Informe 4. Urano App

Ronald Cardona   Anderson Grajales
Sebastian Valencia   Julian Sanchez

29 de octubre de 2018

1. Funciones de apoyo

Esta seccion muestra algunas funciones que son necesarias para lograr una correcta implementacion de los metodos para la solucion numerica de sistemas de ecuaciones lineales.

1.1. Determinantes

Definición 1.1. *Sea $A = [a_{ij}]$ una matriz de tamaño $n \times n$. El cofactor C_{ij} de a_{ij} se define como $(-1)^{i+j} \det M_{ij}$, donde M_{ij} es la matriz de tamaño $(n-1) \times (n-1)$, que se obtiene al eliminar la fila i y la columna j de la matriz.[1]*

Teorema 1.1. *Sea $A = [a_{ij}]$ una matriz de tamaño $n \times n$. [1]*

- *Para cada $1 \leq i \leq n$ se cumple que:* $\det A = a_{i1}C_{i1} + a_{i2}C_{i2} + ... + a_{in}C_{in}$
- *Para cada $1 \leq j \leq n$ se cumple que:* $\det A = a_{1j}C_{1j} + a_{2j}C_{2j} + ... + a_{nj}C_{nj}$

De acuerdo al teorema 1.1 se puede definir una ecuación de recurrencia para encontrar el determinante de una matriz $A = [a_{ij}]$ de la siguiente manera:

$$det(A, n)_{1 \leq i \leq n} = \begin{cases} a_{11}, & \text{if } n = 1. \\ (-1)^{i+1} \times a_{1i} \times det(A', n-1), & \text{if } n > 1. \end{cases} \quad (1)$$

Donde A' es la matriz que se obtiene al eliminar la columna i y la fila n de A. De esta manera, para una matriz B de $n \times n$, la solución se entrega de la forma: $det(B, n)$.

1.2. Multiplicación de matrices

Definición 1.2. *Dadas las matrices* $A \in M_{m\times n}$ *y* $B \in M_{n\times p}$*, entonces el producto de* A *con* B*, denotado* AB*, es una matriz* $C \in M_{m\times p}$*, dada por: [1]*

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + a_{i3}b_{3j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj}$$

con $i = 1, ..., m$ *y* $j = 1, ..., p$

1.3. Escalonamiento de matrices

Definición 1.3. *Sea* $A = [a_{ij}]$ *una matriz de* $n \times n$*. Decimos que* A *está escalonada si* $\forall i, j, 1 \leq i \leq j \leq n$*,* $a_{ij} = 0$*.*

```
Leer A, b
si A ∉ ℜ^{n×n} ó b ∉ ℜ^n entonces
 |  A debe ser cuadrada y b debe ser un arreglo de n posiciones
si no, si det A = 0 entonces
 |  A debe ser invertible
en otro caso
    para k = 1 a n − 1 hacer
        si A_kk = 0 entonces
            j ← k + 1
            mientras j < n y A_jk = 0 hacer
             |  j ← j + 1
            fin
            si j < n entonces
                para l = k a n hacer
                 |  A_kl ← A_kl + A_jl
                fin
                b_k ← b_k + b_j
            fin
        fin
        para i = k + 1 a n hacer
            si A_ki ≠ 0 entonces
                m ← A_ik / A_kk
                para l = k a n hacer
                 |  A_il ← A_il − m × A_kl
                fin
                b_i ← b_i − m × b_k
            fin
        fin
    fin
fin
La solución (A, b)
```

Algoritmo 1: Algoritmo para escalonar matrices

1.4. Sustitución Regresiva

Leer A, b
$marks \leftarrow NULL$
$n \leftarrow Len(A) - 1$
$x \leftarrow 0$
$x_n \leftarrow \frac{b_n}{A_{nn}}$
para $i \leftarrow n-1$ **a** -1 **hacer**
  $sumatoria \leftarrow 0$
  para $p \leftarrow i+1$ **a** $n+1$ **hacer**
    $sumatoria \leftarrow sumatoria + A_{ip} * x_p$
  fin
  $x_i \leftarrow b_i$
fin
si $marks \neq NULL$ **entonces**
  $marcas[Len(A)] \leftarrow 0$
  para $i \leftarrow 0$ **a** $Len(A)$ **hacer**
    $marcas_{marks_i} \leftarrow x_i$
  fin
  Retornar $marcas$
Retornar x

Algoritmo 2: Algoritmo de Sustitución Regresiva

1.5. Sustitución Progresiva

Leer A, b
$n \leftarrow Len(A) - 1$
$x \leftarrow 0$
$x_0 \leftarrow \frac{b_0}{A_0 0}$
para $i \leftarrow 1$ **a** $n+1$ **hacer**
  $sumatoria \leftarrow 0$
  para $p \leftarrow 0$ **a** i **hacer**
    $sumatoria \leftarrow sumatoria + A_{ip} * x_p$
  fin
  $x_i \leftarrow \frac{b_i - sumatoria}{A_{ii}}$
fin
Retornar x

Algoritmo 3: Algoritmo de Sustitución Progresiva

2. Solucion Numerica de Sistemas de Ecuaciones Lineales

Muchos problemas del mundo real se formulan como sistemas de ecuaciones de n variables y m incógnitas que bajo condiciones ideales(n y m no son valores muy grandes), se pueden resolver de manera analítica. Sin embargo, cuando n y m tienden a ser valores muy grandes la solución analítica a estos problemas es muy difícil de calcular ya que requiere de mucho tiempo y claramente no es la forma más eficiente hacerlo. Debido a esto, desde el campo del *Análisis Numérico* se plantean diversas formas computacionales, ya sean algoritmos u otras técnicas que nos permitan resolver estos sistemas rápidamente, teniendo en cuenta que hay una ***propagación de error*** en cada cálculo dependiendo de la capacidad de la computadora donde se ejecuten estos. En esta seccion se presentan algunos algoritmos numéricos que son de gran ayuda a la hora de resolver sistemas de ecuaciones lineales.

2.1. Eliminacion Gaussiana con Pivoteo Parcial

Leer A, b, n, k
$mayor \leftarrow |A_{kk}|$
$filaMayor \leftarrow k$
para $s \leftarrow k+1$ **a** n **hacer**
  si $|A_{sk}| > mayor$ **entonces**
    $mayor \leftarrow |A_{sk}|$
    $filaMayor \leftarrow s$
fin
si $mayor = 0$ **entonces**
  El sistema no tiene solucion unica
si no, si $filaMayor \neq k$ **entonces**
  $temp \leftarrow A_{filaMayor}$
  $A_{filaMayor} \leftarrow temp$
  $A_k \leftarrow A_{filaMayor}$
  $temp \leftarrow b_{filaMayor}$
  $b_{filaMayor} \leftarrow b_k$
  $b_k \leftarrow b_{filaMayor}$
Retornar(A, b)

Algoritmo 4: Algoritmo de Pivoteo Parcial

2.2. Eliminacion Gaussiana con Pivoteo Total

```
Leer A, b, n, k
mayor ← |A_kk|
filaMayor ← k
columnaMayor ← k
para r ← k a n hacer
    para s ← k a n hacer
        si |A_rs| > mayor entonces
            mayor ← |A_rs|
            filaMayor ← r
            columnaMayor ← s
    fin
fin
si mayor = 0 entonces
    El sistema no tiene solucion unica
en otro caso
    si filaMayor ≠ k entonces
        temp ← A_k
        A_k ← A_filaMayor
        A_filaMayor ← temp
        temp ← b_k
        b_k ← b_filaMayor
        b_filaMayor ← temp
    si columnaMayor ≠ k entonces
        temp ← A_0columnaMayor
        A_0columnaMayor ← A_0k
        A_0k ← temp
        temp ← b_0columnaMayor
        b_0columnaMayor ← b_0k
        b_0k ← temp
        temp ← marcas_k
        marcas_k ← marcas_columnaMayor
        marcas_columnaMayor←temp
    Retornar(A, b)
```

Algoritmo 5: Algoritmo de Pivoteo Total

3. Metodos de Factorizacion LU

Dada una martiz cuadrada A de orden nxn, se halla una matriz L triangular inferior y una matriz U triangular superior tal que $A = LU$ Este tipo de sistemas se resuelven de manera trivial haciendo uso de los ya conocidos metodos de sustitucion regresiva y sustitucion progresiva, ya que las matrices son triangulares.

3.1. Factorizacion LU con Gaussiana Simple

En este caso la matriz U corresponde a la matriz A en su forma escalonada. Y la matriz L se forma ubicando 1's en la diagonal y los multiplicadores M_{ij} en las entradas correspondientes.

Leer A, b
$(L, U) \leftarrow Escalonar(A, b)$
$z \leftarrow SustitucionProgresiva(L, b)$
$x \leftarrow SustitucionRegresiva(U, z)$
Retornar x

Algoritmo 6: Algoritmo de Factorizacion LU con Gaussiana Simple

3.2. Factorizacion LU con Gaussiana y Pivoteo Parcial

La matriz L se construye con base en los multiplicadores ubicados segun sus respectivos indices y con 1's en la diagonal, y la matriz U es la matriz resultante del oproceso de eliminacion

Leer A, b
$(L, U) \leftarrow EscalonarParcial(A, b)$
$z \leftarrow SustitucionProgresiva(L, b)$
$x \leftarrow SustitucionRegresiva(U, z)$
Retornar x

Algoritmo 7: Algoritmo de Factorizacion LU con Gaussiana y Pivoteo Parcial

3.3. Factorizacion de Doolittle

```
Leer A, b
si A no es cuadrada entonces
|  Retornar Matriz no cuadrada
n = longitud(A_0)
L ← MatrizIdentidad(n)
U ← MatrizIdentidad(n)
para i ← 0 a n hacer
|  para k ← i a n hacer
|  |  u ← A_ik
|  |  para numero ← 0 a i hacer
|  |  |  u ← u − L_{i,numero} * U_{numero,k}
|  |  fin
|  |  L_ik ← u/L_ii
|  fin
|  para j ← i + 1 a n hacer
|  |  suma ← A_ji
|  |  para numero ← 0 a j hacer
|  |  |  suma ← suma − L_{j,numero} * U_{numero,i}
|  |  fin
|  |  L_ji ← suma / U_ii
|  fin
fin
z ← SustitucionProgresiva(L, b)
x ← SustitucionRegresiva(U, z)
Retornar x
```

Algoritmo 8: Algoritmo de Factorizacion de Doolittle

3.4. Factorizacion de Choletsky $A = LDL$

Leer A
$n \leftarrow longitud(A)$
$L[n][n] \leftarrow 0$
$U[n][n] \leftarrow 0$
para $k \leftarrow 0$ **a** n **hacer**
    $suma_p \leftarrow 0{,}0$
    para $p \leftarrow 0$ **a** k **hacer**
        $suma1 \leftarrow L_{kp} * U_{pk}$
    fin
    $L_{kk} \leftarrow (A_{kk} - suma1)^{0,5}$
    $U_{kk} \leftarrow L_{kk}$
    para $i \leftarrow k$ **a** n **hacer**
        $suma2 \leftarrow 0{,}0$
        para $p \leftarrow 0$ **a** k **hacer**
            $suma2 \leftarrow suma2 + L_{ip} * U_{pk})$
        fin
        $L_{ik} \leftarrow \frac{A_{ik} - suma2}{U_{kk}}$
    fin
    para $j \leftarrow k + 1$ **a** n **hacer**
        $suma3 \leftarrow 0$
        para $p \leftarrow 0$ **a** k **hacer**
            $suma3 \leftarrow suma3 + L_{kp} * U_{pj})$
        fin
        $U_{kj} \leftarrow \frac{A_{kj} - suma3}{L_{kk}}$
    fin
fin
Retornar L, U

Algoritmo 9: Algoritmo de Factorizacion de Choletsky

3.5. Factorizacion de Crout $A = LDU$

Leer A, b
si A *no es cuadrada* **entonces**
  Retornar Matriz no cuadrada
$n = longitud(A_0)$
$L \leftarrow 0$
$U \leftarrow MatrizIdentidad(n)$
para $i \leftarrow 0$ **a** n **hacer**
  para $j \leftarrow i$ **a** n **hacer**
    $suma \leftarrow A_{ji}$
    para $numero \leftarrow 0$ **a** j **hacer**
      $suma \leftarrow suma - L_{j,numero} * U_{numero,i}$
    fin
    $L_{ji} \leftarrow suma$
  fin
  para $k \leftarrow i + 1$ **a** n **hacer**
    $u \leftarrow A_{ik}$
    para $numero \leftarrow 0$ **a** i **hacer**
      $u \leftarrow u - L_{i,numero} * U_{numero,k}$
    fin
    $L_{ik} \leftarrow \frac{u}{L_{ii}}$
  fin
fin
$z \leftarrow SustitucionProgresiva(L, b)$
$x \leftarrow SustitucionRegresiva(U, z)$
Retornar x

Algoritmo 10: Algoritmo de Factorizacion de Crout

4. Metodos Indirectos

Para encontrar soluciones a un sistema de la forma $Ax = b$, encontraremos vectores $x^{(i)}$, aproximaciones a la solucion, a partir de un vector inicial $x^{(0)}$, hasta que se cumpla cierta tolerancia respecto a una norma establecida. Cada $x^{(i)}$ se genera a partir de una funcion analoga a la funcion de punto fijo $G(x) = x$.

4.1. Metodo de Jacobi

```
Leer A, b, tol, x0, niter
cont ← 1
dispersion ← tol + 1
solucion.add((0, x0))
mientras dispersion > tolANDcont < niter hacer
 | x1 ← calcularNuevoJacobi(A, b, x0)
 | dispersion ← normaCuadrada(x1, x0)
 | ss.add((cont, x1, dispersion))
 | x0 ← x1
 | solucion.add(ss)
 | cont ← cont + 1
fin
Retornar solucion
```

Algoritmo 11: Algoritmo del metodo de Jacobi

```
calcularNuevoJacobi(A, b, x0)
Leer A, b, x0
n ← longitud(x0)
para i ← 0 a n hacer
 | suma ← 0,0
 | para j ← 0 a n hacer
 |  | si j ≠ i entonces
 |  |  | suma ← suma + A_ij * x0_j
 |  | fin
 | fin
 | x_i ← (b_i − suma) / A_ii
fin
Retornar x
```

Algoritmo 12: Algoritmo para calcular el nuevo Jacobi

normaCuadrada($x1$, $x0$)
Leer $x1$, $x0$
$suma1 \leftarrow 0{,}0$
$suma2 \leftarrow 0{,}0$
para $i \leftarrow 0$ **a** $longitud(x1)$ **hacer**
  $suma1 \leftarrow suma1 + (x1_i - x0_i)^2$
  $suma2 \leftarrow suma2 + x1_i^2$
fin
Retornar $\sqrt{\frac{sum1}{sum2}}$

Algoritmo 13: Algoritmo Norma Cuadrada

4.2. Metodo de Gauss-Seidel

Leer A, b, tol, $x0$, $niter$
$cont \leftarrow 1$
$dispersion \leftarrow tol + 1$
$solucion.add((0, x0))$
mientras $dispersion > tolANDcont < niter$ **hacer**
  $x1 \leftarrow calcularNuevoGaussSeidel(A, b, x0)$
  $dispersion \leftarrow normaCuadrada(x1, x0)$
  $ss.add((cont, x1, dispersion))$
  $x0 \leftarrow x1$
  $solucion.add(ss)$
  $cont \leftarrow cont + 1$
fin
Retornar $solucion$

Algoritmo 14: Algoritmo del metodo de Gauss-Seidel

calcularNuevoGaussSeidel(A, b, $x0$)
Leer A, b, $x0$
$n \leftarrow longitud(x0)$
$x \leftarrow x0$
para $i \leftarrow 0$ **a** n **hacer**
    $suma \leftarrow 0{,}0$
    para $j \leftarrow 0$ **a** n **hacer**
        si $j \neq i$ **entonces**
            $suma \leftarrow suma + A_{ij} * x_j$
        fin
    fin
    $x_i \leftarrow \frac{b_i - suma}{A_{ii}}$
fin
Retornar x

Algoritmo 15: Algoritmo para calcular el nuevo GaussSeidel

5. Metodos Iterativos de Forma Matricial

5.1. Gauss-Seidel con relajacion

Leer A, b, tol, $x0$, w, $niter$
$cont \leftarrow 1$
$solucion.add((0))$
mientras $dispersion > tolANDcont < niter$ **hacer**
    $x1 \leftarrow calcularNuevoGaussSeidelSOR(A, b, x0, w)$
    $dispersion \leftarrow normaMaximo(x1, x0)$
    $x0 \leftarrow x1$
    $solucion.add((cont, x1, dispersion))$
    $cont \leftarrow cont + 1$
fin
Retornar $solucion$

Algoritmo 16: Algoritmo del metodo SOR Gauss-Seidel

```
calcularNuevoGaussSeidelSOR(A, b, x0, w)
Leer A, b, x0, w
n ← longitud(x0)
x ← x0
para i ← 0 a n hacer
    suma ← 0,0
    para j ← 0 a n hacer
        si j ≠ i entonces
            suma ← suma + A_ij * x_j
        fin
    fin
    x_i ← ((1−w)*(x_i+w)*(b_i−suma)) / A_ii
fin
Retornar x
```

Algoritmo 17: Algoritmo para calcular el nuevo Gauss-Seidel

```
normaMaximo(x1, x0)
Leer x1, x0
max1 ← 0,0
max2 ← 0,0
para i ← 0 a longitud(x1) hacer
    max1 ← max(|x1_i − x0_i|, mx1)
    max2 ← max(|x1_i|, max2)
fin
Retornar max1/max2
```

Algoritmo 18: Algoritmo Norma Maximo

6. Metodos de Interpolacion

6.1. Metodos basados en sistemas de ecuaciones

Teorema 6.1. *Dados $n+1$ puntos con la condicion de que $x_i \neq x_j$ para todo i, j tal que $0 <= i, j <= n$, entonces existe un polinomio $p(x)$ de grado a lo sumo n tal que para todo i, $0 <= i <= n$, se cumple que $p(x_i) = y_i$*

Dado un conjunto con $n+1$ puntos conocidos, entonces por el teorema anterior, existe un unico polinomio interpolante $p(x)$ de grado a lo sumo n. Luego, se puede considerar que el polinomio tiene la forma

$$p(x) = a_n x^n + a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + ... + a_2x^2 + a_1x + a_0$$

Para obtener el polinomio basta determinar el valor de todos los coeficientes

$$a_n, a_{n-1}, a_{n-2}, ..., a_2, a_1, a_0$$

Esto genera un sistema de ecuaciones lineales que siempre es soluble.

```
Leer puntos
definir A, b, auxA
n ← longitud(puntos)
para punto ← puntos hacer
|  b.add(punto_1)
|  para i ← 1 a n+1 hacer
|  |  auxA.add((punto_0)^(n-i))
|  fin
|  A.add(auxA)
fin
Ab ← EscalonarMatriz(A, b)
x ← SustitucionRegresiva(A, b)
Retornar (A, b)
```

Algoritmo 19: Algoritmo para obtener el polinomio interpolante por medio de la matriz de Vandermonde

6.2. Polinomio Interpolante de Newton con Diferencias Divididas

Dados $n+1$ puntos $(x_0, y_0), (x_1, y_1), ..., (x_n, y_n)$ el polinomio del teorema de Interpolacion se puede escribir asi:

$$p(x) = b_0 + b_1(x - x_0) + b_2(x - x_0)(x - x_1) + ... + b_n(x - x_0)(x - x_1)...(x - x_{n-1})$$

Asi para hallar el polinomio $p_n(x)$ basta con allar los b_n. Supongamos que se conoce el siguiente conjunto de puntos $(x_0, y_0), (x_1, y_1), ..., (x_n, y_n)$

- Diferencia dividida de orden 0
  $f[x_k] = f(x_k)$
  $b_0 = f[x_0]$

- Primera diferencia dividida
  $f[x_k, x_{k+1}] = \frac{f[x_{k+1}]-f[x_k]}{x_{k+1}-x_k}$
  $b_1 = f[x_0, x_1]$

- n-esima diferencia dividida
  $f[x_k, ..., x_{k+n}] = \frac{f[x_{k+1},x_{k+2},...,x_{k+n}]-f[x_k,x_{k+1},...,x_{k+n}]}{x_{k+n}-x_k}$
  $b_n = f[x_0, x_1, ..., x_n]$

Leer $nPuntos, valor, x, y$
$tabla[nPuntos][nPuntos]$
para $i \leftarrow 0$ **a** $nPuntos$ **hacer**
  $tabla_i0 \leftarrow y_i$
  para $j \leftarrow 1$ **a** $i+1$ **hacer**
    $tabla_{ij} \leftarrow \frac{tabla_{i,j-1}-tabla_{i-1,j-1}}{x_i-x_{i-j}}$
  fin
fin
Retornar $tabla$

Algoritmo 20: Algoritmo para obtener la tabla de diferencias divididas

n	x_i	$f[x_i]$	$1ra$	$2da$	$3ra$	$4ta$	$5ta$
0	1	0.6747					
1	1.2	0.8491	0.8723				
2	1.4	1.1214	1.3610	1.2218			
3	1.6	1.4921	1.8536	1.2314	0.0160		
4	1.8	1.9607	2.3429	1.2233	-0.0134	-0.0368	
5	2.0	2.5258	2.8258	1.2070	-0.0272	-0.0172	0.0195

A partir de esta tabla el polinomio se obtiene facilmente
$p(x) = 0{,}6747 + 0{,}8723(x-1) + 1{,}2218(x-1)(x-1{,}2) + 0{,}0160(x-1)(x-1{,}2)(x-1{,}4) - 0{,}0368(x-1)(x-1{,}2)(x-1{,}4)(x-1{,}6) + 0{,}0195(x-1)(x-1{,}2)(x-1{,}4)(x-1{,}6)(x-1{,}8)(x-2)$

6.3. Polinomio de Lagrange

Dados $n+1$ puntos $(x_0, y_0), (x_1, y_1), ..., (x_n, y_n)$ el polinomio de Lagrange tiene la siguiente forma:

$$\sum_{k=0}^{n} L_k(x) f(x_k)$$

donde

$$L_k(x) = \frac{(x-x_0)(x-x_1)...(x-x_{k-1})(x-x_{k+1})...(x-x_n)}{(x_k-x_0)(x_k-x_1)...(x_k-x_{k-1})(x_k-x_{k+1})...(x-x_n)}$$

Leer $nPuntos, valor, x, y$
$resultado \leftarrow 0$
$pol \leftarrow' P(x) =' $
para $k \leftarrow 0$ **a** $nPuntos$ **hacer**
  $mult \leftarrow nPuntos$
  para $i \leftarrow 0$ **a** $nPuntos$ **hacer**
    si $i \neq k$ **entonces**
      $mult \leftarrow mult * \frac{valor-x_i}{x_k-x_i}$
      $num \leftarrow (x - x_i)$
      $den \leftarrow x_k - x_i$
      $term \leftarrow term +' [(num/den)]'$
    fin
  fin
  Imprimir $'L_k = y_k * term'$
  $pol \leftarrow pol +' y_k * term'$
  $resultado = resultado + mult * y_k$
fin
Imprimir pol
Retornar $resultado$

Algoritmo 21: Algoritmo para obtener el polinomio de Lagrange

Referencias

[1] Orlando García Jaimes, Jairo A. Villegas Gutiérrez, Jorge Iván. *Álgebra Lineal*. Editorial EAFIT, Medellín 2012.