Taller sistemas de ecuaciones

Pontificia Universidad Javeriana

Análisis Numérico

Docente: Eddy Herrera

Javier Esteban Flechas Barreto
Manuel Alejandro Rios Romero
Luis Felipe Ayala Urquiza
Andrés Felipe Otálora Jarro

2021 - 3

Punto 3

En este punto nos daban un modelo que que describe la cantidad de personas infectadas por un virus, el modelo se guía por el tiempo que transcurre en días, además de el modelo se nos da los valores de algunos datos y a continuación se nos pide determinar el día más cercano donde la cantidad de personas infectadas supere los 1500, 1800 y 2000.

Para la solución de este ejercicio utilizamos la función linalg.solve de la librería numpy, esta funciona para resolver sistemas de ecuaciones lineales, una vez decidida la función a utilizar agregamos la información que se nos fue suministrada de la siguiente forma.

```
a=np.array ([[10,100,4.481],[15,225,9.487],[20,400,20.085]])
b=np.array([25,130,650])
x=np.linalg.solve(a,b)
print(x)
```

Una vez ejecutada esta sección de código pudimos obtener los valores de las variables k que tenía el modelo, ya que conseguimos estos valores el siguiente pasó fue iterar hasta hallar una solución para cada uno de las cantidades requeridas.

```
e = math.e
i = 0
y = 23.000
objetivo = 2000
while i <= objetivo:

    i = (-17.31476212 * y) + (-2.24249889 * (y**2) ) +
    (94.26411739 * (e**(0.15*(y))))
    y = y + 0.0001

print ("Para el objetivo" , objetivo)
print ("el valor de t es aproximadamente", y, "El valor obtenido
con este t es igual a :", i)
```

Para cada solución solo tenemos que cambiar el valor objetivo, en la imagen este valor equivale a 2000, pero este método de iteración es capaz de encontrar cualquier valor superior a 1300, lo hicimos para cualquier valor mayor a 1300 y no desde 0 porque de está manera el programa es más eficiente, sabemos que Y puede empezar en el valor 23 que equivale más o menos a 1300 infectados y a partir de ahí iterar a cualquier objetivo superior.

Lo que se ve a continuación es la respuesta que arroja este programa completo al ser ejecutado.

```
[-17.31476212  -2.24249889  94.26411739]
Para el objetivo 2000
el valor de t es aproximadamente 24.618899999996227 El valor obtenido con este t
 es igual a : 2000.0377834530639
```

La primera línea nos muestra los valores de k, la siguiente línea el valor objetivo que se busca casi finalizando se nos muestra la cantidad de días representadas como t para llegar a un valor aproximado y por último el valor aproximado de infectados respecto a la cantidad de días t hallada.

Punto 4

Este punto solicitaba utilizar el metodo SOR con una precisión de 10^{-5} con un sistema de AX=B.

Dado el sistema $AX = B$, utilice el método de **SOR** con una precisión de 10^{-5}, donde $\mathbf{b} = b_i = \pi, \forall i = 1, ..., 80$ y las entradas de la matriz A estan dadas por

$$a_{i,j} = \begin{cases} 2i, & \text{when } j = i \text{ and } i = 1,2,\ldots,80, \\ 0.5i, & \text{when } \begin{cases} j = i+2 \text{ and } i = 1,2,\ldots,78, \\ j = i-2 \text{ and } i = 3,4,\ldots,80, \end{cases} \\ 0.25i, & \text{when } \begin{cases} j = i+4 \text{ and } i = 1,2,\ldots,76, \\ j = i-4 \text{ and } i = 5,6,\ldots,80, \end{cases} \\ 0, & \text{otherwise,} \end{cases}$$

El método SOR permite mejorar la convergencia a través de la relajación trabajando con una modificación del método de Gauss-Seidel. Primero se genera el código de la relajación en python.

```python
def relajacion(A,b,x,w,tol):
  n=len(x)

  for j in range(n):
      s=0
      for i in range(n):
        s=s+A[i][j]*x[i]
      x[j]=x[i]+w*(b[i]-s)/A[j][j]

  return x

A=np.ones((81,81))
b=np.ones(81)
x=np.ones(81)
tolerancia=10**-5
```

Después de esto se ingresan las condiciones en un ciclo del 1 hasta el 81. Todos los resultados que se realicen dentro del for se almacena en el arreglo de “B”.

```
for j in range(1,81):
  b[j]=math.pi
  for i in range(1,81):
    if(j==i and i>=1):A[i][j]=2
    elif(j==i+2 and i>=1 and i<=78):A[i][j]=0.5
    elif(j==i-2 and 1>=3):A[i][j]=0.5
    elif(j==i+4 and i>=1 and i<=76):A[i][j]=0.25
    elif(j==i-4 and i>=5 ):A[i][j]=0.25
    else :A[i][j]=0
```

Finalmente se realiza y se imprimen los resultados de la relajación.

```
x=relajacion(A,B,x,0.5,tolerancia);
print(x)
```

Para obtener los resultados de X que van del 1 al 80.

```
[-37.92920367  10.70519908  10.70519908   9.3670492    9.3670492
   8.86524299   8.86524299   9.01160313   9.01160313   9.024671
   9.024671     9.01389001   9.01389001   9.01442089   9.01442089
   9.01502834   9.01502834   9.01491923   9.01491923   9.01489491
   9.01489491   9.01490477   9.01490477   9.01490505   9.01490505
   9.0149044    9.0149044    9.01490447   9.01490447   9.0149045
   9.0149045    9.01490449   9.01490449   9.01490449   9.01490449
   9.01490449   9.01490449   9.01490449   9.01490449   9.01490449
   9.01490449   9.01490449   9.01490449   9.01490449   9.01490449
   9.01490449   9.01490449   9.01490449   9.01490449   9.01490449
   9.01490449   9.01490449   9.01490449   9.01490449   9.01490449
   9.01490449   9.01490449   9.01490449   9.01490449   9.01490449
   9.01490449   9.01490449   9.01490449   9.01490449   9.01490449
   9.01490449   9.01490449   9.01490449   9.01490449   9.01490449
   9.01490449   9.01490449   9.07740449   9.07740449   9.06959199
   9.06959199]
```

Punto 6

Este punto utiliza los métodos de Jacobi y Gauss seidel, para asegurar la convergencia de los mismos se utilizaron un α con valor de 3 y un β con un valor de 0, así implementamos la información que se va a utilizar:

```
a = np.array([[2, 0, -1], [0, 2, -1], [-1, 1, 3]])
b = np.array([1, 2, 1])
x0 = np.array([1, 2, 3])
```

Luego de esto realizamos cada uno de los métodos iterativos para encontrar la solución, en el caso de Jacobi utilizamos la tolerancia y el número de iteraciones como límite:

```
MDiag = np.diag(np.diag(A))
LU = A-MDiag
x = x0
for i in range (it):
    MDiagI = np.linalg.inv(MDiag)
    xAux = x
    x = (MDiagI @ (-LU) @ x) + MDiagI @ b
    print('Iteracion: ', i, ' Solucion: ',x)
    if(np.linalg.norm(x-xAux)<t):
        return x
return x
```

Para el método de Gauss:

```
iteracion = 0
x = x0
while iteracion<it:
  for i in range (len(A)):
    cont = 0
    for j in range (len(A)):
      if j != i:
        cont = cont + A[i][j]*x[j]
    xAux = (b[i] - cont)/A[i][i]
    x[i] = xAux
  iteracion=iteracion+1
  print('Iteracion: ', iteracion, ' Solucion: ',x)
return x
```

Luego de encontrar los valores se utiliza el método de Jacobi para encontrar la solución, este utiliza una tolerancia de 10^{-6} obtenemos el siguiente resultado:

```
Solucion por Jacobi:
Iteracion:  0  Solucion:  [2.  2.5 0. ]
Iteracion:  1  Solucion:  [0.5        1.         0.16666667]
Iteracion:  2  Solucion:  [0.58333333 1.08333333 0.16666667]
Iteracion:  3  Solucion:  [0.58333333 1.08333333 0.16666667]
```

El mismo procedimiento con el método de Gauss seidel, genera el siguiente resultado:

```
Solucion por Gauss Seidel:
Iteracion:  1  Solucion:  [2 2 0]
Iteracion:  2  Solucion:  [0 1 0]
Iteracion:  3  Solucion:  [0 1 0]
Iteracion:  4  Solucion:  [0 1 0]
Iteracion:  5  Solucion:  [0 1 0]
Iteracion:  6  Solucion:  [0 1 0]
Iteracion:  7  Solucion:  [0 1 0]
Iteracion:  8  Solucion:  [0 1 0]
Iteracion:  9  Solucion:  [0 1 0]
Iteracion:  10  Solucion:  [0 1 0]
```

Punto 10

Para este punto se pedía resolver un sistema de 2 ecuaciones con 2 variables para un valor inicial de (1,1) por medio de la implementación del método de Newton Raphson para múltiples variables.

Este método consiste en multiplicar la matriz de ecuaciones con su Jacobiana inversa negativa. A esta matriz resultante que llamaremos N se le suman los valores iniciales de la iteración y de esta forma se obtienen los nuevos valores de las variables para la siguiente iteración. Este proceso se repite hasta que la raíz de la suma de los valores de la matriz N elevados al cuadrado sea menor que la Tolerancia.

Se define la matriz de ecuaciones, su Jacobiana y Jacobiana inversa.

```
#DEFINICION DEL SISTEMA DE 2 ECUACIONES
def matrizP():
  f1= x**2+ y**2 -1
  f2= y - x
  return Matrix([[f1],[f2]])
#DEFINICIO DEL JACOBIANO DE 2 ECUACIONES
def JacobianaP():
  f1= x**2+ y**2 -1
  f2= y - x
  J= Matrix([[Derivative(f1,x).doit(),Derivative(f1,y).doit()],[Derivative(f2,x).doit(),Derivative(f2,y).doit()]])
  Jinv= J.inv()
  return [J, Jinv]
```

Se crea un ciclo con un número máximo de iteraciones igual al k designado. (Azul)

Se reemplaza los valores de las variables dentro de las matrices (Rojo)

Se calcula la matriz N y los nuevos Valores de las variables para el siguiente ciclo.

```
while k<iter:
  #Sustituir valores en las matrices
  MSub2 = M2.subs({x:d,y:f})
  JSub2 = J2.subs({x:d,y:f})
  JinvSub2 = Jinv2.subs({x:d,y:f})
  #Verificar substitucion
  '''
  pprint(MSub)
  print("\n")
  pprint(JSub)
  print("\n")
  pprint(JinvSub)
  print("\n")
  '''
  #Calculos
  N=-JinvSub2*MSub2
  #pprint(N)
  X=Matrix([d, f]) + N
  #pprint(X)
  d2 =float(X[0,0])
  f2=float(X[1,0])
  d,f=d2,f2
  error2=float(sqrt(N[0,0]**2+ N[1,0]**2))
  k+=1
  if(error2 < TOL):
    print("\nSe soluciono el metodo")
    print("{0:1d} \t {1:1.8f} \t {2:1.8f} \t {3:1.12f}".format(k,d2,f2,error2))
    break;
  if(k==iter):
    print("\n Se alcanzo el numero maximo de iteraciones")
  print("{0:1d} \t {1:1.8f} \t {2:1.8f} \t {3:1.12f}".format(k,d2,f2,error2))
```

Finalmente se colocan dos condiciones, la primera nos indica que si el error es menor a la tolerancia asignada se muestra que se encontró la solución y se imprime el resultado de la última iteración que lleva el valor encontrado. El segundo es en caso de que se alcance el número máximo de iteraciones designadas sin que el error sea menor a la tolerancia, en cuyo caso se indica que no se alcanzó este número y se imprime el resultado de la última iteración.

Para el caso de una circunferencia x**2 + y**2 =1 que se conecta con la recta y=x, el Método de Newton Raphson nos dice que el valor que en el que se intersectan cercano al punto (1,1) es:

```
Matriz ecuaciones
⎡ 2    2    ⎤
⎢x  + y  - 1⎥
⎢           ⎥
⎣  -x + y   ⎦


Matriz Jacobiana
⎡2⋅x  2⋅y⎤
⎢        ⎥
⎣-1    1 ⎦


Matriz Jacobiana inversa
⎡    1         -y   ⎤
⎢──────────   ──────⎥
⎢2⋅x + 2⋅y   x + y  ⎥
⎢                   ⎥
⎢   -1          x   ⎥
⎢──────────   ──────⎥
⎣-2⋅x - 2⋅y   x + y ⎦


Resultado Metodo de Newton Multivariable
iter      X              Y           Error
1    0.75000000      0.75000000      0.353553390593
2    0.70833333      0.70833333      0.058925565099
3    0.70710784      0.70710784      0.001733104856
4    0.70710678      0.70710678      0.000001501824

Se soluciono el metodo
5    0.70710678      0.70710678      0.000000000001
FIN
```

El cual se puede verificar que es correcto con Wolfram: