# 1 Cerința 5 + Cerința 6

**5)Calculul mediei, dispersiei și a momentelor inițiale și centrate până la ordinul 4(dacă există). Atunci când unul dintre momente nu există, se va afișa un mesaj corespunzător către utilizator.**

**6)Calculul mediei și dispersiei unei variabile aleatoare g(X), unde X are o repartiție continuă cunoscută iar g este o funcție continuă precizată de utilizator.**

Toate funcțiile legate de aceste două cerințe au fost scrise având cazul general ca scop, ceea ce a a avut ca rezultat că am putut folosi aceleași funcții pentru calcularea mediei lui $X$, dar și lui $g(X)$ și chiar cuplurilor de forma $(X, Y)$, unde X și Y sunt două variabile aleatoare continue.

## 1.1 Integrala

Înainte de a începe să prezentăm funcțiile relevante cerințelor, ar fi bine să explicăm ce face mai exact funcția *integrala*, folosită destul de mult de pachetul contRV. Mai întâi, o scurtă descriere a antetului funcției:

| Parametrul | Tipul | Descriere |
|---|---|---|
| X | contRV | Variabila aleatoare pe care vrem să calculăm o integrală |
| dt | integer | Variabila după care vrem să integrăm; este folosită explicit doar pentru determinarea densității marginale, altfel implicit are valoarea 0 și se calculeaza doar integrala densității variabilei aleatoare. Pentru dt = 1 se integrează doar după x, iar pentru dt = 2 doar după y. |

Scopul parametrului *dt* este doar să faciliteze calcularea densităților marginale, altfel poate fi ignorat.

Funcția se uită mai întâi la ce fel de variabilă este X, unidimensională sau bidimensională. Să luăm mai întâi cazul unidimensionalei, care este cel mai ușor:

```
sum <- 0
for (i in X@suport[[1]]) {
  tryCatch(sum <- sum + integrate(Vectorize(X@densitate), i[1], i[2],
      abs.tol = 1.0e-13)$value,
  error= function(err)
  {
    stop("Integrala a esuat.")
  })
}
return (sum)
```

După cum se vede, este o simplă sumă pe intervalele din suport. O problemă pe care am întâmpinat-o în lucrul proiectului a fost fapul că pentru densități care sunt nenule pe intervale mici, funcția *integrate* din R calculează eronat integralele. Astfel, având suportul salvat, putem evita complet această problemă. Valoarea din $abs.tol$ este $1.0e-13$ pentru că am observat că, din cauza erorilor de rotunjire ale float-urilor, $abs.tol = 0$ uneori returnează eroare, deși cu $abs.tol = 1.0e-13$ calculează corect integrala. Este posibil sa existe funcții care chiar și cu valoarea curentă să returneze eroare, dar noi n-am întâmpinat astfel de exemple.

Acum să vedem cazul bidimensional, când vrem să calculăm integrala pe densitatea comună a unui cuplu de variabile aleatoare continue:

```
sum <- 0
for(i in X@suport[[2]])
{
  for(j in X@suport[[1]])
  {
    tryCatch(sum <- sum +
    integrate( function(y) {
      sapply(y, function(y) {
        integrate(function(x) X@densitate(x,y),j[1],j[2])$value
      })
    },i[1],i[2])$value,
    error= function(err)
    {
      stop("Eroare la integrarea densitatii.")
    })
  }
}

return(sum)
```

După cum se vede, calculăm integrala pe $S_X \times S_Y$, unde am notat cu $S_X$ suportul lui $X$ din $(X,Y)$ si $S_Y$ suportul lui $Y$. Motivul pentru care am optat să calculăm pe întreg produsul cartezian este că, în cazul în care $X$ și $Y$ sunt independente, atunci suportul lui $(X,Y)$ va fi chiar $S_X \times S_Y$, fapt ce se vede din formula densității comune $f_{X,Y}(x,y) = f_X(x) \cdot f_Y(y)$. Dacă nu sunt independente, atunci oricum suportul lui $(X,Y)$ va fi sigur inclus în $S_X \times S_Y$, de exemplu, din formula densității marginale $f_X(x) = \displaystyle\int_{-\infty}^{\infty} f_{X,Y}(x,y)\mathrm{d}y$. Atunci, poate vom calcula niște integrale pe intervale în care densitatea comună este 0, dar câștigăm prin faptul că putem folosi același cod pentru două situații diferite, iar timpul pierdut de calcularea unor intervale nule este, în general, mic.

Ultimul caz este cel în care vrem să extragem densitatea marginală. Vom prezenta doar cazul pentru densitatea marginală a lui Y, pentru că la X este identic. Codul:

```
#   Ok, deci ideea care mi a venit aici este sa construiesc un vector
    de functii asa:
#   f_i+1 (y) = (integrala pe [a_i+1,b_i+1]dx) + f_i(y)

funcs <- vector()
factory <- function(i1, i2, pas)
{
```

```
    i1; i2; pas; # pt closure
    if(pas == 1)
    {
      tmpF <-  Vectorize(function(y) {
        sapply(y, function(y) {
          integrate(function(x) X@densitate(x,y),i1,i2)$value
        })
      })
      funcs <<- c(funcs, tmpF)
    }
    else
    {
      tmpF <-  Vectorize(function(y) {
        sapply(y, function(y) {
          integrate(function(x) X@densitate(x,y),i1,i2)$value
        })
      })

      newF <- Vectorize(function(y){
        tmpF(y) + funcs[[pas - 1]](y)
      })
      funcs <<- c(funcs, newF)
    }
  }

  pas <- 0
  for(i in X@suport[[1]])
  {
    pas <- pas + 1
    factory(i[1], i[2], pas)
  }

  return(funcs[[length(funcs)]])
```

Deoarece este posibil ca în formula $f_Y(y) = \int_{-\infty}^{\infty} f_{X,Y}(x,y)\mathrm{d}x$ suportul lui Y să fie spart în intervale, avem deja o problemă la construirea acestei funcții, pentru că va trebui să construim o funcție progresiv, calculând fiecare interval din suport. Soluția pe care am găsit-o este să folosim un vector de funcții, construit dupa următoarele reguli:

- $f_1(y) = \int_{a_1}^{b_1} f_{X,Y}(x,y)\mathrm{d}x$, unde $a_1$ și $b_1$ sunt capetele primului interval din suportul lui Y
- $f_{i+1}(y) = \int_{a_{i+1}}^{b_{i+1}} f_{X,Y}(x,y)\mathrm{d}x + f_i(y)$, unde $a_{i+1}$ și $b_{i+1}$ sunt capetele primului celui de-al $i+1$-lea interval din suportul lui Y, iar $f_i(y)$ este functia rezultată din calcularea primelor $i$ integrale

În final, în $f_n(y)$ o să avem densitatea marginală a lui Y, unde $n$ este numărul de intervale din suport.

Încă o dificultate întâlnită a fost la creearea funcțiilor propriu-zise în R. Dacă nu am fi folosit funcția $factory$ și am fi pus direct codul ei în $for$, am observat că R nu salva corect valoarea lui $pas$ sau intervalele din $i$, ci în schimb punea peste tot ultimul $pas$, respectiv ultimul interval. Din ce am înteles, problema apare din cauza felului în care R implementează scoping-ul variabilelor. În orice caz, soluția noastră este inspirată din urmatorul răspuns: `https://stackoverflow.com/questions/12481404/how-to-create-a-vector-of-functions`.

Sper că am clarificat destul de bine cum funcționează această funcție, de altfel fundamentală pentru implementarea clasei **contRV**, deoarece am încercat să folosim cât mai des funcția aceasta, în loc de *integrate* singură.

## 1.2 Media

Implementarea mediei se bazează pe formula ei obișnuită. Pentru aflarea mediei unei variabile continue $X$, recomandăm folosirea metodei $E(X)$, în loc de apelarea manuală a funcției $media(X)$.

Antetul funcției arată astfel:

| Parametrul | Tipul | Descriere |
|---|---|---|
| X | contRV | Variabila aleatoare căreia vrem să-i determinăm media |

Parametrul poate fi ori unidimensional, ori bidimensional, media este calculată corect în ambele cazuri. Media este implementată astfel:

```
media <- function(X)
{
  subIntegrala <- function(...)
  {
    X@val(...) * X@densitate(...)
  }

  subIntegrala <- Vectorize(subIntegrala)
  tmp <- contRV(densitate = subIntegrala, val = Vectorize(function(...)
      {retval <- 1}), bidimen = X@bidimen, suport = X@suport)

  tryCatch(retval <- integrala(tmp),
  error=function(err){
    stop("Media nu exista")
  }
  )
  return(retval)
}
```

Codul poate arăta ciudat la prima vedere, probabil din cauza variabilei *tmp*.

În primul rând, după cum se vede, am decis să folosim ... ca parametru pentru implementarea functiei „de sub integrală” a mediei, pentru a putea acoperi și cazul unidimensionalei, și bidimensionalei cu o singură funcție. Motivul pentru care am făcut funcția aceasta este și pentru că am considerat că avem un cod mai elegant dacă nu încărcăm câmpul „densitate” din constructorul lui *tmp* cu o funcție definită pe loc.

În al doilea rând, deoarece deja avem funcția *integrala* care știe să calculeze o integrală pe un anumit suport, ar fi bine să o apelăm cumva, decât să rescriem aceiași secvență de cod de mai multe ori. Desigur, dacă am apela $integrala(X)$, nu am obține rezultatul corect, pentru că vrem să integrăm funcția $subIntegrala$, nu densitatea lui $X$. Atunci, am găsit soluția să folosim un fel de pseudo-variabilă, căreia îi dăm ca densitate funcția pe care vrem să o integrăm pentru a putea apela *integrala* pe această pseudo-variabilă. Acest artificiu este folosit de mai multe ori în proiect, deoarece am întâmpinat mai multe situații asemănătoare, în care am fi putut folosi o funcție destinată doar variabilelor continue, pentru a calcula diverse valori.

## 1.3 Dispersia

Dispersia seamănă ca implementare foarte mult cu media, doar că desigur diferă calculele făcute. De asemenea, există o metodă definită pentru dispersie, anume $Var(X)$. Antetul funcției arată astfel:

| Parametrul | Tipul | Descriere |
|---|---|---|
| X | contRV | Variabila aleatoare căreia vrem să-i determinăm dispersia |

```
dispersia <- function(X)
{
  # formula clasica cu integrala

  tryCatch(m <- media(X), warning=function(wr)
  {
    stop("Calcularea dispersiei a esuat, media nu exista")
  })

  xCoef <- function(...)
  {

    X@val(...) - m
  }

  coefRaised <- powF(xCoef, 2)
  coefRaised <- Vectorize(coefRaised)

  subIntegrala <- function(...)
  {
```

```
    coefRaised(...) * X@densitate(...)
  }
  subIntegrala <- Vectorize(subIntegrala)
  tmp <- contRV(densitate = subIntegrala, val = Vectorize(function(...) {retval <- 1}), bidimen = X@bidimen, suport = X@suport)

  tryCatch(retval <- integrala(tmp),
  error= function(err)
  {
    stop("Dispersia nu exista.")
  })
  return(retval)
}
```

Cum spuneam, este calculată în mare ca media, folosind o funcție $subIntegrala$ și un pseudo-contRV $tmp$. Diferența majoră este introducerea funcției $xCoef$, care reprezintă $(x - E[X])$ din formulă. Voi discuta repede și despre funcția $powF$, care este foarte simplă de altfel.

### 1.3.1 powF

Scopul acestei funcții este de a lua funcția $f$ din parametru și de a returna o funcție egală cu $f$ ridicată la o anumită putere, dată de asemenea ca parametru. Antetul:

| Parametrul | Tipul | Descriere |
|---|---|---|
| f | funcție | Funcția pe care vrem să o ridicăm la putere |
| y | integer | Puterea |

Implementarea arată astfel:

```
powF <- function(f, y)
{
  ret <- function(...)
  {
    f(...) ^ y
  }
}
```

Trebuie ținut cont de modul în care R face scoping, astfel că nu putem scrie ceva de genul:

```
  func <- powF(func, 3)

  # Body-ul lui func este acum
  func <- function(...)
  {
    f(...) ^ y # atentie!! f este tot func, deci avem o recursie infinita
  }
```

Se poate evita ușor problema aceasta folosind o funcție $funcRaised$ în care să stocăm $powF(func, 3)$. Asta este și soluția folosită de noi în proiect.

---

Revenind la dispersie, după ce ridicăm $xCoef$ la puterea a 2-a, codul devine aproape identic ca la medie.

## 1.4 Momentul centrat de ordin r

Momentul centrat este implementat într-o măsură asemănătoare. Să vedem mai întâi antetul:

| Parametrul | Tipul | Descriere |
|---|---|---|
| X | contRV | Variabila aleatoare pe care vrem să calculăm momentul centrat |
| ordin | integer | Ordinul |

Implementarea este aproape identică cu dispersia, doar că acum ridicăm $xCoef$ la $r$, în loc de 2. O modificare pe care am făcut-o a fost că, dacă $r < 3$, putem calcula momentul centrat în funcție de funcțiile deja scris în pachet în modul următor:

```
moment_centrat <- function(X, ordin)
{
    # cazurile triviale
  if(ordin == 0)
  return(1)
  else if(ordin == 1)
  tryCatch({ # trebuie totusi verificat daca E(X) exista, ca altfel nu va da nici macar 0
    media(X)
    return(0)
  }, error= function(err)
  {
    stop(paste("Calcularea momentului centrat de ordin ", ordin, " a esuat, nu exista media."))
  })
  else if(ordin == 2)
  tryCatch({  # X dispersia nu exista, vrem un mesaj specific pt momente
    return(dispersia(X))
  }, error= function(err)
  {
    stop(paste("Calcularea momentului centrat de ordin ", ordin, " a esuat, nu exista dispersie."))
})

  tryCatch(m <- media(X), warning=function(wr)
  {
```

```
    stop(paste("Calcularea momentului centrat de ordin ", ordin, " a esuat"))
  })

  xCoef <- function(...)
  {

    X@val(...) - m
  }

  coefRaised <- powF(xCoef, ordin)
  coefRaised <- Vectorize(coefRaised)

  subIntegrala <- function(...)
  {
    coefRaised(...) * X@densitate(...)
  }
  subIntegrala <- Vectorize(subIntegrala)
  tmp <- contRV(densitate = subIntegrala, val = Vectorize(function(...){retval <- 1}), bidimen = X@bidimen, suport = X@suport)

  tryCatch(retval <- integrala(tmp),
  error= function(err)
  {
    stop(paste("Momentul centrat de ordin ", ordin, " nu exista."))
  })
  return(retval)
}
```

În cazurile "triviale", adică cele în care ne putem folosi de funcții deja scrise, am optat să le apelăm pentru a obține un comportament al funcțiilor cât mai apropiat de cel așteptat. De asemenea, momentele centrate și inițiale se apelează direct cu funcțiile lor, deoarece nu am avut vreo idee de cum să le definim un generic mai convenabil.

## 1.5 Momente inițiale de ordin r

Codul este aproape identic cu cel al momentului centrat, doar ca $xCoef$ va fi doar $X@val(...)$. Antetul:

| Parametrul | Tipul | Descriere |
|---|---|---|
| X | contRV | Variabila aleatoare pe care vrem să calculăm momentul inițial |
| ordin | integer | Ordinul |

```
moment_initial <- function(X, ordin)
{
  # cazurile triviale
  if(ordin == 0)
```

```
  return(1)
  else if(ordin == 1)
  tryCatch({ # trebuie totusi verificat daca E(X) exista, ca altfel nu va da nici macar 0
    return(media(X))
  }, error= function(err)
  {
    stop(paste("Calcularea momentului initial de ordin ", ordin, " a esuat."))
  })


  xCoef <- function(...)
  {

    X@val(...)
  }

  coefRaised <- powF(xCoef, ordin)
  coefRaised <- Vectorize(coefRaised)

  subIntegrala <- function(...)
  {
    coefRaised(...) * X@densitate(...)
  }
  subIntegrala <- Vectorize(subIntegrala)
  tmp <- contRV(densitate = subIntegrala, val = Vectorize(function(...){retval <- 1}), bidimen = X@bidimen, suport = X@suport)

  tryCatch(retval <- integrala(tmp),
  error= function(err)
  {
    stop(paste("Momentul initial de ordin ", ordin, " nu exista."))
  })
  return(retval)
}
```

## 1.6 Implementarea pentru exercițiul 6

Pentru a asigura faptul că funcțiile pot fi folosite și în situații de genul $g(X)$, am stocat repartiția lui X în câmpul *val*. De asemenea, am definit genericul *aplica* care, după cum îi spune și numele, ia o funcție $g$ și o variabilă X și returnează o variabilă X', căreia i-am aplicat asupra repartiției $g$. Antetul lui *aplica* este:

| Parametrul | Tipul | Descriere |
|---|---|---|
| X | contRV | Variabila aleatoare |
| g | funcție | Funcția continuă pe care vrem să o aplicăm |

Deci, dacă vrem să aflăm, de exemplu, $E[g(X)]$, putem scrie $E(aplica(X, g))$ și, deoarece toate funcțiile de mai sus apeleaza câmpul *val*, sigur va funcționa. Dacă $g$ returnează efectiv un obiect contRV, putem apela direct $E(g(X))$.

Codul lui *aplica* este:

```
setMethod("aplica", "contRV",
function(object, f){
  retval <- contRV(object@densitate, Vectorize(compunere(f,
      object@val)), object@bidimen, object@suport,
  ref_va_bidimen = object@ref_va_bidimen)
})
```

Iar compunere este definit:

```
compunere <- function(f, g)
{
  function(...) f(g(...))
}
```