Tarea 03 - Solución de sistemas de ecuaciones

Isaac Rodríguez Bribiesca

Resumen Implementación de algoritmos para la solución de sistemas de ecuaciones

$$A * x = y \tag{1}$$

y cálculo de determinantes para los siguientes casos de la matriz A: Diagonal, Triangular superior, Triangular inferior, Eliminación Gaussiana, Eliminación Gaussiana con pivoteo, Doolitle y Cholesky modificado. Así como el cálculo de matriz inversa mediante Factorización Doolitle.

1. Descripción métodos

1.1. Caso matriz A diagonal

$$A = \begin{pmatrix} a_{11} & 0 & \dots & 0 \\ 0 & a_{22} & & \\ \vdots & & \ddots & \\ 0 & & & a_{nn} \end{pmatrix} \tag{2}$$

En el caso de una matriz diagonal la solución del sistema (1) se puede encontrar mediante la siguiente ecuación:

$$x_i = \frac{b_i}{a_{ii}} \tag{3}$$

Por lo que el pseudocódigo para su implementación queda de la siguiente manera:

Algorithm 1: SolveDiagonal(A, b)

```
Data: Matrix A, Vector b
Result: Vector solución x
Crea vector x_{nx1};
for i desde 1 hasta n do
    x[i] = b[i]/A[i][i];
end
```

1.2. Caso matriz A triangular inferior

$$A = \begin{pmatrix} a_{11} & 0 & \dots & 0 \\ a_{21} & a_{22} & & 0 \\ \vdots & & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} \tag{4}$$

En el caso de una matriz triangular inferior la solución del sistema (1) se puede encontrar mediante la siguiente ecuación:

$$x_i = \frac{b_i - \sum_{k=1}^{i-1} a_{ij} * x_j}{a_{ii}} \tag{5}$$

Excepto en el caso de x_0 para la cual su solución será:

$$x_0 = \frac{b_0}{a_{00}} \tag{6}$$

Por lo que el pseudocódigo para su implementación queda de la siguiente manera:

Algorithm 2: SolveLowerTriangular(A, b)

```
Data: Matrix A, Vector b
Result: Vector solución x
Crea vector x_{nx1};
Crea variable auxiliar suma;
x[0] = b[0]/A[0][0];
for i desde 2 hasta n do
    suma = 0;
    for j desde 1 hasta i-1 do
        suma = suma + A[i][j] * x[j];
    end
    x[i] = ( b[i] - suma ) / ( A[i][i] );
end
```

1.3. Caso matriz A triangular superior

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ 0 & a_{22} & \dots & a_{2n} \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \dots & a_{nn} \end{pmatrix} \tag{7}$$

En el caso de una matriz triangular superior la solución del sistema (1) se puede encontrar mediante la siguiente ecuación:

$$x_i = \frac{b_i - \sum_{k=i+1}^{n} a_{ij} * x_j}{a_{ii}} \tag{8}$$

Excepto en el caso de x_n para la cual su solución será:

$$x_n = \frac{b_n}{a_{nn}} \tag{9}$$

Por lo que el pseudocódigo para su implementación queda de la siguiente manera:

Algorithm 3: SolveUpperTriangular(A, b)

Data: Matrix A, Vector b
Result: Vector solución x
Crea vector x_{nx1};
Crea variable auxiliar suma;
x[n] = b[n]/A[n][n];
for *i desde n-1 hasta 1* **do**
    suma = 0;
    for *j desde i+1 hasta n* **do**
        suma = suma + A[i][j] * x[j];
    end
    x[i] = (b[i] - suma) / (A[i][i]);
end

1.4. Caso general matriz A mediante Gauss sin pivoteo

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} \tag{10}$$

Para el método de eliminación Gaussiana sin pivoteo, el objetivo es dejar a la matriz A en forma traingular superior. Una vez que A se deja en dicha forma, la solución del sistema (1) se puede encontrar mediante las ecuaciones (8) y (9).

La eliminación Gaussiana consta de 3 partes:
1.- Se calcula el pivote $p_i = \frac{a_{ji}}{a_{ii}}$
2.- Se reemplaza el elemento a_{jk} por $a_{jk} - a_{ji} * aik$ para k desde i+1 hasta n. Lo cual hará cero a todos los elementos de A de la columna i.
3.- Finalmente, se aplican los cambios que se han hecho en A al vector b, para mantener la igualdad de la ecuación (1). Estos cambios que se requieren hacer al vector b se guardan en la misma matriz A, aprovechando el espacio de las a_{ij} que se vuelven cero.

Por lo que el pseudocódigo para su implementación queda de la siguiente manera:

Algorithm 4: SolveGaussEliminationNoPivoting(A, b)

```
Data: Matrix A, Vector b
Result: Vector solución x
Crea vector x_{nx1};
for i desde 0 hasta n-1 do
    if A[i][i] = 0 then
        Terminar programa. No se puede dividir entre 0;
    else
        for j desde i+1 hasta n do
            // Calcula pivote;
            A[j][i] = A[j][i] / A[i][i];
            // Aplica pivote;
            for k desde i+1 hasta n do
                A[j][k] = A[j][k] - A[j][i] * A[i][k];
            end
        end
    end
end
// Aplica transformaciones a vector b;
for i desde 0 hasta n-1 do
    for j desde i+1 hasta n do
        b[j] = b[j] - A[j][i] * b[i];
    end
end
x = SolveUpperTriangular(A, b);
```

1.5. Caso general matriz A mediante Gauss con pivoteo

Para el método de eliminación Gaussiana con pivoteo también se resuelve el caso general de A en (10), donde el objetivo es dejar a la matriz A en forma traingular superior. Una vez que A se deja en dicha forma, la solución del sistema (1) se puede encontrar mediante las ecuaciones (8) y (9). Con la diferencia de que se realizan cambios de renglones y/o columnas, de ser necesario, para que el pivote sea siempre el elemento mayor de la matriz.

La eliminación Gaussiana con pivoteo consta de las mismas partes que el Algoritmo 4, con la diferencia en que el pivote se calcula encontrando la posición del máximo elemento e intercambiando renglones y columnas para que este quede en la posición actual.

Por lo que el pseudocódigo para su implementación queda de la siguiente manera:

Algorithm 5: SolveGaussEliminationNoPivoting(A, b)

```
Data: Matrix A, Vector b
Result: Vector solución x
Crea vector x_{nx1};
Crea indices imax,jmax donde se encuentra el elemento máximo de A;
for i desde 0 hasta n-1 do
    imax,jmax = maxElemIndex(A, i);
    pocisionaPivote(A, b, i, imax, jmax);
    if A[i][i] = 0 then
        Terminar programa. No se puede dividir entre 0;
    else
        for j desde i+1 hasta n do
            // Calcula pivote;
            A[j][i] = A[j][i] / A[i][i];
            // Aplica pivote;
            for k desde i+1 hasta n do
                A[j][k] = A[j][k] - A[j][i] * A[i][k];
            end
        end
    end
end
// Aplica transformaciones a vector b;
for i desde 0 hasta n-1 do
    for j desde i+1 hasta n do
        b[j] = b[j] - A[j][i] * b[i];
    end
end
x = SolveUpperTriangular(A, b);
```

1.6. Caso general matriz A mediante Doolittle

La factorización de la matriz A en (10) mediante Doolittle consiste en representarla como una multiplicación de 2 matrices L y U. Donde L es una matriz triangular inferior con 1's en la diagonal principal, mientras que U es una matriz diagonal superior. Dando como resultado $A = L * U$.

Esto nos es útil ya que expresar A como LU nos permite resolver el sistema (1) de manera más eficiente haciendo uso de los algoritmos 2 y 3 ya implementados para resolver sistemas triangulares superiores e inferiores.

Es decir, tendremos el sistema

$$L * U * x = b \tag{11}$$

.

Ahora podemos renombrar

$$U * x = y \tag{12}$$

.

Con lo que (11) se convierte en

$$L * y = b \tag{13}$$

.

Ahora resolvemos el sistema (13) obteniendo y, para que después mediante (12) podamos obtener x.

Respecto a la descomposición de Doolittle se obtienen U y L respectivamente con las siguientes ecuaciones:

$$u_{ij} = aij - \sum_{k=0}^{i-1} l_{ik} * u_{kj} \tag{14}$$

$$l_{ji} = \frac{aji - \sum_{k=0}^{i-1} l_{jk} * u_{ki}}{u_{ii}} \tag{15}$$

Por lo que el pseudocódigo para su implementación queda de la siguiente manera:

Algorithm 6: SolveDoolittle(A, b)

```
Data: Matrix A, Vector b
Result: Vector solución x
Crea vector x_{nx1};
Crea vector y_{nx1};
for i desde 1 hasta n do
    // Calcula renglon i de U;
    for j desde i hasta n do
        // U[i][j] = A[i][j] - Sum(k=0 to i-1) L[i][k] * U[k][j];
        for k desde 1 hasta i do
            A[i][j] = A[i][j] - A[i][k] * A[k][j];
        end
    end
    // Calcula columna i de L;
    for j desde i+1 hasta n do
        // L[j][i] = A[j][i]/U[i][i] - Sum(k=0 to i-1) L[j][k] * U[k][i] /U[i][i];
        A[j][i] = A[j][i]/A[i][i];
        for k desde 0 hasta i do
            A[j][i] = A[j][i] - (A[j][k] * A[k][i])/A[i][i];
        end
    end
end
y = SolveLowerTriangular(L, b);
x = SolveUpperTriangular(U, y);
```

1.7. Algoritmo para calcular matriz inversa de A

Si se conoce la matriz inversa A^{-1} se debe cumplir la igualdad

$$A * A^{-1} = I \tag{16}$$

donde I es la matriz identidad de nxn.

Pero también podemos interpretar la ecuación (16) de la siguiente manera:

Tomamos cada columna i de A^{-1} y la renombramos como un vector de incógnitas x_i y cada columna i de la matriz identidad I como un vector de constantes b_i.

Con lo que la ecuación (16) queda como n sistemas de ecuaciones de n incógnitas:

$$\begin{aligned} A * x_1 &= b_1 \\ A * x_2 &= b_2 \\ &\vdots \\ A * x_n &= b_n \end{aligned}$$

Dichos sistemas los podemos resolver de manera eficiente factorizando A como L*U una sóla vez, para después resolver cada uno de los sistemas con el algoritmo 6 de Factorización Doolittle ya implementado.

Por lo que el pseudocódigo para su implementación queda de la siguiente manera:

Algorithm 7: SolveInverse(A)

Data: Matrix A
Result: Inverse matrix of A
Crea vector b_{nx1};
Crea vector x_{nx1};
Crea matriz $Ainv_{nxn}$;
factorizaDoolittle(A);
for *i desde 1 hasta n* **do**
    Inicializa b con columna i de matriz identidad;
    x = solveDoolittle(A, b);
    Añade x a columna i de Ainv;
end

1.8. Caso matriz A simétrica positiva definida mediante Cholesky modificado

En el caso del algoritmo de Cholesky modificado, el objetivo es realizar la factorización

$$A = LDL^T \tag{17}$$

Donde L es una matriz triangular inferiror de nxn donde los elementos de su diagonal son 1. Mientras que D es una matriz diagonal de nxn cuyas entradas son estrictamente mayores a cero.

Ambas matrices D y L se pueden encontrar mediante las siguientes ecuaciones:

$$d_{ii} = aii - \sum_{k=1}^{i-1} l_{ik}^2 * d_{kk} \tag{18}$$

$$l_{ji} = \frac{aji - \sum_{k=1}^{i-1} l_{jk} * l_{ik} * d_{kk}}{d_{ii}} \tag{19}$$

Una vez que se tienen las matrices D y L podemos proceder a resolver el sistema en (1) de la siguiente manera:

Tenemos que

$$L * D * L^T * x = b \tag{20}$$

Ahora podemos renombrar

$$L^T * x = y \tag{21}$$

Con lo que (20) se convierte en

$$L * D * y = b \tag{22}$$

Y a partir de (22) renombramos

$$D * y = z \tag{23}$$

Con lo que (22) se convierte en

$$L * z = b \tag{24}$$

Ahora resolvemos el sistema (24) obteniendo z, para que después mediante (23) podamos obtener y y finalmente mediante (21) podamos obtener x.

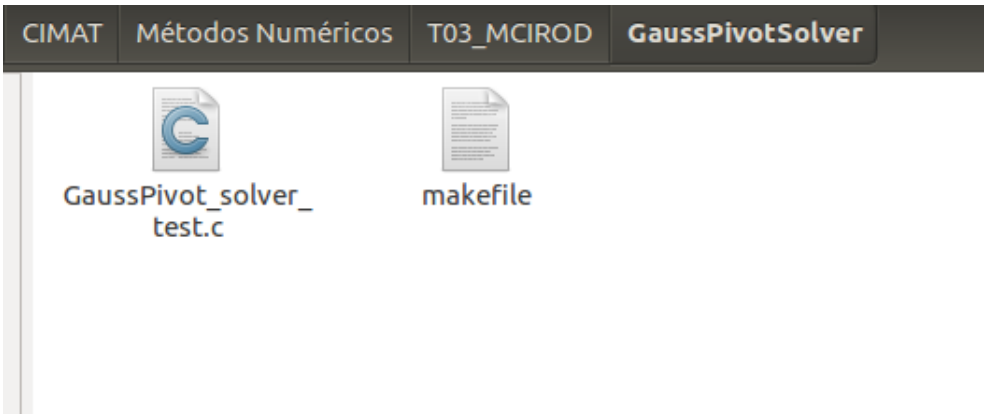
Por lo que el pseudocódigo para su implementación queda de la siguiente manera:

Algorithm 8: SolveModifiedCholesky(A, b)

```
Data: Matrix A, Vector b
Result: Vector solución x
Crea vector x_{nx1};
Crea vector y_{nx1};
Crea vector z_{nx1};
for i desde 1 hasta n do
    // D[i][i] = A[i][i] - Sum(k=1 to i-1)L[i][k]*L[i][k]*D[k][k];
    for k desde 1 hasta i do
        A[i][j] = A[i][j] - L[i][k]*L[i][k]*D[k][k];
    end
    for j desde i+1 hasta n do
        // L[j][i] = ( A[j][i] - Sum(k=1 to i-1)L[j][k]*L[i][k]*D[k][k] ) / D[i][i];
        A[j][i] = A[j][i]/A[i][i];
        for k desde 1 hasta i do
            A[j][i] = (A[j][i] - A[j][k]*A[i][k]*A[k][k])/A[i][i];
        end
        // Copia L[j][i] a su transpuesta L[i][j];
        L[i][j] = L[j][i];
    end
end
z = SolveLowerTriangular(L, b);
y = SolveDiagonal(D, z);
x = SolveUpperTriangular(L^T, y);
```

2. Ejemplo de prueba de la tarea entregada

2.1. Prueba Gauss con pivote



Una vez dentro de la carpeta de la prueba del método Gauss con pivote se compila el programa mediante el comando ”make”.

```
isaac@irb: ~/Documents/CIMAT/Métodos Numéricos/T03_MCIROD/GaussPivotSolver
isaac@irb ~/Documents/CIMAT/Métodos Numéricos/T03_MCIROD/GaussPivotSolver make
gcc -c GaussPivot_solver_test.c
gcc -c ../solve_matrix_direct.c
gcc -c  ../matrix_struct.c
gcc -o runTest GaussPivot_solver_test.o solve_matrix_direct.o matrix_struct.o
isaac@irb ~/Documents/CIMAT/Métodos Numéricos/T03_MCIROD/GaussPivotSolver
```

Una vez compilado, se puede ejecutar el programa pasando como argumento las rutas de las matrices A y b.

```
isaac@irb: ~/Documents/CIMAT/Métodos Numéricos/T03_MCIROD/GaussPivotSolver
✘ isaac@irb ~/Documents/CIMAT/Métodos Numéricos/T03_MCIROD/GaussPivotSolver ./runTest ../MATRICES/M_SMALL.txt ../MATRICES/V_SMALL.txt

Read matrix A
Succesfully read: ../MATRICES/M_SMALL.txt

 2.402822 4.425232 1.929374 1.370355
 1.201411 2.212616 0.964687 0.685178
 1.119958 0.964687 2.053172 0.566574
 0.742142 0.685178 0.566574 1.696828


Read vector b
Succesfully read: ../MATRICES/V_SMALL.txt

 0.060000
 0.542716
 0.857204
 0.761270

------------------------------------------------------------

Determinant of A: 0.000001

------------------------------------------------------------


Solution of system by Gauss with complete pivoting, x_solve:

 -5425479.813547
 1837966.776940
 1812933.651960
 1025432.000084


Comparing A*x_solve with b...

Test PASSED. A * x_solve = b for given A and b.

✘ isaac@irb ~/Documents/CIMAT/Métodos Numéricos/T03_MCIROD/GaussPivotSolver
```

Se muestra el cálculo de la solución al sistema de ecuaciones, así como el valor del determinante de la matriz A.

Una vez terminada la ejecución, se pueden borrar los archivos generados al compilar la prueba mediante el comando ”make clean”.

```
isaac@irb: ~/Documents/CIMAT/Métodos Numéricos/T03_MCIROD/GaussPivotSolver
✘ isaac@irb ~/Documents/CIMAT/Métodos Numéricos/T03_MCIROD/GaussPivotSolver make clean
rm -f runTest GaussPivot_solver_test.o solve_matrix_direct.o matrix_struct.o
isaac@irb ~/Documents/CIMAT/Métodos Numéricos/T03_MCIROD/GaussPivotSolver
```

3. Observaciones sobre los resultados

3.1. Caso matriz diagonal, triangular superior y triangular inferior

Para estos tres casos pude observar que la única condición que se debe cuidar a la hora de resolver el sistema de ecuaciones es que los elemntos de la diagonal principal no sean cero.

Tambien entendí lo importante que son estos métodos, principalmente los casos triangulares, ya que son la base para los demás algoritmos de resolución de sistemas de ecuaciones e incluso para el cálculo de la matriz inversa. Estos métodos también son conocidos como back y forward substitution.

Además de que el cálculo de los determinantes para estos casos también son triviales, dado que se pueden obtener como el producto de los elementos de la diagonal principal.

Finalmente, la complejidad de los métodos triangulares es de $O(n^2)$.

3.2. Caso Gauss sin pivoteo

Para el caso de Gauss sin implementar pivoteo, con el ejemplo dado de matriz M_SMALL.txt, se presentaron problemas ya que durante las iteraciones del algoritmo se presentaron elementos en la diagonal que se volvian cero por lo que no se podía proceder con la ejecución del programa. También observé que el primer y segundo renglón de la matriz ejemplo son múltiplos uno del otro a excepción de los elementos de la última columna, lo cual hace que los primeros $n-1$ elementos del segundo renglón se vuelvan cero y no se pueda continuar con la reducción.

También cabe destacar que incluso cuando los pivotes no son exactamente cero, pero próximos a cero, el algoritmo también puede ser numéricamente inestable ya que al dividir entre el pivote los elementos resultantes serán demasiado grandes haciendo que pueda haber errores de redondeo que se propaguen hasta la sustitución hacia atrás y que por lo tanto la solución x al sistema no sea la correcta.

Es por esto que no es recomendable realizar eliminación Gaussiana sin ningún tipo de pivoteo.

3.3. Caso Gauss con pivoteo

Para el caso de Gauss con pivoteo total, pude observar que es bastante estable numéricamente, ya que al escoger el elemento más grande en valor absoluto como pivote, podemos sortear los errores de redondeo y ceros que se puedan presentar durante la reducción de la matriz, haciendo que nuestra solución al sistema sea más precisa.

A pesar de esto, el realizar la búsqueda del elemento máximo de una submatriz en cada iteración, hace que la complejidad del algoritmo aumente. Es por esto que, en general, no hay muchas razones para preferir el uso de pivoteo total sobre el pivoteo parcial, que sólo realiza intercambio de renglones. A excepción de algunos casos muy específicos.

3.4. Caso factorización Doolittle

Para el caso de la factorización mediante Doolittle, se dió el mismo problema que en el algoritmo de Gauss sin pivoteo ya que aparecieron elementos iguala cero en la diagonal y no se pudo continuar con la factorización para el ejemplo dado. Si la matriz A cumple con ser simétrica positiva definida o diagonalmente dominante, no se presentarán este tipo de problemas.

Este tipo de algoritmos de factorización son bastante útiles ya que una vez obtenida, resolver un sistema de ecuaciones se reduce a llamar a los métodos de sustitución hacia adelante y hacia atrás (matrices triangulares inferiores y superiores respectivamente).

3.5. Caso factorización Cholesky modificado

Para el caso de factorización por Cholesky modificado, observé la importancia de verificar que la matriz que se trata de factoriar cumpla con ser simétrica, ya que de otra forma el resultado no tendrá sentido. A diferencia del método de Cholesky, no se requiere la condición de que los elementos calculados de L sean positivos para que no haya problema al calcular la raíz cuadrada.

Si la matriz A es positiva definida, se puede asegurar que existe dicha factorización y todos los elementos de D serán estrictamente mayores a cero. Esto hace que la factorización de Cholesky modifcado se pueda usar como prueba de una matriz es positiva definida.

Si A es positiva semidefinida, elementos de D pueden ser cero y la factorización no es única. Sin embargo, se puede encontrar una matriz de permutación P tal que PAP^T tenga una factorización de Cholesky modificado única.

3.6. Caso matriz inversa

Para el caso del cálculo de la matriz inversa, pude darme cuenta de la utilidad que pueden tener los algoritmos de factorización, ya que en este caso, el cálculo de la inversa, se implementó mediante la factorización LU de Doolittle.

Al usar el algoritmo de Doolittle, el problema que se podría presentar al tratar de encontrar la inversa, es que se encuente algún pivote igual a cero.

4. Posibles mejoras en los programas realizados

Para los primeros casos, como el de matriz diagonal, triangular superior y triangular inferior, no hay mayor problema en que se puedan resolver, mas que verificar que los elementos de la diagonal sean diferentes de cero.

4.1. Caso Gauss con pivoteo

Para el caso de Gauss con pivoteo, se podrían hacer mejoras o cambios en cuanto al algoritmo de pivoteo que se usa.

Ya que hacer pivoteo total no siempre es la mejor estrategia. De hecho, es mucho más común que se implemente el pivoteo parcial, ya que el costo de encontrar el elemento máximo de una matriz se sobrepone al beneficio en estabilidad numérica que da el pivoteo total. En este sentido, una mejora que se podría le podría hacer al pivoteo total, es realizar la búsqueda del elemento máximo de manera paralela, para cada una de las columnas o renglones, si es que se tiene la infraestructura para realizar esto.

Otro caso que se podría dar, es que los elementos de A sea de ordenes de magnitud muy diferentes, si se pudiera detectar esto, se podría usar el pivoteo escalado que toma en cuenta esta información.

4.2. Caso factorización Doolittle

Para el caso de la factorización de Doolittle, como se observó en los resultados, si no se implementa un algoritmo de pivoteo, estamos expuestos a encontrar elementos en la diagonal que resulten ser cero o muy cercanos a cero.

4.3. Caso factorización Cholesky modificado

Como se comentó en los resultados, Podríamos extender el algoritmo para que soporte la factorización de matrices positivas semidefinidas encontrando la matriz de permutación P.

En el caso que la matriz A no sea definida ni semidefinida, habría posibilidad de perturbar a la matriz A, sumándole alguna otra matriz E de forma que A se vuelva positiva definida pero que a la vez la matriz perturbada aún sea relevante a la aplicación original.

Por lo que, en general buscaríamos la siguiente igualdad:

$$P * (A + E) * P^T = L * D * L^T \tag{25}$$

Cuidando que se cumplan algunas condiciones para E y su relación con A.