Tema 2: Optimización de Funciones

Optimización de Sistemas (4º G. Ing. Software)

Pedro Luis Luque

21 de febrero de 2016

Índice

1 Ejercicios resueltos **1**
1.1 Unidimensional: Método de la búsqueda dicotómica . . . 1
1.2 Unidimensional: Método de la sección áurea (golden ratio) . . . 4
1.3 Unidimensional diferenciable: Método de la bisección . . . 7
1.4 Unidimensional: usando algoritmos ya implementadas en R . . . 9
1.4.1 Unidimensional: optimize . . . 9
1.4.2 Unidimensional: findmins . . . 12
1.4.3 Comentarios: problemas en los extremos del intervalo . . . 12
1.5 Multidimensional no restringida: Método coordenadas cíclicas . . . 13
1.6 Multidimensional no restringida: Método de pasos descendentes . . . 18
1.6.1 Cálculo de derivadas simbólicas con R . . . 21
1.6.2 Cálculo de vectores gradientes y matrices hessianas con cálculo simbólico con R . . . 23
1.7 Multidimensional no restringida: usando algoritmos implementados ya en R . . . 24
1.8 Multidimensional restringida: método de penalizaciones . . . 26

2 Ejercicios propuestos **28**
2.1 Ejercicio 1 . . . 28
2.2 Ejercicio 2 . . . 28
2.3 Ejercicio 3 . . . 28
2.4 Ejercicio 4 . . . 28
2.5 Ejercicio 5 . . . 28
2.6 Ejercicio 6 . . . 28
2.7 Ejercicio 7 . . . 28
2.8 Ejercicio 8 . . . 29
2.9 Ejercicio 9 . . . 29

1. Ejercicios resueltos

1.1. Unidimensional: Método de la búsqueda dicotómica

Codificamos en R el método:

```
optuni_nores_busqdicotomica = function(funx, extizq.ini,extder.ini,
                                       longinc=0.1,epsilon=0.01,report.calculos=T) {
  # funx debe ser estrict. cuasiconvexa
  # longinc, epsilon > 0
 #browser()
 n = 0
 an = extizq.ini
 bn = extder.ini
 mres = NULL
 repeat {
```

```
   lambda.n = ((an+bn)/2) - epsilon
   mu.n = ((an+bn)/2) + epsilon
   f.an = funx(an)
   f.bn = funx(bn)
   f.lambda.n = funx(lambda.n)
   f.mu.n = funx(mu.n)
   if (f.lambda.n<f.mu.n) {
      an1 = an
      bn1 = mu.n
   } else {
      an1 = lambda.n
      bn1 = bn
   }
   vres = c(n+1,an,bn,lambda.n,mu.n,f.an,f.bn,f.lambda.n,f.mu.n,bn-an,an1,bn1)
   if (report.calculos) {
     if (is.null(mres)) {
       mres = vres
     } else {
     mres = rbind(mres,vres)
     }
   }
   if ((bn-an)<=longinc) {
     break
   }
   n = n + 1
   an = an1
   bn = bn1

 }

  if (report.calculos) {
    colnames(mres) = c("n","an","bn","lambdan","mun","fan","fbn","flambdan","fmun",
                       "Amplitud","an1","bn1")
    rownames(mres) = NULL
    dfres = as.data.frame(mres)
  } else {
    names(vres) = c("n","an","bn","lambdan","mun","fan","fbn","flambdan","fmun",
                    "Amplitud","an1","bn1")
dfres = as.data.frame(vres)
 }

 print("Óptimo en el intervalo: (an1,bn1)")
 return(dfres)

}
```

Lo utilizamos sobre la función $f(x) = x^2 + 2x$.

```
# Con información de las iteraciones
funx = function(x) x^2+2*x
df.res = optuni_nores_busqdicotomica(funx,-2,1)
```

[1] "Óptimo en el intervalo: (an1,bn1)"

```
knitr::kable(df.res,digits = 4)
```

n	an	bn	lambdan	mun	fan	fbn	flambdan	fmun	Amplitud	an1	bn1
1	-2.0000	1.0000	-0.5100	-0.4900	0.0000	3.0000	-0.7599	-0.7399	3.0000	-2.0000	-0.4900
2	-2.0000	-0.4900	-1.2550	-1.2350	0.0000	-0.7399	-0.9350	-0.9448	1.5100	-1.2550	-0.4900
3	-1.2550	-0.4900	-0.8825	-0.8625	-0.9350	-0.7399	-0.9862	-0.9811	0.7650	-1.2550	-0.8625
4	-1.2550	-0.8625	-1.0688	-1.0488	-0.9350	-0.9811	-0.9953	-0.9976	0.3925	-1.0688	-0.8625
5	-1.0688	-0.8625	-0.9756	-0.9556	-0.9953	-0.9811	-0.9994	-0.9980	0.2063	-1.0688	-0.9556
6	-1.0688	-0.9556	-1.0222	-1.0022	-0.9953	-0.9980	-0.9995	-1.0000	0.1131	-1.0222	-0.9556
7	-1.0222	-0.9556	-0.9989	-0.9789	-0.9995	-0.9980	-1.0000	-0.9996	0.0666	-1.0222	-0.9789

Ahora sin iteraciones

```
# Sin información de las iteraciones
funx = function(x) x^2+2*x
df.res2 = optuni_nores_busqdicotomica(funx,-2,1,report.calculos = F)
```

[1] "Óptimo en el intervalo: (an1,bn1)"

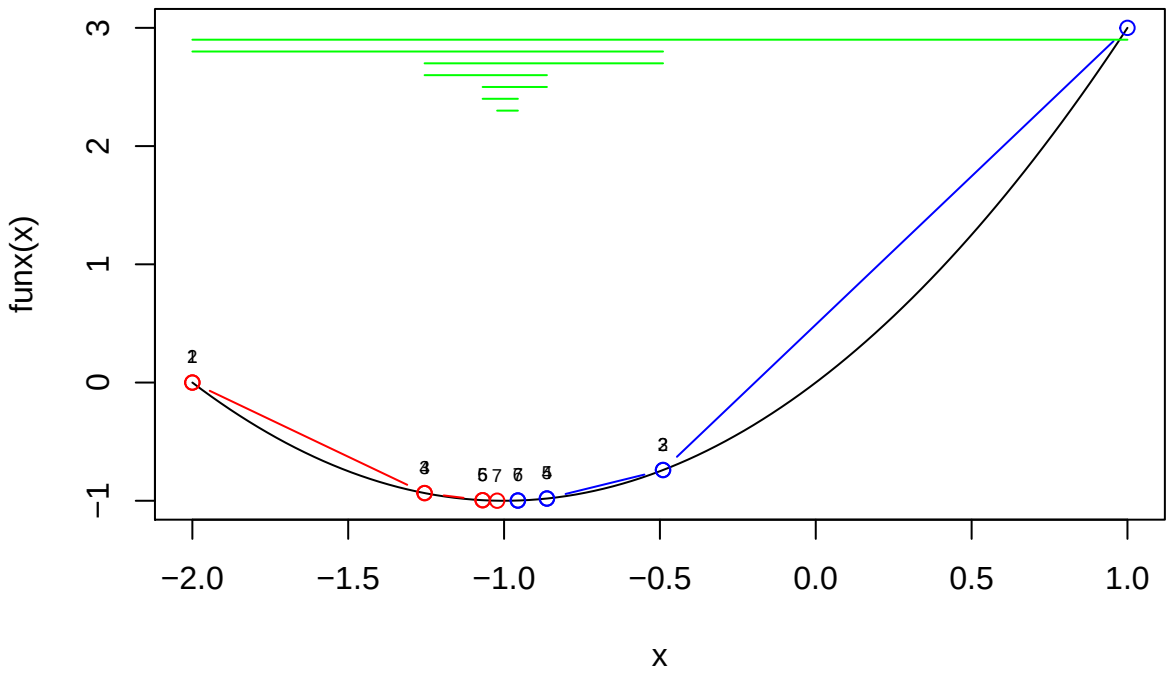
```
knitr::kable(df.res2,digits = 6)
```

	vres
n	7.000000
an	-1.022188
bn	-0.955625
lambdan	-0.998906
mun	-0.978906
fan	-0.999508
fbn	-0.998031
flambdan	-0.999999
fmun	-0.999555
Amplitud	0.066562
an1	-1.022188
bn1	-0.978906

Podemos ver como evoluciona el algoritmo gráficamente:

```
curve(funx,-2,1)

lines(df.res$an,df.res$fan,type="b",col="red")
text(df.res$an,df.res$fan,labels=1:length(df.res$an),pos=3,cex=0.6)
lines(df.res$bn,df.res$fbn,type="b",col="blue")
text(df.res$bn,df.res$fbn,labels=1:length(df.res$an),pos=3,cex=0.6)
cord.y = 3
inc.y = 0.1
for (i in 1:length(df.res$an)) {
  segments(df.res$an[i],cord.y-i*inc.y,df.res$bn[i],cord.y-i*inc.y,col="green")
}
```



1.2. Unidimensional: Método de la sección áurea (golden ratio)

Codificamos en R el método:

```
optuni_nores_goldenratio = function(funx, extizq.ini,extder.ini,
                                    longinc=0.1,report.calculos=T) {
  # funx debe ser estrict. cuasiconvexa
  # longinc, epsilon > 0
 #browser()
 #rho = 0.382  # ((3-sqrt(5))/2)  # 0.381966
 rho = ((3-sqrt(5))/2)  # 0.381966
 n = 0
 nval = 0
 a0 = extizq.ini
 b0 = extder.ini
 a1 = a0 + rho * (b0-a0)
 b1 = a0 + (1-rho) * (b0-a0)
 f.a1 = funx(a1)
 f.b1 = funx(b1)
 mres = NULL
 repeat {
   f.a1 = funx(a1)
   f.b1 = funx(b1)
   if (f.a1>f.b1) {
      a2 = a1
      b2 = b0
      a3 = b1
      b3 = a2 + (1-rho)*(b2-a2)
   } else {
      a2 = a0
      b2 = b1
      a3 = a2 + rho*(b2-a2)
      b3 = a1
```

```
   }
   vres = c(n+1,a0,b0,a1,b1,f.a1,f.b1,b0-a0)
   if (report.calculos) {
     if (is.null(mres)) {
       mres = vres
     } else {
     mres = rbind(mres,vres)
     }
   }
   if ((b0-a0)<=longinc) {
     break
   }
   n = n + 1
   nval = nval + 2
   a0 = a2
   b0 = b2
   a1 = a3
   b1 = b3

 }

  if (report.calculos) {
    colnames(mres) = c("n","a0","b0","a1","b1","fa1","fb1","Amplitud")
    rownames(mres) = NULL
    dfres = as.data.frame(mres)
  } else {
    names(vres) = c("n","a0","b0","a1","b1","fa1","fb1","Amplitud")
dfres = as.data.frame(vres)
 }

 print("Óptimo en el intervalo: (a0,b0)")
 return(dfres)

}
```

Lo utilizamos sobre la función $f(x) = x^4 - 14x^3 + 60x^2 - 70x$.

```
# Con información de las iteraciones
funx = function(x) x^4-14*x^3+60*x^2-70*x
df.res = optuni_nores_goldenratio(funx,0,2)
```

[1] "Óptimo en el intervalo: (a0,b0)"

```
knitr::kable(df.res,digits = 6)
```

n	a0	b0	a1	b1	fa1	fb1	Amplitud
1	0.000000	2.000000	0.763932	1.236068	-24.36068	-18.95816	2.000000
2	0.000000	1.236068	0.472136	0.763932	-21.09851	-24.36068	1.236068
3	0.472136	1.236068	0.763932	0.944272	-24.36068	-23.59246	0.763932
4	0.472136	0.944272	0.652476	0.763932	-23.83743	-24.36068	0.472136
5	0.652476	0.944272	0.763932	0.832816	-24.36068	-24.28789	0.291796
6	0.652476	0.832816	0.721360	0.763932	-24.25795	-24.36068	0.180340
7	0.721360	0.832816	0.763932	0.790243	-24.36068	-24.36691	0.111456
8	0.763932	0.832816	0.790243	0.806504	-24.36691	-24.34953	0.068884

Ahora sin iteraciones

```
# Sin información de las iteraciones
funx = function(x) x^4-14*x^3+60*x^2-70*x
df.res2 = optuni_nores_goldenratio(funx,0,2,report.calculos = F)
```

[1] "Óptimo en el intervalo: (a0,b0)"

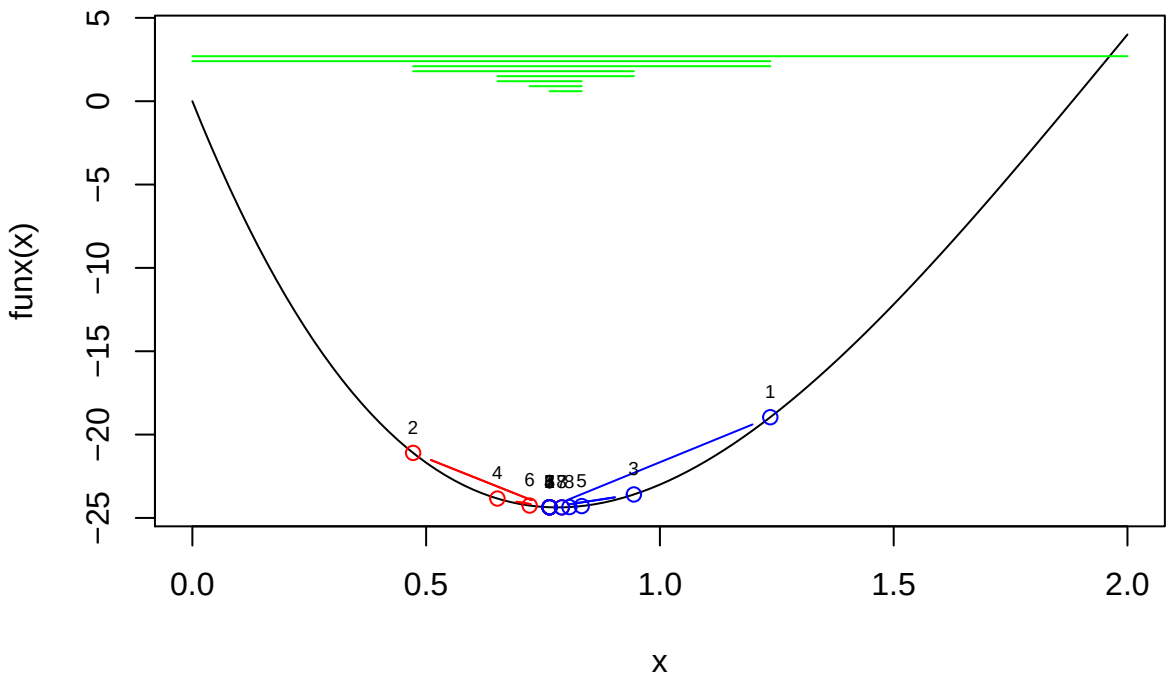
```
knitr::kable(df.res2,digits = 6)
```

	vres
n	8.000000
a0	0.763932
b0	0.832816
a1	0.790243
b1	0.806504
fa1	-24.366907
fb1	-24.349526
Amplitud	0.068884

Podemos ver como evoluciona el algoritmo gráficamente:

```
curve(funx,0,2)

lines(df.res$a1,df.res$fa1,type="b",col="red")
text(df.res$a1,df.res$fa1,labels=1:length(df.res$a1),pos=3,cex=0.6)
lines(df.res$b1,df.res$fb1,type="b",col="blue")
text(df.res$b1,df.res$fb1,labels=1:length(df.res$b1),pos=3,cex=0.6)
cord.y = 3
inc.y = 0.3
for (i in 1:length(df.res$a1)) {
  segments(df.res$a0[i],cord.y-i*inc.y,df.res$b0[i],cord.y-i*inc.y,col="green")
}
```



1.3. Unidimensional diferenciable: Método de la bisección

Codificamos en R el método:

```
optunider_nores_biseccion = function(dfunx, extizq.ini,extder.ini,
                                     longinc=0.1,report.calculos=T) {
  # funx debe ser pseudoconvexa
  # longinc > 0
 #browser()

 n = 0
 an = extizq.ini
 bn = extder.ini
 mres = NULL
 repeat {
   lambda.n = (an+bn)/2
   fd.lambdan = dfunx(lambda.n)
   if (fd.lambdan==0) {
     print("Óptimo en lambdan")
     vres = c(n+1,an,lambda.n,bn,fd.lambdan,bn-an)
     break
   } else if (fd.lambdan>0) {
      an1 = an
      bn1 = lambda.n
   } else {
     an1 = lambda.n
     bn1 = bn
   }
   vres = c(n+1,an,lambda.n,bn,fd.lambdan,bn-an)
   if (report.calculos) {
     if (is.null(mres)) {
       mres = vres
     } else {
```

```
      mres = rbind(mres,vres)
      }
    }
    if ((bn-an)<=longinc) {
      break
    }
    n = n + 1
    an = an1
    bn = bn1
}

  if (report.calculos) {
    colnames(mres) = c("n","a","lambda","b","derflambada","Amplitud")
    rownames(mres) = NULL
    dfres = as.data.frame(mres)
  } else {
    names(vres) = c("n","a","lambda","b","derflambada","Amplitud")
dfres = as.data.frame(vres)
 }

 print("Óptimo en lambda")
 return(dfres)



}
```

Lo utilizamos sobre la función $f(x) = x^2 + 2x$, y con $f'(x) = 2x + 2$.

```
# Con información de las iteraciones
dfunx = function(x) 2*x+2
df.res = optunider_nores_biseccion(dfunx,-2,1)
```

[1] "Óptimo en lambda"

```
knitr::kable(df.res,digits = 6)
```

n	a	lambda	b	derflambada	Amplitud
1	-2.0000	-0.500000	1.00000	1.00000	3.00000
2	-2.0000	-1.250000	-0.50000	-0.50000	1.50000
3	-1.2500	-0.875000	-0.50000	0.25000	0.75000
4	-1.2500	-1.062500	-0.87500	-0.12500	0.37500
5	-1.0625	-0.968750	-0.87500	0.06250	0.18750
6	-1.0625	-1.015625	-0.96875	-0.03125	0.09375

Ahora sin iteraciones

```
# Sin información de las iteraciones
dfunx = function(x) 2*x+2
df.res2 = optunider_nores_biseccion(dfunx,-2,1,report.calculos = F)
```

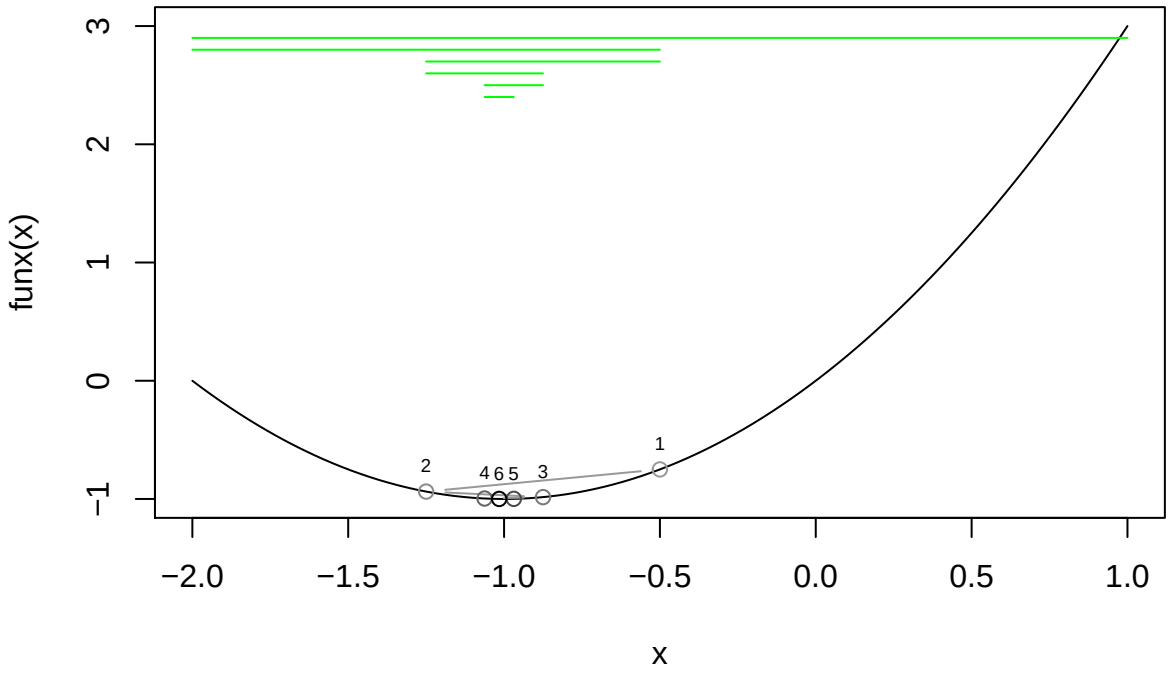
[1] "Óptimo en lambda"

```
knitr::kable(df.res2,digits = 6)
```

	vres
n	6.000000
a	-1.062500
lambda	-1.015625
b	-0.968750
derflambada	-0.031250
Amplitud	0.093750

Podemos ver como evoluciona el algoritmo gráficamente:

```
funx = function(x) x^2+2*x
curve(funx,-2,1)
lines(df.res$lambda,funx(df.res$lambda),type="b",col=rev(grey.colors(length(df.res$lambda),start=0,end = 0
text(df.res$lambda,funx(df.res$lambda),labels=1:length(df.res$lambda),pos=3,cex=0.6)
cord.y = 3
inc.y = 0.1
for (i in 1:length(df.res$a)) {
  segments(df.res$a[i],cord.y-i*inc.y,df.res$b[i],cord.y-i*inc.y,col="green")
}
```



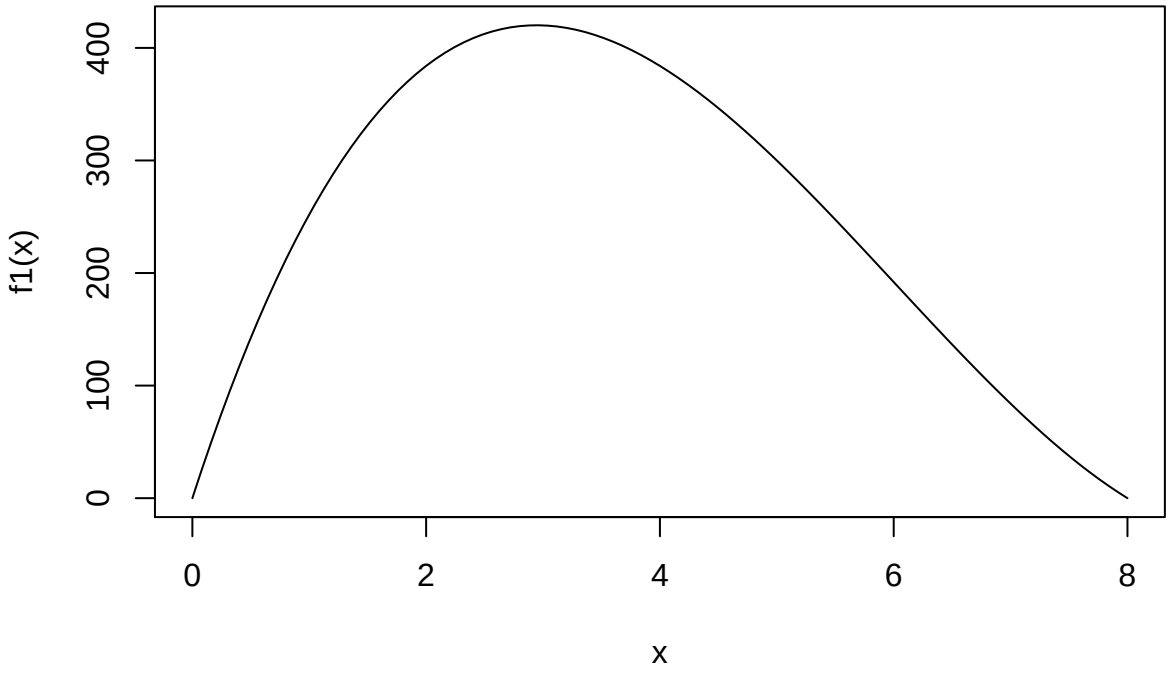
1.4. Unidimensional: usando algoritmos ya implementadas en R

1.4.1. Unidimensional: optimize

```
optimize(function(x) x*(20-2*x)*(16-2*x), c(0,8), maximum=T)
```

```
## $maximum
## [1] 2.94495
##
## $objective
## [1] 420.1104
```

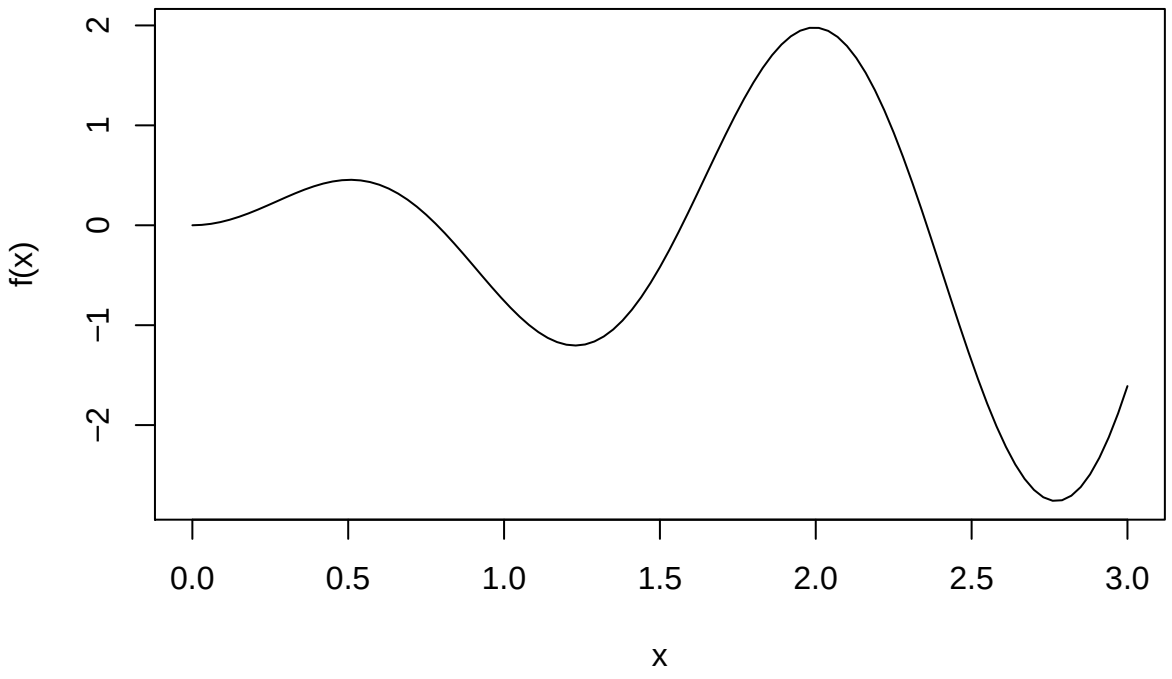
```
f1 = function(x) x*(20-2*x)*(16-2*x)
curve(f1,0,8)
```




```
optimize(f1,c(0,8),maximum = T)
```

```
## $maximum
## [1] 2.94495
##
## $objective
## [1] 420.1104
```

```
f = function(x) x*sin(4*x)
curve(f,0,3)
```




```
optimize(f,c(0,3))
```

```
## $minimum
## [1] 1.228297
##
## $objective
## [1] -1.203617
```

```
optimize(f,c(1.5,3))
```

```
## $minimum
## [1] 2.771403
##
## $objective
## [1] -2.760177
```

```
optimize(f,c(1,3),maximum=TRUE)
```

```
## $maximum
## [1] 1.994684
##
## $objective
## [1] 1.979182
```

```
optimize(function(x) -f(x),c(1,3))
```

```
## $minimum
## [1] 1.994684
##
## $objective
## [1] -1.979182
```

1.4.2. Unidimensional: findmins

El paquete `pracma` contiene la función `findmins()`, la cual encuentra las posiciones de todos los mínimos en el intervalo de búsqueda al dividir en n veces (por defecto n=100) y aplica la optimización a cada intervalo. Para encontrar los mínimos evalua la función en los valores encontrados.

```
# install.packages("pracma") ## para instalar el paquete: pracma
require(pracma)
```

```
## Loading required package: pracma
```

```
f.mins = findmins(f,0,3)
f.mins  # x values at the minima
```

```
## [1] 1.228312 2.771382
```

```
# [1] 1.228312 2.771382
f(f.mins[1:2]) # function evaluated at the minima
```

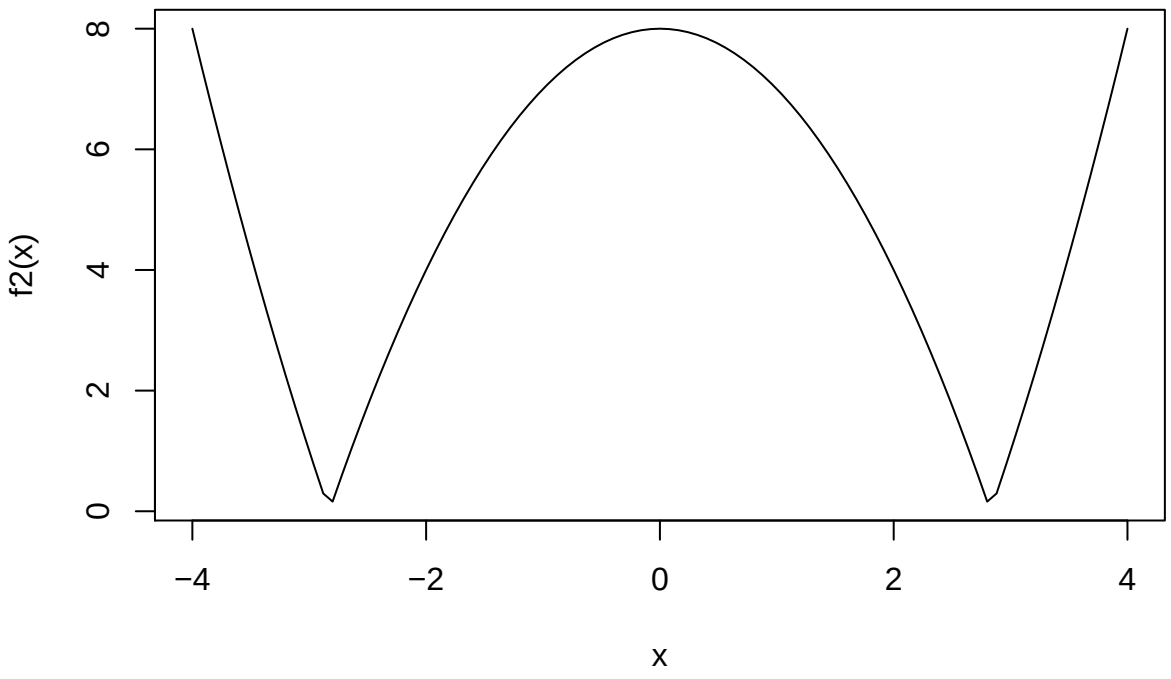
```
## [1] -1.203617 -2.760177
```

```
# [1] -1.203617 -2.760177
```

1.4.3. Comentarios: problemas en los extremos del intervalo

Resolver el problema sobre el intervalo completo (-4,4), `optimize()` encuentra la solución en $x = 0$:

```
f2 = function(x) abs(x^2-8)
curve(f2,-4,4)
```



```r
optimize(f2,c(-4,4),maximum=T)
```

```
## $maximum
## [1] -1.110223e-16
##
## $objective
## [1] 8
```

Considerando intervalos que excluyan el 0:

```r
optimize(f2,c(-4,-2),maximum=T)
```

```
## $maximum
## [1] -3.999959
##
## $objective
## [1] 7.999672
```

```r
optimize(f2,c(2,4),maximum=T)
```

```
## $maximum
## [1] 3.999959
##
## $objective
## [1] 7.999672
```

Sin embargo, `findmins()` nunca considerará como mínimo local los extremos del intervalo, como puede verse al usarlo sobre el ejemplo anterior:

```r
findmins(function(x) -f2(x),-4,4)
```

```
## [1] -1.040834e-17
```

```r
findmins(function(x) -f2(x),-4,-3)
```

```
## NULL
```

1.5. Multidimensional no restringida: Método coordenadas cíclicas

Codificamos en R el método:

```r
# vx.ini = c(0,3)
# funx = function(vx) (vx[1]-2)^4+(vx[1]-2*vx[2])^2
optmul_nores_coordciclicas = function(funx, vx.ini,
                  epsilon=0.01,nummax.iter = 1000,report.calculos=T,
                  cota.inf=-1000,cota.sup=1000) {
 # longinc, epsilon > 0
 #browser()
 mres = NULL
 k = 1
 vyi = vx.ini
```

```
 vxki = vx.ini
 ndimension = length(vx.ini)
 m.vectoresD = diag(x=1,nrow=ndimension,ncol=ndimension)
 # Paso 3
 repeat {
   j = 1

   # Paso 2
   for (j in 1:ndimension) {
     vDj = m.vectoresD[j,]
     ffunxj = function(lambda) funx(vyi+lambda*vDj)
     sol.uni = optimize(ffunxj,c(cota.inf,cota.sup),maximum = F)
     #if (j<ndimension) {
     #uDj1 = m.vectoresD[j+1,]
     vyi1 = vyi + sol.uni$minimum * vDj
     #}
     vres = c(k,j,vyi,sol.uni$minimum,vyi1)
     if (report.calculos) {
       if (is.null(mres)) {
         mres = vres
       } else {
         mres = rbind(mres,vres)
       }
     }

     vyi = vyi1
   }

   vxki1 = vyi1
   norma.l2 = sqrt(sum((vxki1-vxki)^2))
   if ((norma.l2<epsilon) | (k>=nummax.iter)) {
     break
   } else {
     k = k+1
     vyi = vxki1
   }
 } # final repeat

 if (report.calculos) {
   colnames(mres) = c("k","j",paste0("vyi",1:ndimension),"lambda",paste0("vyimas1",1:ndimension))
   rownames(mres) = NULL
   dfres = as.data.frame(mres)
 } else {
   names(vres) = c("k","j",paste0("vyi",1:ndimension),"lambda",paste0("vyimas1",1:ndimension))
   dfres = as.data.frame(vres)
 }

 print("Óptimo en vyimas1")
 return(dfres)

}
```

Lo utilizamos sobre la función $f(x_1, x_2) = (x_1 - 2)^4 + (x_1 - 2x_2)^2$.

```
vx.ini = c(0,3)
funx = function(vx) (vx[1]-2)^4+(vx[1]-2*vx[2])^2
```

```
df.res = optmul_nores_coordciclicas(funx, vx.ini,
                  epsilon=0.0001,nummax.iter = 1000,report.calculos=T)
```

```
## [1] "Óptimo en vyimas1"
```

```
knitr::kable(df.res[c(1:6,nrow(df.res)),],digits = 4)
```

	k	j	vyi1	vyi2	lambda	vyimas11	vyimas12
1	1	1	0.0000	3.0000	3.1282	3.1282	3.0000
2	1	2	3.1282	3.0000	-1.4359	3.1282	1.5641
3	2	1	3.1282	1.5641	-0.4987	2.6294	1.5641
4	2	2	2.6294	1.5641	-0.2494	2.6294	1.3147
5	3	1	2.6294	1.3147	-0.1807	2.4487	1.3147
6	3	2	2.4487	1.3147	-0.0904	2.4487	1.2244
2000	1000	2	2.0199	1.0100	0.0000	2.0199	1.0100

Ahora sin iteraciones

```
df.res2 = optmul_nores_coordciclicas(funx, vx.ini,
                  epsilon=0.0001,nummax.iter = 1000,report.calculos=F)
```

[1] “Óptimo en vyimas1”

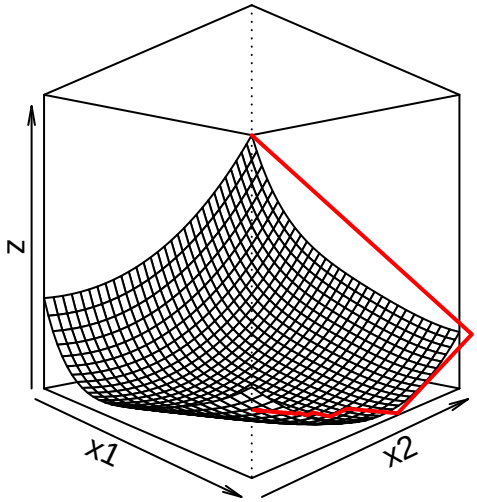
```
knitr::kable(df.res2,digits = 6,format = 'latex')
```

	vres
k	1000.000000
j	2.000000
vyi1	2.019937
vyi2	1.009971
lambda	-0.000003
vyimas11	2.019937
vyimas12	1.009968

Podemos ver como evoluciona el algoritmo gráficamente en 3d, con ayuda de la función `persp()`:

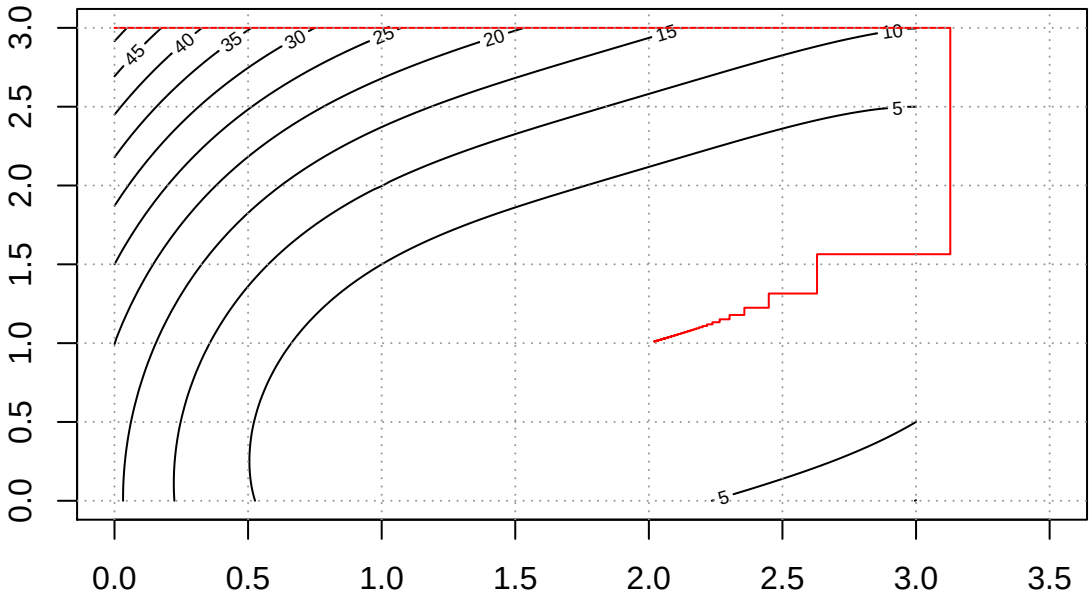
```
#x1 = x2 = seq(0,3,.02)
x1 = x2 = seq(0,3,.1)
z = outer(x1,x2,FUN=function(x1,x2) (  (x1-2)^4+(x1-2*x2)^2   )   )
# z = matrix(0,nrow=length(x1),ncol=length(x2))
# for (i in 1:length(x1)) {
#   for (j in 1:length(x2)) {
#     z[i,j] = funx(c(x1[i],x2[j]))
#   }
# }
res = persp(x1,x2,z,theta=45,phi=0)

ptos.vx = df.res[,3:4]
vz = funx(ptos.vx)
lines(trans3d(x=ptos.vx[,1],y=ptos.vx[,2],z=vz,pmat=res),col="red",lwd=2)
```



También podemos ver como evoluciona el algoritmo gráficamente en 2d sobre las curvas de nivelen, con ayuda de la función `contour()`:

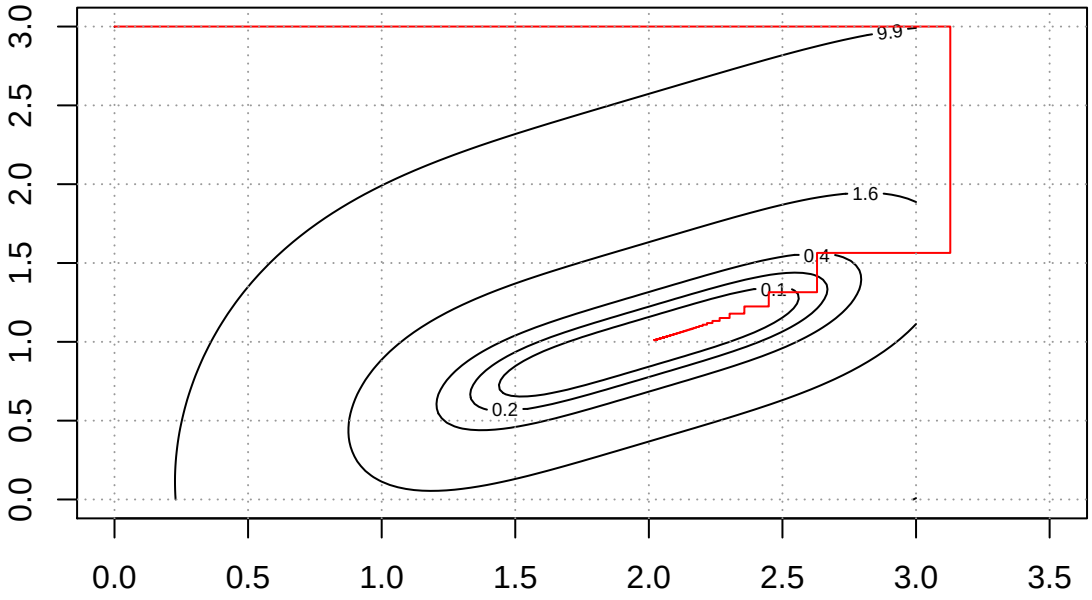
```
ptos.vx = df.res[,3:4]
x <- seq(0, 3, length.out=100)
y <- seq(0, 3, length.out=100)
z <- funx(expand.grid(x, y))
contour(x, y, matrix(z, length(x)),xlim=c(0,3.5))
lines(ptos.vx,type="l",col="red")
grid(col=gray(0.6))
```



En el siguiente código, se eligen los niveles para los que se representan las curvas de nivel:

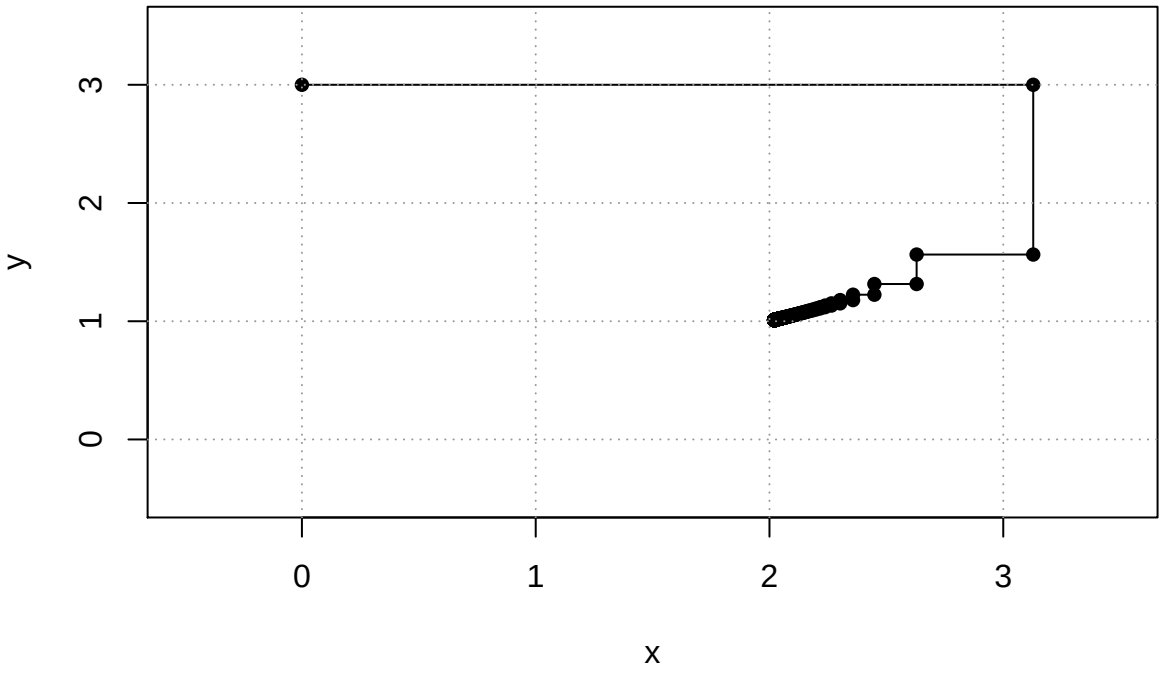
```
ptos.vx = df.res[,3:4]
niveles = unique(round(funx(ptos.vx),1))
x <- seq(0, 3, length.out=100)
y <- seq(0, 3, length.out=100)
z <- funx(expand.grid(x, y))
#contour(x, y, matrix(z, length(x)),nlevels = 20)
```

```
contour(x, y, matrix(z, length(x)),levels = niveles,xlim=c(0,3.5))
lines(ptos.vx,type="l",col="red")
grid(col=gray(0.6))
```



También puede verse como se mueve en el plano XY:

```
ptos.vx = df.res[,3:4]
plot(ptos.vx[,1],ptos.vx[,2],type="l",xlim=c(-0.5,3.5),ylim=c(-0.5,3.5),
     xlab="x",ylab="y")
points(ptos.vx[,1],ptos.vx[,2],pch=16)
grid(col=gray(0.6))
```



1.6. Multidimensional no restringida: Método de pasos descendentes

Codificamos en R el método:

```
optmul_nores_pasosdescendentes = function(funx,gradfunx, vx.ini,
                  epsilon=0.01,nummax.iter = 1000,report.calculos=T,
                  cota.inf=0.00001,cota.sup=1000) {
 # longinc, epsilon > 0
 #browser()
 mres = NULL
 k = 1
 #vyi = vx.ini
 vxki = vx.ini
 ndimension = length(vx.ini)
 gradfxi = gradfunx(vxki)

 repeat {
   vD = gradfxi
   fxi = funx(vxki)
   ffunxj = function(lambda) funx(vxki-lambda*vD)
   sol.uni = optimize(ffunxj,c(cota.inf,cota.sup),maximum = F)
   vxki1 = vxki - sol.uni$minimum * vD
   gradfxi1 = gradfunx(vxki1)
   norma.l2 = sqrt(sum((gradfxi1)^2))
   vres = c(k,vxki,fxi,-gradfxi,sol.uni$minimum,vxki1,norma.l2,norma.l2<epsilon)
   if (report.calculos) {
     if (is.null(mres)) {
       mres = vres
     } else {
       mres = rbind(mres,vres)
     }
   }

   if ((norma.l2<epsilon) | (k>=nummax.iter)) {
     break
   } else {
     k = k+1
     vxki = vxki1
     gradfxi = gradfunx(vxki)

   }

 }# final repeat

 if (report.calculos) {
   colnames(mres) = c("k",paste0("vxki",1:ndimension),"fxi",paste0("vd",1:ndimension),"lambda",
  paste0("vxkimas1",1:ndimension),"NormaGrad","Fin")
   rownames(mres) = NULL
   dfres = as.data.frame(mres)
 } else {
   names(vres) = c("k",paste0("vxki",1:ndimension),"fxi",paste0("vd",1:ndimension),"lambda",
  paste0("vxkimas1",1:ndimension),"NormaGrad","Fin")
   dfres = as.data.frame(vres)
 }

 print("Óptimo en vxkimas1")
```

```
 return(dfres)

}
```

Lo utilizamos sobre la función $f(x_1, x_2) = (x_1 - 2)^4 + (x_1 - 2x_2)^2$.

```
vx.ini = c(0,3)
funx = function(vx) (vx[1]-2)^4+(vx[1]-2*vx[2])^2
gradfunx = function(vx) c(4*(vx[1]-2)^3+2*(vx[1]-2*vx[2]),-4*(vx[1]-2*vx[2]))

df.res = optmul_nores_pasosdescendentes(funx,gradfunx,vx.ini,epsilon=0.0001,nummax.iter = 1000,
        report.calculos=T)
```

```
## [1] "Óptimo en vxkimas1"
```

```
knitr::kable(df.res[c(1:8,nrow(df.res)),],digits = 3)
```

	k	vxki1	vxki2	fxi	vd1	vd2	lambda	vxkimas11	vxkimas12	NormaGrad	Fin
1	1	0.000	3.000	52.000	44.000	-24.000	0.062	2.708	1.523	1.544	0
2	2	2.708	1.523	0.365	-0.740	-1.355	0.231	2.537	1.210	0.969	0
3	3	2.537	1.210	0.096	-0.851	0.465	0.112	2.442	1.262	0.376	0
4	4	2.442	1.262	0.045	-0.180	-0.330	0.268	2.394	1.174	0.382	0
5	5	2.394	1.174	0.026	-0.336	0.183	0.125	2.352	1.197	0.190	0
6	6	2.352	1.197	0.017	-0.091	-0.166	0.280	2.326	1.150	0.218	0
7	7	2.326	1.150	0.012	-0.191	0.105	0.131	2.301	1.164	0.119	0
8	8	2.301	1.164	0.009	-0.057	-0.105	0.287	2.285	1.134	0.145	0
839	839	2.029	1.014	0.000	0.000	0.000	0.145	2.029	1.014	0.000	1

Ahora sin iteraciones

```
df.res2 = optmul_nores_pasosdescendentes(funx,gradfunx,vx.ini,epsilon=0.0001,nummax.iter = 1000,
        report.calculos=F)
```

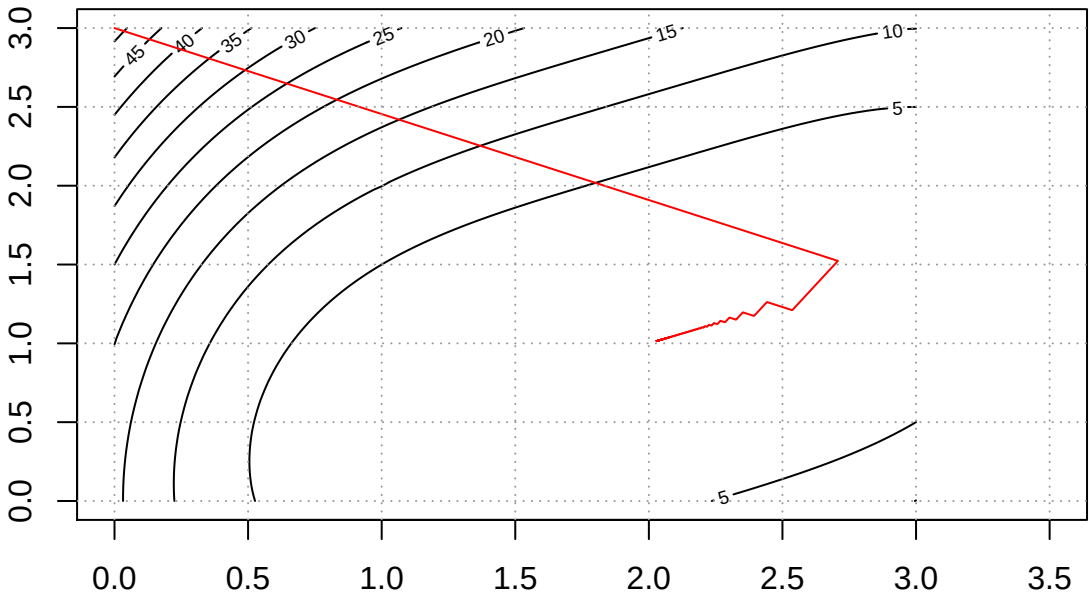
[1] “Óptimo en vxkimas1”

```
knitr::kable(df.res2,digits = 3,format = 'latex')
```

	vres
k	839.000
vxki1	2.029
vxki2	1.014
fxi	0.000
vd1	0.000
vd2	0.000
lambda	0.145
vxkimas11	2.029
vxkimas12	1.014
NormaGrad	0.000
Fin	1.000

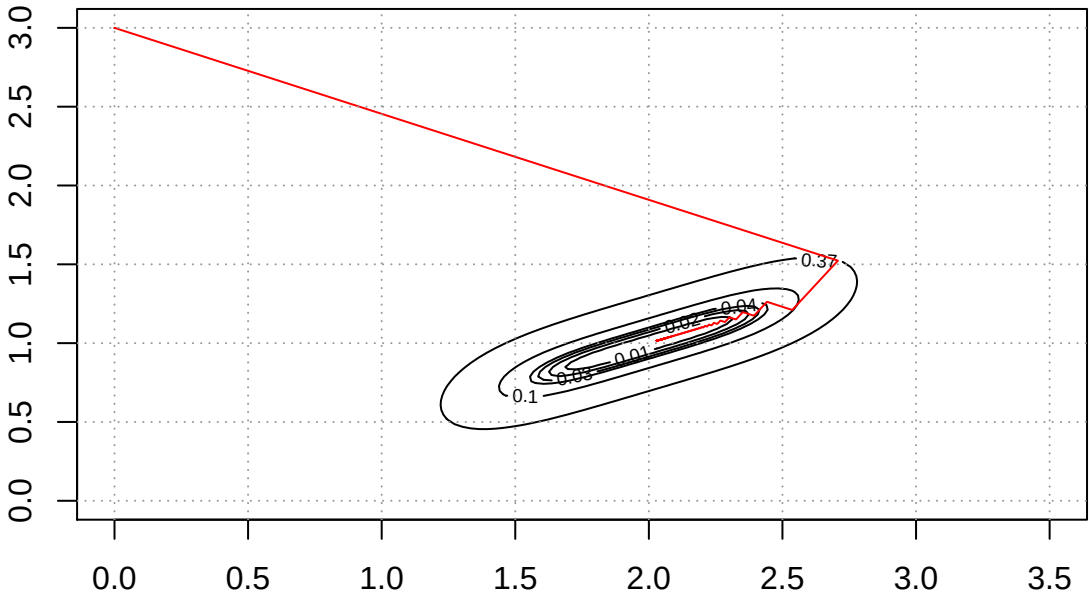
También podemos ver como evoluciona el algoritmo gráficamente en 2d sobre las curvas de nivelen, con ayuda de la función `contour()`:

```
ptos.vx = df.res[,2:3]
x <- seq(0, 3, length.out=100)
y <- seq(0, 3, length.out=100)
z <- funx(expand.grid(x, y))
contour(x, y, matrix(z, length(x)),xlim=c(0,3.5))
lines(ptos.vx,type="l",col="red")
grid(col=gray(0.6))
```



En el siguiente código, se eligen los niveles para los que se representan las curvas de nivel:

```
ptos.vx = df.res[,2:3]
niveles = unique(round(funx(ptos.vx),2))
x <- seq(0, 3, length.out=100)
y <- seq(0, 3, length.out=100)
z <- funx(expand.grid(x, y))
#contour(x, y, matrix(z, length(x)),nlevels = 20)
contour(x, y, matrix(z, length(x)),levels = niveles,xlim=c(0,3.5))
lines(ptos.vx,type="l",col="red")
grid(col=gray(0.6))
```



1.6.1. Cálculo de derivadas simbólicas con R

En R existen dos funciones que permiten obtener derivadas: `D()` y `deriv()`. Veremos aquí el uso de `deriv()`.

Para derivar una función, debemos definirla como una expresión:

$$f(x) = \sin(x)\ e^{-ax}$$

```
f = expression(sin(x)*exp(-a*x))
```

Siempre se puede convertir una expresión en una función de la siguiente manera:

```
ffun = function(x,a) eval(f)
```

Podemos obtener la derivada respecto a `x`:

```
(D1 = deriv(f,"x"))
```

```
## expression({
##     .expr1 <- sin(x)
##     .expr4 <- exp(-a * x)
##     .value <- .expr1 * .expr4
##     .grad <- array(0, c(length(.value), 1L), list(NULL, c("x")))
##     .grad[, "x"] <- cos(x) * .expr4 - .expr1 * (.expr4 * a)
##     attr(.value, "gradient") <- .grad
##     .value
## })
```

```
x = 0:3
a=1
#options(digits=3)
eval(D1)
```

```
## [1] 0.000000000 0.309559876 0.123060025 0.007025951
## attr(,"gradient")
##                x
## [1,]  1.00000000
## [2,] -0.11079377
## [3,] -0.17937937
## [4,] -0.05631478
```

Para usar el resultado de `deriv()` en una función, y obtener la segunda derivada, haremos lo siguiente:

```
(D1fun = deriv(f,"x", hessian = T, func=T))
```

```
## function (x)
## {
##     .expr1 <- sin(x)
##     .expr4 <- exp(-a * x)
##     .expr5 <- .expr1 * .expr4
##     .expr6 <- cos(x)
##     .expr8 <- .expr4 * a
##     .expr11 <- .expr6 * .expr8
##     .value <- .expr5
##     .grad <- array(0, c(length(.value), 1L), list(NULL, c("x")))
##     .hessian <- array(0, c(length(.value), 1L, 1L), list(NULL,
##         c("x"), c("x")))
##     .grad[, "x"] <- .expr6 * .expr4 - .expr1 * .expr8
##     .hessian[, "x", "x"] <- -(.expr11 + .expr5 + (.expr11 - .expr1 *
##         (.expr8 * a)))
##     attr(.value, "gradient") <- .grad
##     attr(.value, "hessian") <- .hessian
##     .value
## }
```

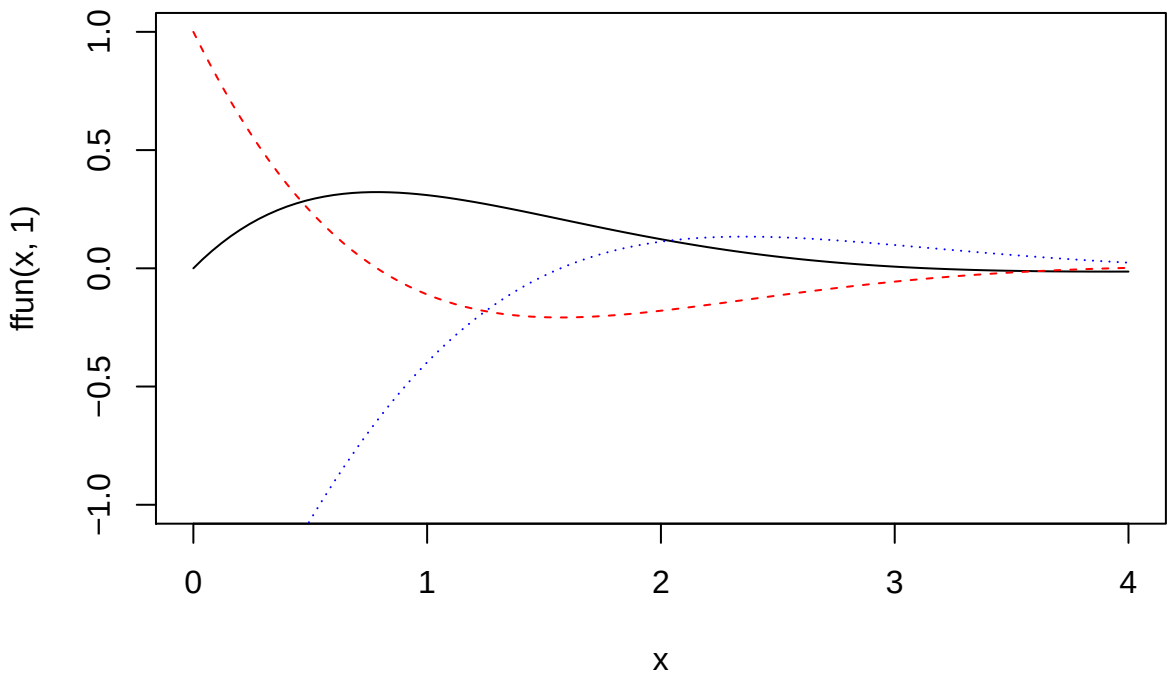
```
D1grad = function(x) attr(D1fun(x),"gradient")
D1hess = function(x) attr(D1fun(x),"hessian")
```

El parámetro `a` debe definirse como una variable externa (global)

```
a=1
```

Con el siguiente código podemos representar gráficamente las funciones:

```
curve(ffun(x,1), 0, 4, ylim = c(-1,1))
curve(D1grad(x),  lty=2, add=T,col="red")
curve(D1hess(x),  lty=3, add=T,col="blue")
```



1.6.2. Cálculo de vectores gradientes y matrices hessianas con cálculo simbólico con R

A continuación se muestra la forma generalizada de usar la función `deriv()` para obtener vectores gradiente y matrices hessiana de una función matemática.

```
vx.ini = c(0,3)
fex = expression( (x-2)^4+(x-2*y)^2 )
fx = function(x,y) eval(fex)
fx(0,3)
```

```
## [1] 52
```

```
x=0;y=3;eval(fex)
```

```
## [1] 52
```

```
## Gradientes y Hessianos
D1fun = deriv(fex,c("x","y"), hessian = T, func=T)
#D1fun(0,3)
D1grad = function(x,y) attr(D1fun(x,y),"gradient")
(Grad = D1grad(0,3))
```

```
##        x  y
## [1,] -44 24
```

```
D1hess = function(x,y)
  matrix(as.vector(attr(D1fun(x,y),"hessian")),nrow=2,ncol=2)
(Hess = D1hess(0,3))
```

```
##      [,1] [,2]
## [1,]   50   -4
## [2,]   -4    8
```

```
(invHess = solve(D1hess(0,3)))
```

```
##            [,1]       [,2]
## [1,] 0.02083333 0.01041667
## [2,] 0.01041667 0.13020833
```

```
(SegTerNewton = as.vector(- invHess %*% t(Grad)))
```

```
## [1]  0.6666667 -2.6666667
```

```
(SigIteNewton = vx.ini + SegTerNewton)
```

```
## [1] 0.6666667 0.3333333
```

Vectorizar las funciones obtenidas:

```
# Vectorizar la función
fx.v = function(vx) fx(vx[1],vx[2])
fx.v(vx.ini)
```

```
## [1] 52
```

```
# Vectorizar la función gradiente
D1grad.v = function(vx) {
  x = vx[1]
  y = vx[2]
  as.vector(attr(D1fun(x,y),"gradient"))
}
D1grad.v(vx.ini)
```

```
## [1] -44  24
```

```
# Vectorizar la matriz hessiana
D1hess.v = function(vx) {
  x = vx[1]
  y = vx[2]
  matrix(as.vector(attr(D1fun(x,y),"hessian")),nrow=2,ncol=2)
}
D1hess.v(vx.ini)
```

```
##      [,1] [,2]
## [1,]   50   -4
## [2,]   -4    8
```

1.7. Multidimensional no restringida: usando algoritmos implementados ya en R

Consideramos la función:

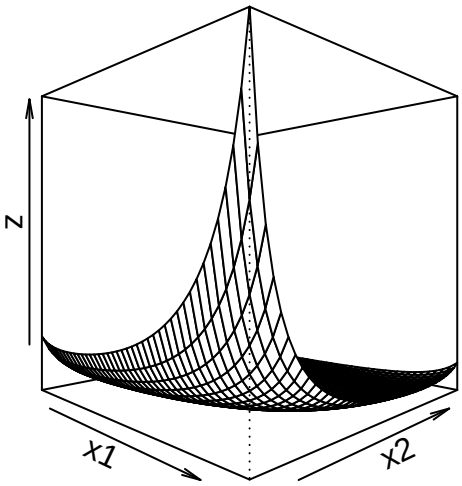
$$f(x_1, x_2) = \frac{1}{x_1} + \frac{1}{x_2} + \frac{1 - x_2}{x_2(1 - x_1)} + \frac{1}{(1 - x_1)(1 - x_2)}$$

Vemos en primer lugar la función en 3d con ayuda de `persp()`:

```
x1 = x2 = seq(.1,.9,.02)

z = outer(x1,x2,FUN=function(x1,x2) (1/x1 + 1/x2 + 1-x2)/(x2*(1-x1)) + 1/((1-x1)*(1-x2)))

persp(x1,x2,z,theta=45,phi=0)
```



Para minimizar una función `f` con respecto a (x_1, x_2) utilizamos la función `optim()`:

```
f = function(x) {
  x1 = x[1]
  x2 = x[2]
  return(1/x1 + 1/x2 + (1-x2)/(x2*(1-x1)) + 1/((1-x1)*(1-x2)))
}
(soluc = optim(c(.5,.5),f) )
```

```
## $par
## [1] 0.3636913 0.5612666
##
## $value
## [1] 9.341785
##
## $counts
## function gradient
##       55       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

```
(vx.sol = soluc$par)
```

```
## [1] 0.3636913 0.5612666
```

```
(fvx.sol = soluc$value)
```

```
## [1] 9.341785
```

1.8. Multidimensional restringida: método de penalizaciones

Codificamos en R el método:

```
optmul_res_penalizaciones = function(vx.ini,funx,lrest.igualdad,
                                     phi.igualdad=function(x) abs(x)^2,mu1 = 0.1,beta=10,
                                     epsilon=0.01,nummax.iter = 1000,report.calculos=T) {
  #epsilon > 0
  #browser()
  mres = NULL
  k = 1
  vxki = vx.ini
  ndimension = length(vx.ini)
  f.alfa = function(vx,muk) {
    pri = funx(vx)
    pri = pri + muk * phi.igualdad(lrest.igualdad[[1]](vx))
    return(pri)
  }
  #mu1 = 0.1
  #beta = 10
  muk = mu1
  # Paso 3
  repeat {

    fmu.temp = function(vxki) f.alfa(vxki,muk)
    solk =  optim(vxki,fmu.temp)
    vxki1 = solk$par
    fpvxki1 = solk$value
    fvxki1 = funx(vxki1)
    alfavki1 = lrest.igualdad[[1]](vxki1)^2
    muk.alfavki1 = muk * alfavki1
    vres = c(k,muk,vxki1,fvxki1,fpvxki1,alfavki1,muk.alfavki1)
    if (report.calculos) {
      if (is.null(mres)) {
        mres = vres
      } else {
        mres = rbind(mres,vres)
      }
    }

    if ((muk.alfavki1<epsilon) | (k>=nummax.iter)) {
      break
    } else {
      k = k+1
      vxki = vxki1
      muk = muk*beta
    }
  } # final repeat

  if (report.calculos) {
    colnames(mres) = c("k","muk",paste0("vxkmas1",1:ndimension),"fvxkmas1","fpvxkmas1",
                       "h2vxkmas1","mukporh2vxkmas1")
    rownames(mres) = NULL
    dfres = as.data.frame(mres)
  } else {
    names(vres) = c("k","muk",paste0("vxkmas1",1:ndimension),"fvxkmas1","fpvxkmas1",
```

```
                    "h2vxkmas1","mukporh2vxkmas1")
    dfres = as.data.frame(vres)
  }

  print("Óptimo en vxkmas1")
  return(dfres)

}
```

Lo utilizamos sobre la función $f(x_1, x_2) = (x_1 - 2)^4 + (x_1 - 2x_2)^2$, sujeto a $x_1^2 - x_2 = 0$.

```
vx.ini = c(2,1)
funx = function(vx) (vx[1]-2)^4+(vx[1]-2*vx[2])^2
lrest.igualdad = list(function(vx) (vx[1]^2-vx[2]) )
df.res = optmul_res_penalizaciones(vx.ini,funx,lrest.igualdad,
        phi.igualdad=function(x) abs(x)^2,mu1 = 0.1,beta=10,
        epsilon=0.0001,nummax.iter = 1000,report.calculos=T)
```

```
## [1] "Óptimo en vxkmas1"
```

```
knitr::kable(df.res[c(1:5,nrow(df.res)),],digits = 5)
```

	k	muk	vxkmas11	vxkmas12	fvxkmas1	fpvxkmas1	h2vxkmas1	mukporh2vxkmas1
1	1	1e-01	1.45393	0.76077	0.09349	0.27659	1.83098	0.18310
2	2	1e+00	1.16870	0.74067	0.57529	0.96617	0.39088	0.39088
3	3	1e+01	0.99059	0.84239	1.52005	1.71295	0.01929	0.19290
4	4	1e+02	0.95077	0.88749	1.89128	1.91840	0.00027	0.02712
5	5	1e+03	0.94610	0.89342	1.94053	1.94335	0.00000	0.00282
7	7	1e+05	0.94558	0.89411	1.94613	1.94616	0.00000	0.00003

Ahora sin iteraciones

```
df.res2 = optmul_res_penalizaciones(vx.ini,funx,lrest.igualdad,
        phi.igualdad=function(x) abs(x)^2,mu1 = 0.1,beta=10,
        epsilon=0.0001,nummax.iter = 1000,report.calculos=F)
```

[1] “Óptimo en vxkmas1”

```
knitr::kable(df.res2,digits = 5,format = 'latex')
```

	vres
k	7.00000e+00
muk	1.00000e+05
vxkmas11	9.45580e-01
vxkmas12	8.94110e-01
fvxkmas1	1.94613e+00
fpvxkmas1	1.94616e+00
h2vxkmas1	0.00000e+00
mukporh2vxkmas1	3.00000e-05

2. Ejercicios propuestos

2.1. Ejercicio 1

Implemente en R el algoritmo de Newton de búsqueda unidimensional para funciones dos veces diferenciables. Utilícelo con varios funciones ejemplo y muestre gráficamente como evoluciona.

2.2. Ejercicio 2

Implemente en R el algoritmo de Hooke y Jeeves de búsqueda multidimensional no restringida sin usar diferenciabilidad. Utilícelo para resolver el problema:

$$Min \quad (x_1 - 2)^4 + (x_1 - 2x_2)^2$$

partiendo del punto inicial: $(x_1 = 0, x_2 = 3)$. Muestre gráficamente como evoluciona.

2.3. Ejercicio 3

Implemente en R el algoritmo de Newton de búsqueda multidimensional no restringida para funciones con diferenciabilidad de segundo orden. Utilícelo para resolver el problema:

$$Min \quad (x_1 - 2)^4 + (x_1 - 2x_2)^2$$

partiendo del punto inicial: $(x_1 = 0, x_2 = 3)$. Muestre gráficamente como evoluciona.

2.4. Ejercicio 4

Implemente en R el algoritmo de Barreras (visto en clase de teoría) de búsqueda multidimensional restringida. Utilícelo para resolver el problema:

$$Min \quad (x_1 - 2)^4 + (x_1 - 2x_2)^2 \qquad s.a.\ x_1^2 - x_2 \leq 0$$

partiendo del punto inicial: $(x_1 = 0, x_2 = 3)$. Muestre gráficamente como evoluciona.

2.5. Ejercicio 5

Encontrar la solución de los siguientes problemas, aplicando todos los algoritmos conocidos. Tomar $l = 0.1$

1. $Min\ (x-2)^4 + (x-6)^2$ s.a.: $x \in [-5, 5]$.
2. $Min\ x^4 - 3x^3 + 2x^2 + 3x$ s.a.: $x \in [-5, 5]$.

2.6. Ejercicio 6

Resolver el problema máx$(\frac{1}{1+x^2} - y^2)$ a partir de la semilla $(1, 1)$.

2.7. Ejercicio 7

Resolver el siguiente problema usando el método de Newton

$$Min\ f(\mathbf{x}) = (3x_1 - 1)^3 + 4x_1x_2 + x_2^2$$

comenzando desde el punto inicial $\mathbf{x}_0 = (1, 2)$.

2.8. Ejercicio 8

Resolver el problema

$$\begin{array}{lrl} Min & f(\mathbf{x}) & = x_1^2 + x_2^2 \\ s.a. & x_1 + x_2 - 1 & = 0 \end{array}$$

Comenzando en el punto $(0, 0)$.

2.9. Ejercicio 9

Sea el problema de programación no lineal:

$$Max\ 4x_1 + 6x_2 - 2x_1^2 - 2x_1x_2 - 2x_2^2$$

Resolver el problema partiendo del punto inicial $\mathbf{x}_0 = (1, 1)$.