TAREA 8 - METODOS NUMERICOS

Guillermo Segura Gómez
Centro de Investigación en Matemáticas
Métodos Numéricos
08 de octubre de 2023

1 Introducción

En el campo de la matemática, los métodos numéricos juegan un papel de suma importancia al brindar herramientas eficientes para resolver problemas matemáticos complejos que, en muchos casos, no tienen una solución analítica o son intratables por métodos tradicionales. Entre estos métodos, varios se dedican a resolver problemas relacionados con autovalores y autovectores, así como sistemas de ecuaciones lineales, que están directamente relacionados con la vida problemas que enfrentamos día con día. A continuación, se presenta una breve descripción de algunos de estos métodos:

Método de Iteración en Subespacio: Este método busca aproximaciones de un conjunto de autovalores (y sus correspondientes autovectores) de una matriz grande y densa. A través de iteraciones, se trabaja en un subespacio menor, significativamente reducido, en lugar de en el espacio original. Es especialmente útil para matrices de gran tamaño.

Método de Rayleigh: Es una técnica iterativa que utiliza el cociente de Rayleigh, una fórmula que proporciona una aproximación del autovalor de una matriz, para encontrar tanto autovalores como autovectores. Aunque es generalmente efectivo, su convergencia puede ser acelerada al usarse en combinación con otros métodos.

Método QR: Se basa en la factorización de matrices para encontrar autovalores. El proceso consiste en descomponer repetidamente una matriz en sus componentes Q (ortogonal) y R (triangular superior) y luego recombinarlas. La convergencia hacia una matriz diagonal o casi diagonal proporciona aproximaciones de los autovalores. Pag. 452 [1]

Método de Gradiente Conjugado: Es un método iterativo diseñado para resolver sistemas de ecuaciones lineales cuyas matrices son simétricas y definidas positivas. Este método es conocido por su eficiencia y rapidez en la convergencia, especialmente en matrices dispersas o sistemas de gran escala. Pag 354. [1]

Precondicionador de Jacobi: A menudo, el método de gradiente conjugado puede ser acelerado utilizando precondicionadores, que transforman el sistema original en uno equivalente que es más fácil de resolver. El precondicionador de Jacobi es uno de estos, y su principal ventaja radica en su simplicidad y en su capacidad para mejorar la convergencia del método de gradiente conjugado. Pag 360 [1]

2 Método de Rayleigh

2.1 Pseudocódigo

```
Funcion RayleighQuotientMethod(A: Matriz, v0: Vector, v1: Vector, n: Entero,
    epsilon: Decimal, maxIterations: Entero, eigenvalue: Referencia a Decimal)
    // Inicialización
    sigma <- 0
    B <- Matriz nxn
    Av0 <- Vector[n]

    // Iteración principal
    para k desde 0 hasta maxIterations - 1:
        // B = A - sigma * I
        para i desde 0 hasta n - 1:
            para j desde 0 hasta n - 1:
                B[i*n + j] <- A[i*n + j]
```

```
                if i == j:
                    B[i*n + i] <- B[i*n + i] - sigma

        // Resolución del sistema
        eliminacionGaussiana(n, B, v0)
        resuelveSistema(n, B, v0, v1)

        // Normalización de v1
        norm_v1 <- sqrt(DotProd(v1, v1, n))
        Divide(v1, norm_v1, v1, n)

        // Actualización de v0 y cálculo de sigma
        para i desde 0 hasta n - 1:
            v0[i] <- v1[i]
        MatrixProduct(A, v0, Av0, n, n, 1)
        sigma <- DotProd(v0, Av0, n) / DotProd(v0, v0, n)

        // Verificación de convergencia
        SubtractVector(v1, v0, n)
        if sqrt(DotProd(v1, v1, n)) < epsilon:
            salir del bucle

    // Actualizar el eigenvalor
    eigenvalue <- sigma

    // Liberación de memoria
    liberar(B)
    liberar(Av0)
Fin Funcion
```

Listing 1: Método de cociente de Rayleigh

2.2 Código

En el siguiente código se implementa la rutina del método de Rayleigh en C

```c
void RayleighQuotientMethod(double *A, double *v0, double *v1, int n, double
    epsilon, int maxIterations, double *eigenvalue) {

    // Inicialización de la variable sigma
    double sigma = 0;

    // Alocación dinámica de memoria para matrices temporales
    double *B = (double *)malloc(n * n * sizeof(double));
    double *Av0 = (double *)malloc(n * sizeof(double));

    // Iteración principal
    for (int k = 0; k < maxIterations; k++) {

        // B = A - sigma * I
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                B[i*n + j] = A[i*n + j];
                if (i == j) {
                    B[i*n + i] -= sigma;
                }
```

```
            }
        }

        // Resolución del sistema usando eliminación Gaussiana
        eliminacionGaussiana(n, B, v0);
        resuelveSistema(n, B, v0, v1);

        // Normalización de v1
        double norm_v1 = sqrt(DotProd(v1, v1, n));
        Divide(v1, norm_v1, v1, n);

        // Actualización de v0 y cálculo de sigma
        for (int i = 0; i < n; i++) {
            v0[i] = v1[i];
        }
        MatrixProduct(A, v0, Av0, n, n, 1);
        sigma = DotProd(v0, Av0, n) / DotProd(v0, v0, n);

        // Verificación de convergencia
        SubtractVector(v1, v0, n);
        if (sqrt(DotProd(v1, v1, n)) < epsilon) {
            break;
        }
    }

    // Actualización del eigenvalor resultante
    *eigenvalue = sigma;

    // Liberación de memoria
    free(B);
    free(Av0);
}
```

Listing 2: Código del método de Rayleigh

Ahora se muestra la función main para el método de Rayleigh y su ejecución

```
# include <stdio.h>
# include <stdlib.h>
# include <math.h>
# include "matrix.h"

int main(int argc, char *argv[]){

    // Verificación de argumentos de entrada
    if(argc != 2) {
        printf("Uso: %s <nombre_archivo_matriz>\n", argv[0]);
        return 1;
    }

    const char* filename = argv[1];
    double *A;
    int rows, cols;

    // Lectura de la matriz desde el archivo
    if (ReadMatrix(filename, &A, &rows, &cols) == 1) {
        free(A);
        return 0;
```

```
    }

    // Verificación de que la matriz sea cuadrada
    if(rows != cols) {
        printf("La matriz no es cuadrada.\n");
        free(A);
        return 1;
    }

    // Inicialización del vector de entrada v0
    double v0[rows];
    for(int i = 0; i < rows; i++) {
        v0[i] = 1.0;
    }

    double v1[rows];
    double eigenvalue;

    // Aplicación del método cociente de Rayleigh
    RayleighQuotientMethod(A, v0, v1, rows, 1e-9, 1000, &eigenvalue);

    // Impresión de resultados
    printf("Eigenvalor aproximado: %lf\n", eigenvalue);
    printf("El vector propio aproximado: \n");
    MatrixShow(rows, 1, v0);

    // Liberación de memoria
    free(A);
    return 0;
}
```

Listing 3: Main del método de Rayleigh

```
guillermo_sego@MacBook-Air Tarea08 % ./build/Rayleigh Eigen.txt
Eigenvalor aproximado: 4.703727
El vector propio aproximado:
0.754630
0.536811
0.377315
```

Listing 4: Ejecución del método de Rayleigh

3 Método de QR

3.1 Pseudocódigo

```
Funcion QR_Factorization(A: Matriz, Q: Matriz, R: Matriz, n: Entero)
    u, e <- Matrices nxn

    para i desde 0 hasta n - 1:
        // Copiar columna de A en u[i]
        para j desde 0 hasta n - 1:
            u[i][j] <- A[j + i*n]

        para k desde 0 hasta i - 1:
```

```
            proj <- Vector[n]
            // Proyección de u[i] en e[k]
            para j desde 0 hasta n - 1:
                proj[j] <- e[k][j] * DotProd(u[i], e[k], n)
            u[i] <- u[i] - proj

        // Normalización para obtener e[i]
        norm <- sqrt(DotProd(u[i], u[i], n))
        para j desde 0 hasta n - 1:
            e[i][j] <- u[i][j] / norm

    // Construcción de las matrices Q y R
    para i desde 0 hasta n - 1:
        para j desde 0 hasta n - 1:
            Q[j + i*n] <- e[i][j]
            if i <= j:
                R[i + j*n] <- DotProd(e[i], u[j], n)
            else:
                R[i + j*n] <- 0.0
Fin Funcion
```

Listing 5: Factorización QR

3.2 Código

En el siguiente código se implementa la rutina del método de QR en C

```c
// Función para realizar la factorización QR
void QR_Factorization(double *A, double *Q, double *R, int n) {
    // Creamos matrices temporales
    double u[n][n];
    double e[n][n];

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            u[i][j] = A[j + i*n];
        }

        for (int k = 0; k < i; k++) {
            double proj[n];
            for (int j = 0; j < n; j++) {
                proj[j] = e[k][j] * DotProd(u[i], e[k], n);
            }
            SubtractVector(u[i], proj, n);
        }

        // Normalizamos el vector u[i] para obtener e[i]
        double norm = sqrt(DotProd(u[i], u[i], n));
        for (int j = 0; j < n; j++) {
            e[i][j] = u[i][j] / norm;
        }
    }

    // Obtenemos Q y R
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
```

```
            Q[j + i*n] = e[i][j];
            if (i <= j) {
                R[i + j*n] = DotProd(e[i], u[j], n);
            } else {
                R[i + j*n] = 0.0;
            }
        }
    }
}
```

Listing 6: Código del método de QR

Ahora se muestra la función main para el método de QR y su ejecución

```
# include <stdio.h>
# include <stdlib.h>
# include <math.h>
# include "matrix.h"

int main(int argc, char *argv[]){

    // Manejo de argumentos de la línea de comandos
    if(argc != 2) {
        printf("Uso: %s <nombre_archivo_matriz>\n", argv[0]);
        return 1;
    }

    const char* filename = argv[1];

    // No tenemos certeza del size de la matriz
    // Declaramos un apuntador hacia un bloque de memoria específico de la
    matriz A
    double *A;
    int rows, cols;

    // Leemos la matriz en el archivo, pasamos por referencia el valor del
    apuntador
    if (ReadMatrix(filename ,&A, &rows, &cols) == 1) {
        free(A); // Liberamos memoria
        return 0;
    }

    // Comprobar si la matriz es cuadrada
    if (rows != cols) {
        printf("La matriz debe ser cuadrada para la factorización QR.\n");
        free(A);
        return 1;
    }

    double *Q = (double *)malloc(rows * rows * sizeof(double));
    double *R = (double *)malloc(rows * rows * sizeof(double));

    QR_Factorization(A, Q, R, rows);

    printf("Matriz Q:\n");
    MatrixShow(rows, rows, Q);
```

```
    printf("\nMatriz R:\n");
    MatrixShow(rows, rows, R);

    free(Q);
    free(R);

    return 0;
}
```

Listing 7: Main del método de QR

```
guillermo_sego@MacBook-Air Tarea08 % ./build/QR Eigen.txt
Matriz Q:
0.942201        -0.335047       0.000000
0.326839        0.919119        -0.219992
0.073708        0.207277        0.975502

Matriz R:
5.306721        0.000000        0.000000
0.000000        8.082098        0.000000
0.000000        0.000000        9.386477
```

Listing 8: Ejecución del método de QR

4 Método de Gradiente Conjugado

4.1 Pseudocódigo

```
Funcion Conjugate_gradient(A: Matriz, B: Vector, x: Vector, rows: Entero,
    cols: Entero)
    error_threshold <- 0.0001
    r, r_next, p, Ap, Ax <- Vector[rows]

    // Calcular residuo inicial r_0 = B - A*x_0
    Ax <- A * x
    para i desde 0 hasta rows - 1:
        r[i] <- B[i] - Ax[i]
    p <- r  // Inicializamos p_0 = r_0

    max_iterations <- rows
    para k desde 0 hasta max_iterations - 1:
        Ap <- A * p
        r_dot <- r . r
        alpha <- r_dot / (p . Ap)

        // Actualizar solucion y calcular el nuevo residuo
        para i desde 0 hasta rows - 1:
            x[i] <- x[i] + alpha * p[i]
            r_next[i] <- r[i] - alpha * Ap[i]

        // Verificar la convergencia
        r_next_dot <- r_next . r_next
        if sqrt(r_next_dot) < error_threshold:
            break
```

```
        beta <- r_next_dot / r_dot
        para i desde 0 hasta rows - 1:
            p[i] <- r_next[i] + beta * p[i]
            r[i] <- r_next[i]

    // Liberar memoria
    liberar(r, r_next, p, Ap, Ax)
Fin Funcion
```

Listing 9: Método de Gradiente Conjugado

4.2 Código

En el siguiente código se implementa la rutina del método de Gradiente Conjugado en C

```c
void Conjugate_gradient(double *A, double *B, double *x, int rows, int cols){
    double error_threshold = 0.0001;
    double alpha, beta, r_dot, r_next_dot;

    double *r = (double *)malloc(rows * sizeof(double));
    double *r_next = (double *)malloc(rows * sizeof(double));
    double *p = (double *)malloc(rows * sizeof(double));
    double *Ap = (double *)malloc(rows * sizeof(double));

    // r_0 = B - A*x_0
    double *Ax = (double *)malloc(rows * sizeof(double));
    MatrixProduct(A, x, Ax, rows, cols, 1);
    for(int i = 0; i < rows; i++) {
        r[i] = B[i] - Ax[i];
    }

    for(int i = 0; i < rows; i++){
        p[i] = r[i];  // p_0 = r_0
    }

    int max_iterations = rows;
    for(int k = 0; k < max_iterations; k++){

        MatrixProduct(A, p, Ap, rows, cols, 1);

        r_dot = DotProd(r, r, rows);
        alpha = r_dot / DotProd(p, Ap, rows);

        for(int i = 0; i < rows; i++){
            x[i] += alpha * p[i];
            r_next[i] = r[i] - alpha * Ap[i];
        }

        r_next_dot = DotProd(r_next, r_next, rows);
        if(sqrt(r_next_dot) < error_threshold){
            break;
        }

        beta = r_next_dot / r_dot;
        for(int i = 0; i < rows; i++){
            p[i] = r_next[i] + beta * p[i];
```

```
            r[i] = r_next[i];
        }
    }

    free(r);
    free(r_next);
    free(p);
    free(Ap);
    free(Ax);
}
```

Listing 10: Código del método de Gradiente Conjugado

Ahora se muestra la función main para el método de Gradiente Conjugado y su ejecución

```
# include <stdio.h>
# include <stdlib.h>
# include <math.h>
# include "matrix.h"

int main(int argc, char *argv[]){

    // Manejo de argumentos de la línea de comandos
    if(argc != 3) {
        printf("Uso: %s <nombre_archivo_matriz>\n", argv[0]);
        return 1;
    }

    const char* filename1 = argv[1];
    const char* filename2 = argv[2];

    // Declaramos un apuntador hacia un bloque de memoria específico de la
    matriz A y B
    double *A, *B;
    int rows, cols, r, c;

    // Leemos la matriz A en el archivo, pasamos por referencia el valor del
    apuntador
    if (ReadMatrix(filename1 ,&A, &rows, &cols) == 1) {
        free(A); // Liberamos memoria
        return 0;
    }

    // Leemos la matriz B en el archivo, pasamos por referencia el valor del
    apuntador
    if (ReadMatrix(filename2 ,&B, &r, &c) == 1) {
        free(B); // Liberamos memoria
        return 0;
    }

    // Declaramos la memoria para la solución
    double *X = malloc( rows * sizeof(double) );
    Initialize(X, rows);

    // Calculamos el gradiente conjugado
    Conjugate_gradient( A, B, X, rows, cols );
```

```
    // Mostramos la matriz
    printf("La solución del sistema X:\n");
    MatrixShow(rows, 1, X);


    return 0;
}
```

Listing 11: Main del método de Gradiente Conjugado

```
guillermo_sego@MacBook-Air Tarea08 % ./build/Conjugate_gradient Matrix.txt
    Vector.txt
La solución del sistema X:
3.000000
4.000000
-5.000000
```

Listing 12: Ejecución del método de Gradiente Conjugado

5 Método Precondicionador de Jacobi

5.1 Pseudocódigo

```
Funcion Conjugate_gradient_preconditioned(A: Matriz, B: Vector, x: Vector,
    rows: Entero, cols: Entero)
    error_threshold <- 0.0001
    r, z, p, Ap, Ax <- Vector[rows]

    // Calcular residuo inicial r_0 = B - A*x_0
    Ax <- A * x
    para i desde 0 hasta rows - 1:
        r[i] <- B[i] - Ax[i]

    // Aplicar precondicionador de Jacobi: z_0 = D^-1 * r_0
    para i desde 0 hasta rows - 1:
        z[i] <- r[i] / A[i*cols + i]
    p <- z

    max_iterations <- rows
    para k desde 0 hasta max_iterations - 1:
        Ap <- A * p
        r_dot <- r . z
        alpha <- r_dot / (p . Ap)

        // Actualizar solucion y residuo
        para i desde 0 hasta rows - 1:
            x[i] <- x[i] + alpha * p[i]
            r[i] <- r[i] - alpha * Ap[i]

        // Aplicar de nuevo el precondicionador de Jacobi
        para i desde 0 hasta rows - 1:
            z[i] <- r[i] / A[i*cols + i]

        r_next_dot <- r . z
        if sqrt(r_next_dot) < error_threshold:
```

```
            break

        beta <- r_next_dot / r_dot
        para i desde 0 hasta rows - 1:
            p[i] <- z[i] + beta * p[i]

    // Liberar memoria
    liberar(r, z, p, Ap, Ax)
Fin Funcion
```

Listing 13: Método de Gradiente Conjugado con Precondicionamiento

5.2 Código

En el siguiente código se implementa la rutina del método Precondicionador de Jacobi en C

Listing 14: Código del método Precondicionador de Jacobi

Ahora se muestra la función main para el método Precondicionador de Jacobi y su ejecución

```
// Método de gradiente conjugado con precondicionamiento
void Conjugate_gradient_preconditioned(double *A, double *B, double *x, int
    rows, int cols){

    // Definimos el umbral de error
    double error_threshold = 0.0001;
    double alpha, beta, r_dot, r_next_dot;

    // Reserva de memoria para vectores temporales
    double *r = (double *)malloc(rows * sizeof(double));  // Residuo
    double *z = (double *)malloc(rows * sizeof(double));  // z es el residuo
    precondicionado
    double *p = (double *)malloc(rows * sizeof(double));
    double *Ap = (double *)malloc(rows * sizeof(double));

    // Residuo inicial r_0 = B - A*x_0
    double *Ax = (double *)malloc(rows * sizeof(double));
    MatrixProduct(A, x, Ax, rows, cols, 1);
    for(int i = 0; i < rows; i++) {
        r[i] = B[i] - Ax[i];
    }

    // Aplicamos el precondicionador de Jacobi: z_0 = D^-1 * r_0
    for(int i = 0; i < rows; i++) {
        z[i] = r[i] / A[i*cols + i];
    }
    for(int i = 0; i < rows; i++){
        p[i] = z[i];  // p_0 = z_0
    }

    // Establecemos el número máximo de iteraciones
    int max_iterations = rows;
    for(int k = 0; k < max_iterations; k++){

        // Calculamos Ap
```

```
        MatrixProduct(A, p, Ap, rows, cols, 1);
        r_dot = DotProd(r, z, rows);

        // Calculamos alpha
        alpha = r_dot / DotProd(p, Ap, rows);

        // Actualizamos la solución y el residuo
        for(int i = 0; i < rows; i++){
            x[i] += alpha * p[i];
            r[i] -= alpha * Ap[i];
        }

        // Aplicamos nuevamente el precondicionador de Jacobi
        for(int i = 0; i < rows; i++) {
            z[i] = r[i] / A[i*cols + i];
        }

        // Calculamos el producto punto para el siguiente residuo y
    verificamos la convergencia
        r_next_dot = DotProd(r, z, rows);
        if(sqrt(r_next_dot) < error_threshold){
            break;
        }

        // Calculamos beta y actualizamos p
        beta = r_next_dot / r_dot;
        for(int i = 0; i < rows; i++){
            p[i] = z[i] + beta * p[i];
        }
    }

    // Liberamos la memoria de los vectores temporales
    free(r);
    free(z);
    free(p);
    free(Ap);
    free(Ax);
}
```

Listing 15: Main del método Precondicionador de Jacobi

```
# include <stdio.h>
# include <stdlib.h>
# include <math.h>
# include "matrix.h"

int main(int argc, char *argv[]){

    // Manejo de argumentos de la línea de comandos
    if(argc != 3) {
        printf("Uso: %s <nombre_archivo_matriz>\n", argv[0]);
        return 1;
    }

    const char* filename1 = argv[1];
    const char* filename2 = argv[2];
```

```
    // Declaramos un apuntador hacia un bloque de memoria específico de la
    matriz A y B
    double *A, *B;
    int rows, cols, r, c;

    // Leemos la matriz A en el archivo, pasamos por referencia el valor del
    apuntador
    if (ReadMatrix(filename1 ,&A, &rows, &cols) == 1) {
        free(A); // Liberamos memoria
        return 0;
    }

    // Leemos la matriz B en el archivo, pasamos por referencia el valor del
    apuntador
    if (ReadMatrix(filename2 ,&B, &r, &c) == 1) {
        free(B); // Liberamos memoria
        return 0;
    }

    // Declaramos la memoria para la solución
    double *X = malloc( rows * sizeof(double) );
    Initialize(X, rows);

    // Calculamos el gradiente conjugado
    Conjugate_gradient_preconditioned( A, B, X, rows, cols );

    // Mostramos la matriz
    printf("La solución del sistema X:\n");
    MatrixShow(rows, 1, X);


    return 0;
}
```

Listing 16: Ejecución del método Precondicionador de Jacobi

```
guillermo_sego@MacBook-Air Tarea08 % ./build/Jacobi_pre Matrix.txt Vector.txt
La solución del sistema X:
3.000000
4.000000
-5.000000
```

Listing 17: Ejecución del método Precondicionador de Jacobi

6 Conclusión

En los métoodos que se implementaron, el Método de Iteración en Subespacio, el Método de Rayleigh, el Método QR y el Gradiente Conjugado, tenemos avances significativos en nuestra capacidad de solución de nuevos problemas que antes debido a la falta de conocimiento de estas técnicas no podíamos haber realizado en soluciones particulares. En especial, el uso de precondicionadores, como el de Jacobi, demuestra cómo los refinamientos y adaptaciones en estos métodos pueden llevar a mejoras sustanciales en términos de eficiencia y velocidad de convergencia. En un mundo donde los sistemas y modelos se vuelven cada

vez más complejos, estos métodos no solo son útiles, sino esenciales. Su aplicación práctica en áreas como la física, la ingeniería y la economía tiene su relevancia y su impacto potencial en la formulación y solución de desafíos del mundo real. Con el continuo avance de la tecnología y la computación, es probable que veamos más evoluciones y adaptaciones de estos métodos, fortaleciendo aún más el puente entre la teoría matemática y sus aplicaciones prácticas. Además esto complementó en gran medida los avances que ya se tenián en cuanto a otros métodos, por ejempolo los métodos de subespacio y Rayleigh proporcionaron manera adicionales para el cálculo de autovalores, mientras que el método de QR nos dió un nuevo tipo de factorización de matrices. Finalmente los métodos de gradiente conjugado proporcionaron una alternativa para resolver sistemas de ecuaciones lineales.

NOTA: Falta agregar el método de sub espacio. Me encuentro tabajand en el ya que tuve problemas al realizar los métodos de la potencia inversa y de Jacobi.

References

[1] R. L. Burden, J. D. Faires, and A. M. Burden, *Numerical analysis*. Cengage learning, 2015.