GIIT_AMA_Numerico_Ejercicios

Práctica de Ampliación de Matemáticas (Tema 3: Numérico)

Alumnos que integran el grupo

1. Alfonso Nguema Ela Nanguan
2. Carlos Rebolledo Aliseda
3. David Pozo Nuñez

Ejercicios

PARTE A)

1. Aplica 4 pasos del método de la bisección a la función $f(x) = e^{-x} - sen(x)$ para encontrar una raíz entre -0.2 y 1.1. Utiliza un condicional (if) para que el intervalo en el que está la raíz se escoja automáticamente.

2. Crea un bucle que realice n pasos del algoritmo de la bisección (el valor de n se introducirá anteriormente).

3. Crea una función que reciba una función f, un intervalo $[a, b]$ y un número de pasos n y aplique n pasos de la bisección para calcular una aproximación de la raíz de f en el intervalo $[a, b]$.

4. Crea una modificación de la función anterior para que reciba la función, el intervalo y un valor máximo de error y devuelva una aproximación de la raíz con un error menor que el valor máximo introducido. Para calcular el número de pasos se puede utilizar que $E \leq (b-a)/2^n$ y aplicando logaritmos, $n \geq log((b-a)/E)/log(2)$.

5. Crea una función que reciba una función f, un valor inicial $x0$ y un número de pasos n y delvuelva el resultado de aplicar n pasos del método de Newton-Raphson.

6. Crea una función que reciba una función f, un valor inicial $x0$ y un error E y delvuelva el resultado de aplicar el método de Newton-Raphson para un error máximo E.

7. Modifica el método de Newton-Raphson para obtener el método de la secante con n pasos.

Ejercicio 1:

```
f(x)=e^(-x)-sin(x);
a,b=-0.2,1.1
```

```
#1 Iteracion
c=(a+b)/2
if f(c)<0:
    b=c
else:
    a=c
(a,b)
```

```
(0.450000000000000, 1.10000000000000)
```

```
#2 Iteracion
c=(a+b)/2
if f(c)<0:
    b=c
else:
    a=c
(a,b)
```

```
(0.450000000000000, 0.775000000000000)
```

```
#3 Iteracion
c=(a+b)/2
if f(c)<0:
    b=c
else:
    a=c
(a,b)
```

```
(0.450000000000000, 0.612500000000000)
```

```
#4 Iteracion
c=(a+b)/2
if f(c)<0:
    b=c
else:
    a=c
(a,b)
```

```
(0.531250000000000, 0.612500000000000)
```

Ejercicio 2:

```
# Creamos la funcion de biseccion que recibe un entero
# Si el resultado del producto es negativo,
# significa que tienen signos distintos
def biseccion(n):
    f(x)=e^(-x)-sin(x)
    a,b=-0.2,1.1
    i = 0
    while i < n:
        c = (a+b)/2
        if (f(a)*f(c))<0:
```

```
            b = c
        else:
            a = c
        # Mostramos el intervalo
        print "Iteracion ",(i+1),": [",a,", ",b,"]"
        i+=1 # Incrementamos el indice
```

```
# Hacemos una prueba con 4 iteraciones
biseccion(4)
```

```
    Iteracion  1 : [ 0.450000000000000 ,  1.10000000000000 ]
    Iteracion  2 : [ 0.775000000000000 ,  1.10000000000000 ]
    Iteracion  3 : [ 0.937500000000000 ,  1.10000000000000 ]
    Iteracion  4 : [ 1.01875000000000 ,  1.10000000000000 ]
```

```
# Hacemos una prueba con 6 iteraciones
biseccionn(6)
```

```
    Iteracion  1 : [ 0.450000000000000 ,  1.10000000000000 ]
    Iteracion  2 : [ 0.775000000000000 ,  1.10000000000000 ]
    Iteracion  3 : [ 0.937500000000000 ,  1.10000000000000 ]
    Iteracion  4 : [ 1.01875000000000 ,  1.10000000000000 ]
    Iteracion  5 : [ 1.05937500000000 ,  1.10000000000000 ]
    Iteracion  6 : [ 1.07968750000000 ,  1.10000000000000 ]
```

Ejercicio 3:

```
# Recibe una funcion, dos valores de un intervalo y un entero
def biseccion2(funcion,a,b,n):
    print "Funcion: ", funcion(x)
    print "Intervalo: [",a,", ",b,"]"
    print "Numero de pasos: ",n
    i = 0
    while i < n:
        c = (a+b)/2
        if (funcion(a)*funcion(c))<0:
            b = c
        else:
            a = c
        print "Iteracion ",(i+1),": [",a,", ",b,"]"
        i+=1 # Incrementamos el indice
```

```
# Prueba 1
funcion(x)=e^(-x)-sin(x)
a, b = -0.2,1.1
n = 4
biseccion2(funcion,a,b,n)
```

```
    Funcion:  0.0100000000000000^(-x) - sin(x)
    Intervalo: [ -0.200000000000000 ,  1.10000000000000 ]
    Numero de pasos:  4
    Iteracion  1 : [ 0.450000000000000 ,  1.10000000000000 ]
    Iteracion  2 : [ 0.775000000000000 ,  1.10000000000000 ]
    Iteracion  3 : [ 0.937500000000000 ,  1.10000000000000 ]
```

```
    Iteracion  3 : [ 0.937500000000000 ,  1.10000000000000 ]
    Iteracion  4 : [ 1.01875000000000 ,  1.10000000000000 ]
```

```
# Prueba 2
funcion(x)=-cos(x)+sin(x)
a, b= 0, 2
n = 3
biseccion2(funcion,a.n(),b.n(),n)
```

```
    Funcion:  -cos(x) + sin(x)
    Intervalo: [ 0.000000000000000 ,  2.00000000000000 ]
    Numero de pasos:  3
    Iteracion  1 : [ 0.000000000000000 ,  1.00000000000000 ]
    Iteracion  2 : [ 0.500000000000000 ,  1.00000000000000 ]
    Iteracion  3 : [ 0.750000000000000 ,  1.00000000000000 ]
```

Ejercicio 4:

```
# Recibe un funcion, dos puntos de un intervalo y un error maximo
def biseccionError(funcion,a,b,e):
    print "Funcion: ", funcion(x)
    print "Intervalo: [",a,b,"]"
    print "Error maximo: ",e
    i = 0
    error = e
# Obtenemos los puntos con los que usaremos para calcular los errores
    ainicial, binicial = a,b
# Recorremos mientras el error obtenido sea mayor o igual que el
# error maximo pasado por parametro
    while error >= e:
        c = (a+b)/2
        if (funcion(a)*funcion(c))<0:
            b = c
        else:
            a = c
        i+=1
        error = (binicial-ainicial)/(2^i) # Calculamos el valor del error en
dicha iteracion
        print "Iteracion ",i,": [",a,", ",b,"], Error: ",error.n()
```

```
# Esta funcion ha sido cogida del test
# Prueba 1. Con esta prueba tienen que salir 9 pasos (iteraciones)
funcion(x) = x-sin(x)
a,b=-1,2
e = 0.01
biseccionError(funcion,a,b,e)
```

```
    Funcion:  x - sin(x)
    Intervalo: [ -1 2 ]
    Error maximo:  0.0100000000000000
    Iteracion  1 : [ -1 ,  1/2 ], Error:  1.50000000000000
    Iteracion  2 : [ -1/4 ,  1/2 ], Error:  0.750000000000000
    Iteracion  3 : [ -1/4 ,  1/8 ], Error:  0.375000000000000
```

```
Iteracion  4 : [ -1/16 ,  1/8 ], Error:  0.187500000000000
Iteracion  5 : [ -1/16 ,  1/32 ], Error:  0.0937500000000000
Iteracion  6 : [ -1/64 ,  1/32 ], Error:  0.0468750000000000
Iteracion  7 : [ -1/64 ,  1/128 ], Error:  0.0234375000000000
Iteracion  8 : [ -1/256 ,  1/128 ], Error:  0.0117187500000000
Iteracion  9 : [ -1/256 ,  1/512 ], Error:  0.00585937500000000
```

```
# Esta funcion, el error y el intervalo ha sido cogida del test
# Prueba 2. Con esta prueba tienen que salir 8 pasos (iteraciones)
funcion(x) = cos(3*x)-(1/x)
a,b=2.500,4
e = 0.01
biseccionError(funcion,a,b,e)
```

```
Funcion:  -1/x + cos(3*x)
Intervalo: [ 2.50000000000000 4 ]
Error maximo:  0.0100000000000000
Iteracion  1 : [ 3.25000000000000 ,  4 ], Error:  0.750000000000000
Iteracion  2 : [ 3.62500000000000 ,  4 ], Error:  0.375000000000000
Iteracion  3 : [ 3.62500000000000 ,  3.81250000000000 ], Error:
0.187500000000000
Iteracion  4 : [ 3.71875000000000 ,  3.81250000000000 ], Error:
0.0937500000000000
Iteracion  5 : [ 3.71875000000000 ,  3.76562500000000 ], Error:
0.0468750000000000
Iteracion  6 : [ 3.74218750000000 ,  3.76562500000000 ], Error:
0.0234375000000000
Iteracion  7 : [ 3.75390625000000 ,  3.76562500000000 ], Error:
0.0117187500000000
Iteracion  8 : [ 3.75390625000000 ,  3.75976562500000 ], Error:
0.00585937500000000
```

Ejercicio 5:

```
def funcion(f,x0,n):
    print "Punto inicial: ",x0
    for i in [1..n]:
        xcopia=x0
        x0=(x0-(f(x0)/f.diff(x)(x0))).n() ;
        print "Iteración ",i, ": ",x0
        error=abs(x0-xcopia);
        print "Error:",error;
        print "Derivada: ",f.diff(x)(x0)
```

```
f(x)=e^x-4
c=1.6
funcion(f,c,4)
```

```
Punto inicial:  1.60000000000000
Iteración  1 :  -1374.80358854283
Error: 1376.40358854283
Derivada:  -1.86390069393707e2750
Iteración  2 :  -1374.58644130188
```

```
Error: 0.217147240951590
Derivada:  -6.85690745684745e2749
Iteración  3 :  -1374.36929406093
Error: 0.217147240951590
Derivada:  -2.52251528338975e2749
Iteración  4 :  -1374.15214681997
Error: 0.217147240951590
Derivada:  -9.27981512799998e2748
```

Ejercicio 6:

```
def fError(f,x0,e):
    error=100
    i=1
    print "Punto inicial: ",x0
    while error>e:
        xcopia=x0
        x0=(x0-(f(x0)/f.diff(x)(x0)))
        print "Iteración ",i, ": ",x0
        error=abs(x0-xcopia)
        print "Error:",error
        print "Derivada: ",f.diff(x)(x0)
        i=i+1
```

```
# Prueba
f(x)=e^x-4
c=1.6
fError(f,c,10^(-5))
```

```
Punto inicial:  1.60000000000000
Iteración  1 :  1.40758607197862
Error: 0.192413928021378
Derivada:  4.08607998661581
Iteración  2 :  1.38651942940242
Error: 0.0210666425762027
Derivada:  4.00090037444918
Iteración  3 :  1.38629438644586
Error: 0.000225042956562449
Derivada:  4.00000010130386
Iteración  4 :  1.38629436111989
Error: 2.53259655469407e-8
Derivada:  4.00000000000000
```

Ejercicio 7:

```
def fSecante(f,x0,x1,n):
    print "x0: ",x0,"x1:",x1
    for i in [1..n]:
```

```
        x0copia=x0
        x1copia=x1
        x1=((f(x1)*x0-(f(x0)*x1))/(f(x1)-f(x0))) ;
        x0=x1copia ;
        print "Iteración ",i, ": ",x1
```

```
# Prueba
f(x)=e^x-4
x0=0.8
x1=1.6
fSecante(f,x0,x1,3)
```

```
x0:  0.800000000000000 x1: 1.60000000000000
Iteración  1 :  1.32046624502352
Iteración  2 :  1.37944168549308
Iteración  3 :  1.38652265104928
```

PARTE B)

En los ejercicios siguientes, DNI es la suma de los números de DNI de los miembros del grupo, a la última cifra de dicho número, y $dni = DNI/2.9^{ln(DNI)}$.

1. Resolver mediante el método de Gauss (con pivote) el sistema. Comprueba el error cometido (sustituye la solución en el sistema y calcula los errores cometidos en cada ecuación).

$$\left.\begin{aligned} 2x-(3+dni)y+100z=1\\ x-10y+0.00002z=0\\ 3x-100y+0.00003z=0 \end{aligned}\right\}$$

2. Aplicar cinco pasos del método de Gauss-Seidel al sistema:

$$\left.\begin{aligned} (12+dni)x-y+z=1\\ 5x+4y-2z=2\\ -2x+y-6z=3 \end{aligned}\right\}$$

3. Aplicar cuatro pasos del método de Newton para, partiendo de $(0,0,0)$, aproximar un cero del sistema:

$$\left.\begin{aligned} (12+dni)x - y^3 - exp(z) = 1 \\ -sin(x) + (10+a)y - 2z = 2 \\ -2x - \cos(y) + 16\sin(z) = 3 \end{aligned}\right\}$$

Si quieres representar las ecuaciones y los puntos, deberás usar las versiones 3d de implicit_plot y points.

```
DNI1 = 4093582.0;
DNI2 = 09090955.0;
DNI3 = 09222960.0;
DNI = DNI1+DNI2+DNI3;
dni = DNI/2.9^(ln(DNI))
a=7
```

Ejercicio 1:

```
#Definimos la matriz con los datos dados en el enunciado
matriz = matrix(RDF,[[2,-(3+dni),+100,1],[1,-10,0.00002,0],
[3,-100,0.00003,0]])
show(matriz)
```

$$\begin{pmatrix} 2.0 & -3.33446496263 & 100.0 & 1.0 \\ 1.0 & -10.0 & 2\times 10^{-05} & 0.0 \\ 3.0 & -100.0 & 3\times 10^{-05} & 0.0 \end{pmatrix}$$

```
# Cambiamos las filas, ya que hay que resolverlo
# por el sistma de pivote, tomando como elemento pivote al 3.0
matriz.swap_rows(0, 2)
show(matriz)
```

$$\begin{pmatrix} 3.0 & -100.0 & 3\times 10^{-05} & 0.0 \\ 1.0 & -10.0 & 2\times 10^{-05} & 0.0 \\ 2.0 & -3.33446496263 & 100.0 & 1.0 \end{pmatrix}$$

```
# Hacemos cero al elemento (1, 0) modificando la fila entera
matriz[1,:]=matriz[1,:]-(1/3)*matriz[0,:]
show(matriz)
```

$$\begin{pmatrix} 3.0 & -100.0 & 3\times 10^{-05} & 0.0 \\ 0.0 & 23.3333333333 & 1\times 10^{-05} & 0.0 \\ 2.0 & -3.33446496263 & 100.0 & 1.0 \end{pmatrix}$$

```
# Hacemos cero al elemento (2, 0) modificando la fila entera
matriz[2,:]=3*matriz[2,:]-2*matriz[0,:]
show(matriz)
```

$$\begin{pmatrix} 3.0 & -100.0 & 3\times 10^{-05} & 0.0 \\ 0.0 & 23.3333333333 & 1\times 10^{-05} & 0.0 \\ 0.0 & 189.996605112 & 299.99994 & 3.0 \end{pmatrix}$$

```
# Intercambiamos las fila (1, 2) para tomar como pivote
# el elemento (2,1) = 189.996605112
matriz.swap_rows(1, 2)
show(matriz)
```

$$\begin{pmatrix} 3.0 & -100.0 & 3\times10^{-05} & 0.0 \\ 0.0 & 189.996605112 & 299.99994 & 3.0 \\ 0.0 & 23.3333333333 & 1\times10^{-05} & 0.0 \end{pmatrix}$$

```
# Hacemos cero al elemento (2, 1) modificando la fila entera
matriz[2,:]=(1/23.3333333333).n()*matriz[2,:]-
(1/189.996605112).n()*matriz[1,:]
show(matriz)
```

$$\begin{pmatrix} 3.0 & -100.0 & 3\times10^{-05} & 0.0 \\ 0.0 & 189.996605112 & 299.99994 & 3.0 \\ 0.0 & 9.26814180957\times10^{-13} & -1.57897483692 & -0.0157897558129 \end{pmatrix}$$

```
# Aproximamos al valor del elemento (2, 1) a 0, ya que
# 9.26814180957×10^(-13) está cerca de 0, para ello modifica
# ese elemento
matriz[2,1] = 0.0
show(matriz)
```

$$\begin{pmatrix} 3.0 & -100.0 & 3\times10^{-05} & 0.0 \\ 0.0 & 189.996605112 & 299.99994 & 3.0 \\ 0.0 & 0.0 & -1.57897483692 & -0.0157897558129 \end{pmatrix}$$

```
# Obtenemos la solucion, despejando (de abajo a arriba).
# Guardamos las soluciones en un vector
v = vector(RDF,3) # vector vacío
v
```

```
(0.0, 0.0, 0.0)
```

```
# Obtenemos el valor de la z
v[2]=matriz[2,3]/matriz[2,2];
v
```

```
(0.0, 0.0, 0.01000000471423944)
```

```
# Obtenemos el valor de la y, sabiendo que ya tenemos el de la z
v[1]=(matriz[1,3]-(matriz[1,2]*v[2]))/matriz[1,1]
v
```

```
(0.0, -4.2857163072687405e-09, 0.01000000471423944)
```

```
# Obtenemos el valor de la x, habiendo calculado y,z.
v[0]=(matriz[0,3]-(matriz[0,2]*v[2])-(matriz[0,1]*v[1]))/matriz[0,0]
v
```

```
(-2.428572573846858e-07, -4.2857163072687405e-09,
0.01000000471423944)
```

```
# Sustituimos la solucion en el sistema
matriz = matrix(RDF,
[
      [2*v[0],          -(3+dni)*v[1],     +100*v[2],            1],
      [1*v[0],          -10*v[1],          0.00002*v[2],         0],
```

```
      [3*v[0],         -100*v[1],         0.00003*v[2],         0]
])
show(matriz)
```

$$\begin{pmatrix} -4.85714514769 \times 10^{-07} & 1.42905708664 \times 10^{-08} & 1.00000047142 & 1.0 \\ -2.42857257385 \times 10^{-07} & 4.28571630727 \times 10^{-08} & 2.00000094285 \times 10^{-07} & 0.0 \\ -7.28571772154 \times 10^{-07} & 4.28571630727 \times 10^{-07} & 3.00000141427 \times 10^{-07} & 0.0 \end{pmatrix}$$

```
# La primera ecuación. Calculando el error cometido
matriz[0,0]+matriz[0,1]+matriz[0,2]
```

    1.0000000000000002

```
# La segunda ecuación. Calculando el error cometido
matriz[1,0]+matriz[1,1]+matriz[1,2]
```

    -2.720955700200511e-17

```
# La tercera ecuación. Calculando el error cometido
matriz[2,0]+matriz[2,1]+matriz[2,2]
```

    -5.293955920339377e-23

Ejercicio 2:

```
A=matrix(RDF,[[(12+dni),-1,1],[5,4,-2],[-2,1,-6]])
b=vector(RDF,[1,2,3])
D=matrix(RDF,[[(12+dni),0,0],[0,4,0],[0,0,-6]])
L=matrix(RDF,[[0,0,0],[5,0,0],[-2,1,0]])
U=matrix(RDF,[[0,-1,1],[0,0,-2],[0,0,0]])
```

```
x0=vector(RDF,[0,0,0])
M0=matrix(RDF,[[0],[0],[0]])       # Aquí vamos guardando los pasos para
mostrarlos al final.
for i in [1..5]:
    x0=(D+L).solve_right(-U*x0+b)
    M0=M0.augment(x0)              # Guardamos las soluciones
html.table( [ [("Paso {pi}".format(pi=k)) for k in [0..5]],[(M0[:,k]) for k
in [0..5]]], header = True )
```

```
/home/sage1516/sage-7.0/local/lib/python2.7/site-packages/sage/misc/\
html.py:82: DeprecationWarning: use table() instead of html.table()
See http://trac.sagemath.org/18292 for details.
  output = method(*args, **kwds)
```

Paso 0	Paso 1	Paso 2	Paso 3	Paso 4
$\begin{pmatrix} 0.0 \\ 0.0 \\ 0.0 \end{pmatrix}$	$\begin{pmatrix} 0.0810736422722 \\ 0.39865794716 \\ -0.460581556231 \end{pmatrix}$	$\begin{pmatrix} 0.150735318396 \\ 0.0812900738893 \\ -0.536696760484 \end{pmatrix}$	$\begin{pmatrix} 0.131176085811 \\ 0.0676815124941 \\ -0.532445109855 \end{pmatrix}$	$\begin{pmatrix} 0.1297 \\ 0.07161 \\ -0.5313 \end{pmatrix}$

Ejercicio 3:

```
#En primer lugar, definimos la función F(x,y) de modo que las soluciones al
#sistema sean un cero de F
F(x,y,z)=((12+dni)*x-y^3-exp(z)-1,-sin(x)+(10+a)*y-2*z-2,-2*x-
cos(y)+16*sin(z)-3);F
```

```
(x, y, z) |--> (-y^3 + 12.3344649626349*x - e^z - 1, 17*y - 2*z -
sin(x) - 2, -2*x - cos(y) + 16*sin(z) - 3)
```

```
#Partimos de (0,0,0)
x0,y0,z0=0.0,0.0,0.0
```

```
#Calculamos la matriz jacobiana en el punto x0
J0=matrix(RDF,F.diff()(x0,y0,z0))
J0
```

```
[12.334464962634913                0.0               -1.0]
[              -1.0               17.0               -2.0]
[              -2.0                0.0               16.0]
```

```
J0=matrix(RDF,F.diff()(x0,y0,z0))
xd1,yd1,zd1=J0.solve_right(-F(x0,y0,z0));
xd1,yd1,zd1
```

```
(0.1842832594946429, 0.16060906319813548, 0.2730354074368303)
```

```
x1,y1,z1=x0+xd1,y0+yd1,z0+zd1
x1,y1,z1
```

```
(0.184283259494643, 0.160609063198135, 0.273035407436830)
```

```
#2 Iteración
J1=matrix(RDF,F.diff()(x1,y1,z1))
xd2,yd2,zd2=J1.solve_right(-F(x1,y1,z1));
x2,y2,z2=x1+xd2,y1+yd2,z1+zd2
x2,y2,z2
```

```
(0.188278946516820, 0.161153870251859, 0.276222893377982)
```

```
#3 Iteración
J1=matrix(RDF,F.diff()(x2,y2,z2))
xd3,yd3,zd3=J1.solve_right(-F(x2,y2,z2));
x3,y3,z3=x2+xd3,y2+yd3,z2+zd3
x3,y3,z3
```

```
(0.188279662204706, 0.161154002749495, 0.276224404701058)
```

```
#4 Iteración
J1=matrix(RDF,F.diff()(x3,y3,z3))
xd4,yd4,zd4=J1.solve_right(-F(x3,y3,z3));
x4,y4,z4=x3+xd4,y3+yd4,z3+zd4
x4,y4,z4
```

```
(0.188279662204865, 0.161154002749542, 0.276224404701402)
```

C. Difuminando y perfilando una imagen

Vamos a ver un ejemplo de cómo se puede aplicar el método iterativo de Gauss-Seidel. Concretamente, veremos cómo funciona (de modo básico) un modelo de difuminado y perfilado. Estos modelos se aplican por ejemplo a imágenes de satélite para eliminar los efectos de difusión de la atmósfera, al análisis de imágenes por ordenador, para retocar fotografías, etc.

Cada imagen viene cargada en una matriz (por simplicidad asumiremos que está en blanco y negro). Vamos a necesitar algunas funciones que nos pasen los datos de la matriz a un vector y viceversa. Para ello, basta ejecutar la siguiente línea.

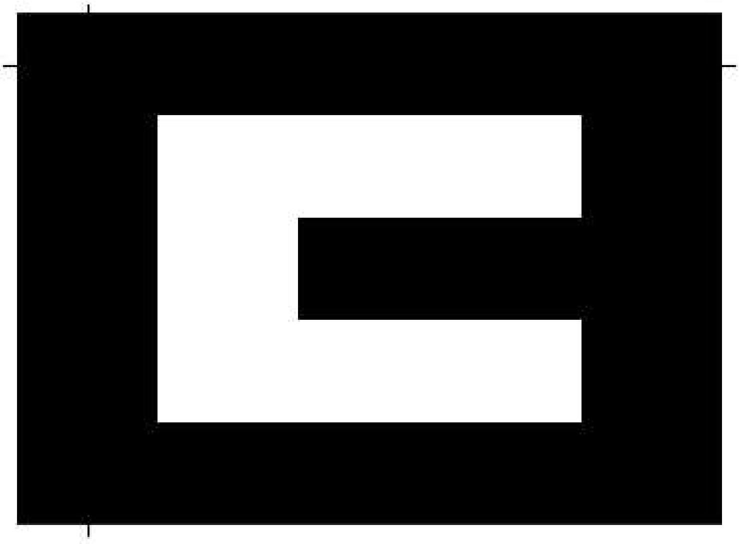
```
def matrix_to_vector(A):
    n,m=A.dimensions();
    v=vector(RDF,n*m)
    for i in range(0,n):
      for j in range(0,m):
          v[m*i+j]=A[i,j];
    return v;
def vector_to_matrix(v):
    n=sqrt(v.degree());
    A=matrix(RDF,n);
    for i in range(0,n):
      for j in range(0,n):
          A[i,j]=v[n*i+j];
    return A;
```

Para pasar la matriz a un vector, numeramos las celdas comenzando por la primera fila:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \rightarrow (1,2,3,4,5,6,7,8,9)$$

Definimos la matriz en la que tenemos la imagen. Cuando creamos la matriz estará a cero (color negro). Vamos a poner algunas posiciones a uno (blanco).

```
n=5;
A=matrix(RDF,n)
A[1,1:4]=Matrix([1,1,1])
A[3,1:4]=Matrix([1,1,1])
A[2,1]=1
show(plot(A),figsize=4)
```



Creamos la matriz de difuminado (está libremente basado en el difuminado gaussiano). Para ello cada posición pasa a tomar "un poco del color de los vecinos". Estamos considerando cuatro vecinos (arriba, abajo, derecha e izquierda). Si A(i,j) es el color del punto (i,j), después de la transformación será:

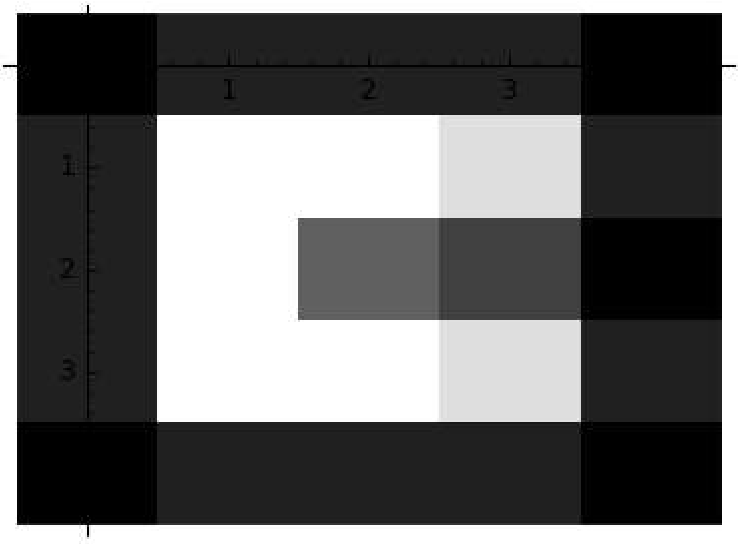
$$A(i,j) \to (1-4d)A(i,j) + d\,A(i-1,j) + d\,A(i+1,j) + d\,A(i,j-1) + d\,A(i,j+1)$$

Esto puede hacerse escribiendo la matriz A en forma de vector y multiplicando el vector por cierta matriz (dada por la expresión anterior). Guardaremos en N la matriz por la que hay que multiplicar (matriz de difuminado).

```
N=matrix(RDF,n*n);
d=1/10;
for i in range(0,n-1):
  for j in range(0,n):
    N[i+j*n,i+1+j*n]=d;   # vecino a la derecha (i+1,j)
    N[i+1+j*n,i+j*n]=d;   # vecino a la izquierda (i-1,j)
    N[i*n+j,(i+1)*n+j]=d; # vecino superior (i,j+1)
    N[(i+1)*n+j,i*n+j]=d; # vecino inferior (i,j-1)
for i in range(0,n*n):
    N[i,i]=1-4*d;         #  posición original (i,j)
```

Para difuminar la imagen, escribimos la matriz como vector y multiplicamos ese vector por la matriz de difuminado.

```
Av=matrix_to_vector(A);Av # Necesitamos considerar la imagen como un vector
b=N*Av # Multiplicamos la imagen Av por la matriz N
show(plot(vector_to_matrix(b)), figsize=4)  # Para dibujar necesitamos volver a pasar a la forma de matriz
```



Y podemos hacer el proceso inverso resolviendo el sistema $Nx = b$, para lo cual Gauss-Seidel es un método apropiado (a cada paso se perfila un poco más la imagen). Véamoslo en el ejemplo.

Construimos las matrices de Gauss-Seidel

```
U=matrix(RDF,n*n);
for i in range(0,n*n):
    for j in range(i+1,n*n):
        U[i,j]=N[i,j];
DL=N-U;
```

Y aplicamos el método

```
x0=vector(RDF,n*n)
m=3                                # Número de iteraciones del método
M0=Matrix(RDF,m+1,n*n)               # Aquí guardamos los pasos para pintarlo
después
M0[0,:]=x0;                        # guardamos la condición inicial
for i in range(1,m+1):
    x0=(DL).solve_right(-U*x0+b)
    M0[i,:]=x0;                   # guardamos el vector en la fila i
```

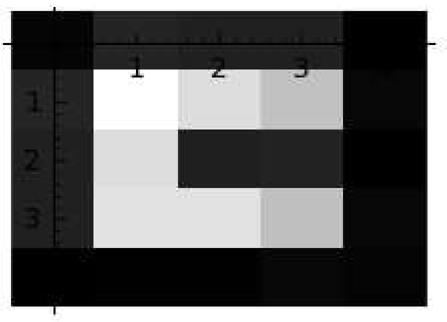
A continuación mostramos los pasos de Gauss-Seidel. Se puede observar que con 3 iteraciones prácticamente estamos en la matriz original. Esto no depende del número de puntos de la imagen, por lo que para matrices muy grandes este método es mucho más rápido que la eliminación gaussiana o que los métodos LU.

```
c0=plot(vector_to_matrix(vector(M0.row(0))), figsize=2.5)
c1=plot(vector_to_matrix(vector(M0.row(1))), figsize=2.5)
c2=plot(vector_to_matrix(vector(M0.row(2))), figsize=2.5)
c3=plot(vector_to_matrix(vector(M0.row(3))), figsize=2.5)
html.table([["Iteración 0", "Iteración 1","Iteración 2", "Iteración 3"],
[c0,c1,c2,c3]], header=True)
```

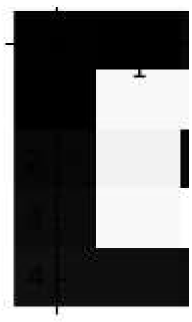
```
/home/sage1516/sage-7.0/local/lib/python2.7/site-packages/sage/misc/\
html.py:82: DeprecationWarning: use table() instead of html.table()
See http://trac.sagemath.org/18292 for details.
  output = method(*args, **kwds)
/home/sage1516/sage-7.0/local/lib/python2.7/site-packages/sage/misc/\
decorators.py:471: DeprecationWarning: the filename and linkmode
arguments are deprecated, use save() to save
See http://trac.sagemath.org/17234 for details.
  return func(*args, **kwds)
```

Iteración 0

Iteración 1



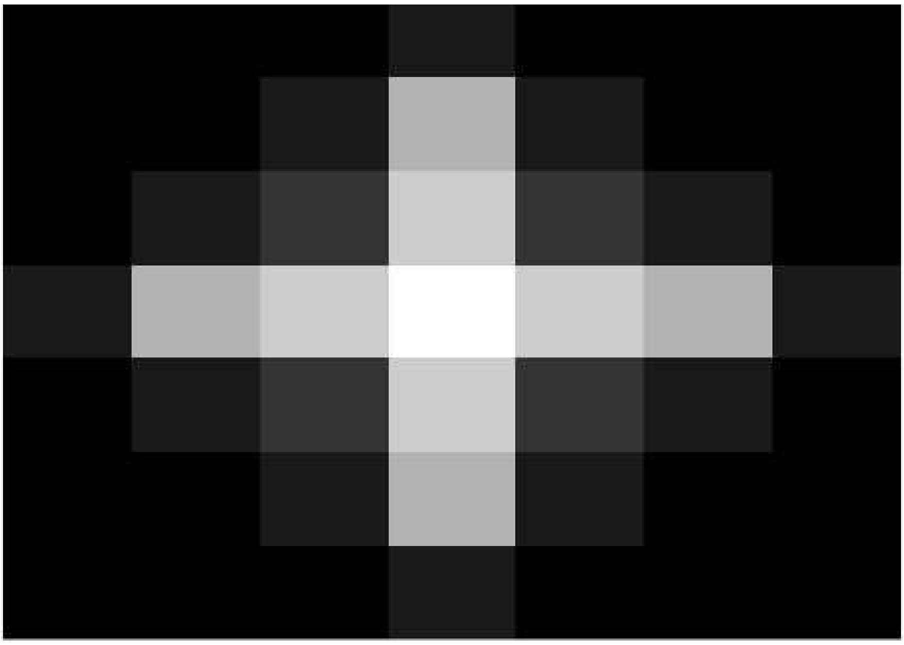
Iteración 2



Problema. Se tiene la siguiente matriz M, resultado de aplicar la matriz N sobre cierta imagen.

```
M=matrix(RDF,[[0.0, 0.0, 0.0, 0.1, 0.0, 0.0, 0.0],[0.0, 0.0, 0.1, 0.7, 0.1,
0.0, 0.0],[0.0, 0.1, 0.2, 0.8, 0.2, 0.1, 0.0],[0.1, 0.7, 0.8, 1.0, 0.8, 0.7,
0.1],[0.0, 0.1, 0.2, 0.8, 0.2, 0.1, 0.0],[0.0, 0.0, 0.1, 0.7, 0.1, 0.0,
0.0],[0.0, 0.0, 0.0, 0.1, 0.0, 0.0, 0.0]])
n=7
N=matrix(RDF,n*n);
d=1/10;
for i in range(0,n-1):
  for j in range(0,n):
    N[i+j*n,i+1+j*n]=d;   # vecino a la derecha (i+1,j)
    N[i+1+j*n,i+j*n]=d;   # vecino a la izquierda (i-1,j)
    N[i*n+j,(i+1)*n+j]=d; # vecino superior (i,j+1)
    N[(i+1)*n+j,i*n+j]=d; # vecino inferior (i,j-1)
for i in range(0,n*n):
    N[i,i]=1-4*d;         #  posición original (i,j)
```

```
show(plot(M), figsize=5,axes=False)
```



Comparar las imágenes resultantes tras aplicar 3 pasos del método de Jacobi y tres pasos del método de Gauss-Seidel.

Normas de entrega

1. Rellenar en la parte superior el nombre de los integrantes del grupo.
2. Compartir esta hoja de Sage con el profesor (etlopez18).
3. En el desplegable "File" de la parte superior de la página, elegir "Print"
4. Imprimir la página resultante como pdf.
5. Subirla al campus virtual antes de la fecha límite, junto con la memoria de planificación y trabajo en equipo. Cada día a partir de la fecha límite, la nota sobre la que se puntúa el ejercicio bajará en un punto.