) = G(r)$ atunci algoritmul întoarce că cele două polinoame sunt egale. Algoritmul poate greși doar dacă cele două polinoame nu sunt egale dar $F(r) = G(r)$. Vrem să evaluăm această probabilitate.

Experimentul nostru poate fi modelat cu ajutorul tripletului $(\Omega, \mathcal{F}, \mathbb{P})$ unde $\Omega = \{1, 2, \dots, 100d\}$, $\mathcal{F} = \mathcal{P}(\Omega)$ iar $\mathbb{P}$ este echirepartiția pe $\Omega$. Fie $E$ evenimentul că algoritmul greșește, acest eveniment se realizează doar dacă numărul aleator $r$ este o rădăcină a polinomului $F(x) - G(x)$, de grad cel mult $d$. Cum din *Teorema Fundamentală a Algebrei* acest polinom nu poate avea mai mult de $d$ rădăcini rezultă că

$$\mathbb{P}(\text{algoritmul greseste}) = \mathbb{P}(E) \leq \frac{d}{100d} = \frac{1}{100}.$$

Putem să îmbunătățim această probabilitate? O variantă ar fi să mărim spațiul stărilor la $\Omega = \{1, 2, \dots, 1000d\}$ și atunci șansa ca algoritmul să greșească ar fi de $\frac{1}{1000}$. O altă variantă ar fi să repetăm procedeul de mai multe ori, iar în această situație algoritmul ar fi eronat doar dacă ar întoarce că cele două polinoame sunt egale când în realitate ele nu sunt. Atunci când repetăm procesul, alegerea numărului aleator $r$ se poate face în două moduri diferite: cu întoarcere (nu ținem cont de numerele ieșite) sau fără întoarcere (ținem cont de numerele ieșite).

Dacă luăm $E_i$ evenimentul prin care la a i-a rulare a algoritmului am extras o rădăcină $r_i$ astfel încât $F(r_i) = G(r_i)$, atunci probabilitatea ca algoritmul să întoarcă un răspuns greșit după $k$ repetări, este

$$\mathbb{P}(E_1 \cap E_2 \cap \dots \cap E_k).$$

În cazul în care alegerea se face cu întoarcere (evenimentele sunt independente) obținem

$$\mathbb{P}(E_1 \cap E_2 \cap \cdots \cap E_k) = \prod_{i=1}^{k} \mathbb{P}(E_i) \leq \prod_{i=1}^{k} \frac{d}{100d} = \left(\frac{1}{100}\right)^k,$$

iar dacă extragerea s-a făcut fără întoarcere (evenimentele nu mai sunt independente), atunci folosind regula de multiplicare avem

$$\begin{aligned}\mathbb{P}(E_1 \cap E_2 \cap \cdots \cap E_k) &= \mathbb{P}(E_1)\mathbb{P}(E_2|E_1)\cdots\mathbb{P}(E_k|E_1 \cap \cdots \cap E_{k-1})\\ &\leq \prod_{i=1}^{k} \frac{d-(i-1)}{100d-(i-1)} \leq \left(\frac{1}{100}\right)^k.\end{aligned}$$

```
Fx = function(x){
  return((x+1)*(x-2)*(x+3)*(x-4)*(x+5)*(x-6))
}

Gx = function(x){
  return(x^6 - 7*x^3 + 25)
}

Gx2 = function(x){
  return(x^6 - 3*x^5 - 41*x^4 + 87*x^3 + 400*x^2 - 444*x - 720)
}

comparePols = function(Fx, Gx){
  k = 3 # repetam algoritmul de 3 ori
  d = 6 # gradul polinomului

  for (i in 1:k){
    r = floor(100*d*runif(1)) + 1

    f1 = Fx(r)
    g1 = Gx(r)

    if (f1!=g1){
      return(cat("Cele doua polinoame sunt diferite ! \nPentru r =",
                 r, "avem ca F(r)!=G(r) (", f1, "!=", g1, ")"))
    }

  }

  return(cat("Polinoamele sunt egale. Eroarea este de", 100^(-k)))
}

comparePols(Fx, Gx)
Cele doua polinoame sunt diferite !
Pentru r = 352 avem ca F(r)!=G(r) ( 1.885362e+15 != 1.902199e+15 )

comparePols(Fx, Gx2)
Polinoamele sunt egale. Eroarea este de 1e-06
```

# 5 Aplicația 2: Verificarea produsului a două matrice

În cele ce urmează vom prezenta o altă aplicație în care algoritmul randomizat este mult mai eficient decât orice algoritm determinist cunoscut până în acest moment.

Să presupunem că avem trei matrice pătratice $\mathbf{A}$, $\mathbf{B}$ și $\mathbf{C}$ de dimensiune $n \times n$. Pentru simplitate considerăm că avem de-a face cu matrice cu elemente de 0 și 1 iar operațiile se fac $\mod 2$. Vrem să construim un algoritm randomizat care să verifice egalitatea:

$$\mathbf{A} \cdot \mathbf{B} = \mathbf{C} \pmod 2.$$

O modalitate ar fi să calculăm elementele matricii $\mathbf{C}' = \mathbf{A} \cdot \mathbf{B}$, folosind relația $C'_{i,j} = \sum_{k=1}^{n} A_{i,k} B_{k,j}$, și să le comparăm cu elementele matricii $\mathbf{C}$. Metoda *naivă* de multiplicare a celor două matrice necesită cel mult $O(n^3)$ operații. Știm că există algoritmi (netriviali) care permit multiplicarea matricelor într-un număr mai mic de operații, ca de exemplu:

(Strassen 1969) Este posibil să înmulțim două matrice în aproape $n^{\log_2 7} \approx n^{2.81}$ operații.

iar o versiune îmbunătățită a algoritmului lui Strassen (și cea care deținea recordul până în 2014[1])

(Coppersmith-Winograd 1987) Este posibil să înmulțim două matrice în aproape $n^{2.376}$ operații.

Noi nu vom vorbi despre algoritmi de multiplicare a două matrice, ci de algoritmi de verificare a acestei operații. Pentru aceasta vom considera un algoritm randomizat care verifică înmulțirea în aproximativ $n^2$ operații ($O(n^2)$) cu precizarea că acest algoritm poate conduce la un răspuns eronat.

(Freivalds 1979) Există un algoritm probabilist care poate verifica dacă $\mathbf{A} \cdot \mathbf{B} = \mathbf{C} \pmod 2$ în $O(n^2)$ operații și având o eroare de $2^{-200}$.

Algoritmul lui de bază a lui Freivalds este următorul:

1. Alegem uniform un vector $\mathbf{r} = (\mathbf{r_1}, \mathbf{r_2}, \dots, \mathbf{r_n}) \in \{\mathbf{0}, \mathbf{1}\}^{\mathbf{n}}$ (unde prin ales uniform înțelegem că fiecare $r_i$ este ales de manieră independentă cu probabilitatea de 0.5 să ia valoarea 0 sau 1[2])
2. Calculăm $y = \mathbf{ABr}$ și $z = \mathbf{Cr}$. Dacă $y = z$ atunci algoritmul întoarce *multiplicarea este corectă* altfel *multiplicarea matricelor este eronată.*

Observăm că acest algoritm necesită trei operații de multiplicare între o matrice și un vector, prin urmare acesta necesită $O(n^2)$ pași. De asemenea, remarcăm că algoritmul întoarce un răspuns eronat atunci când $\mathbf{A} \cdot \mathbf{B} \neq \mathbf{C}$ dar $\mathbf{ABr} = \mathbf{Cr}$. Probabilitatea ca algoritmul să întoarcă un raspuns eronat verifică

$$\mathbb{P}(\text{algoritmul intoarce raspuns gresit}) = \mathbb{P}(\mathbf{ABr} = \mathbf{Cr}) \leq \frac{\mathbf{1}}{\mathbf{2}}.$$

Pentru a verifica acest rezultat, fie $\mathbf{D} = \mathbf{A} \cdot \mathbf{B} - \mathbf{C} \neq \mathbf{0}$. Evenimentul $\{\mathbf{ABr} = \mathbf{Cr}\}$ implică $\mathbf{Dr} = \mathbf{0}$ și cum $\mathbf{D} \neq \mathbf{0}$ putem presupune că elementul $d_{1,1} \neq 0$. Deoarece $\mathbf{Dr} = \mathbf{0}$ rezultă că $\sum_{j=1}^{n} d_{1,j} r_j = 0$ sau, echivalent

[1] Actualul record este $O(n^{2.3728})$ dat de Vassilevska Williams
[2] Putem să ne imaginăm că aruncăm cu un ban echilibrat de $n$ ori și convertim capul în 1 și pajura în 0.

$$r_1 = -\frac{\sum_{j=2}^{n} d_{1,j} r_j}{d_{1,1}}.$$

Avem

$$\begin{aligned}
\mathbb{P}(\mathbf{ABr} = \mathbf{Cr}) &= \sum_{(x_2,\cdots,x_n)\in\{0,1\}^n} \mathbb{P}\left(\{\mathbf{ABr} = \mathbf{Cr}\} \cap \{(\mathbf{r_2},\cdots,\mathbf{r_n}) = (\mathbf{x_2},\cdots,\mathbf{x_n})\}\right)\\
&\leq \sum_{(x_2,\cdots,x_n)\in\{0,1\}^n} \mathbb{P}\left(\left\{r_1 = -\frac{\sum_{j=2}^{n} d_{1,j} r_j}{d_{1,1}}\right\} \cap \{(r_2,\cdots,r_n) = (x_2,\cdots,x_n)\}\right)\\
&\leq \sum_{(x_2,\cdots,x_n)\in\{0,1\}^n} \mathbb{P}\left(\left\{r_1 = -\frac{\sum_{j=2}^{n} d_{1,j} r_j}{d_{1,1}}\right\}\right) \mathbb{P}\left(\{(r_2,\cdots,r_n) = (x_2,\cdots,x_n)\}\right)\\
&\leq \sum_{(x_2,\cdots,x_n)\in\{0,1\}^n} \frac{1}{2}\mathbb{P}\left(\{(r_2,\cdots,r_n) = (x_2,\cdots,x_n)\}\right)\\
&= \frac{1}{2}.
\end{aligned}$$

În relațiile de mai sus am folosit faptul că $r_1$ și $(r_2,\cdots,r_n)$ sunt independente.

Pentru a îmbunătății eroarea algoritmului putem să repetăm procedeul de $k$ ori[3] și în acest caz probabilitatea de eroare devine $2^{-k}$ iar numărul de operații $O(kn^2)$.

```
FreivaldsAlg = function(A, B, C){
  # Algoritmul Freivalds
  n = dim(A)[1] # dimensiunea matricelor

  k = ceiling(log(n)) # numarul de repetari ale algoritmului

  for (i in 1:k){
    r = rbinom(n, 1, prob = 0.5)

    y = B%*%r
    y = A%*%y
    y = y%%2

    z = C%*%r
    z = z%%2

    if (any(y != z)){
      return(print("Multiplicarea matricelor este incorecta!"))
    }

  }

  return(cat("Multiplicarea matricelor este corecta!
             \nProbabilitatea de eroare a algoritmului este", 2^{-k}))
}

# Exemplul 1
```

[3] Algoritmul lui Freivalds presupune $k = 200$.

```
set.seed(1234)
A = matrix(rbinom(225, 1, 0.5), nrow = 15)
B = matrix(rbinom(225, 1, 0.5), nrow = 15)

C = A%*%B
C = C%%2

FreivaldsAlg(A, B, C)
Multiplicarea matricelor este corecta!

Probabilitatea de eroare a algoritmului este 0.125

# Exemplul 2
set.seed(5678)
A1 = matrix(rbinom(625, 1, 0.5), nrow = 25)
B1 = matrix(rbinom(625, 1, 0.5), nrow = 25)

C1 = A1%*%t(B1)
C1 = C1%%2

C2 = A1%*%B1
C2 = C2%%2

FreivaldsAlg(A1, B1, C1)
[1] "Multiplicarea matricelor este incorecta!"
FreivaldsAlg(A1, B1, C2)
Multiplicarea matricelor este corecta!

Probabilitatea de eroare a algoritmului este 0.0625

# Exemplul 3
set.seed(5678910)
A3 = matrix(rbinom(1000000, 1, 0.5), nrow = 1000)
B3 = matrix(rbinom(1000000, 1, 0.5), nrow = 1000)

C3 = A3%*%B3
C3 = C3%%2

FreivaldsAlg(A3, B3, C3)
Multiplicarea matricelor este corecta!

Probabilitatea de eroare a algoritmului este 0.0078125
```