EXAMEN 1 - METODOS NUMERICOS

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

1 Problema 1

Escribe un programa en C para calcular la constante matemática e, considerando la definición

$$e = lim_{e\to\infty}\left(1+\frac{1}{n}\right)^n = 2,7182818..... \tag{1}$$

- Calcula $(1+1/n)^n$ para $n = 10^k$, $k = 1, 2, ..., 20$. (1 punto)
- Determina el error relativo y absoluto de las aproximaciones comparándolas con $exp(1)$. Proporciona una explicación detallada de lo sucedido. (2 punto)

El programa en C que calcula el primer punto se muestra a continuación. Los valores del exponencial calculados son guardados en un arreglo y posteriormente se calcula el error. El conjunto de resultados se imprime en una tabla para mejor visualización.

```c
# include <stdio.h>
# include <math.h>

// Funcion para calcular el numero e
double exponencial( int k ){

    // Calcula n
    int n = pow(10, k);

    // Declaramos el valor donde guardaremos e
    double e;

    // Calculamos e por la formula del limite
    e = pow(1.0 + 1.0 / (double)n, n);

    return e;
}

int main(){

    int k;
    double exp_calculado, valor, error_absoluto, error_relativo;
    double exp_calculados[20];  // Almacena los valores calculados de e para
    diferentes k

    valor = exp(1.0);  // El valor verdadero de e


    // Utilizamos el contador del for como el valor de k para llamar la funci
    ón y aproximar e
    for (int k = 1; k <= 20; k++) {
        exp_calculados[k - 1] = exponencial(k); // Calcula e para k = 1, 2,
    ..., 20 y almacena en exp_calculados
    }


    // Imprime la tabla
    printf("
    ----------------------------------------------------------------------\n");
    printf("|    k    |     e calculado     |  Error absoluto | Error relativo
     |\n");
```

```
    printf("
    ----------------------------------------------------------------------\n");

    // Calculamos los errores en un ciclo for y los vamos imprimiendo para no
     tener que almacenarlos
    for (int k = 1; k <= 20; k++) {
        // Calculo del error absoluto
        error_absoluto = fabs(exp_calculados[k - 1] - valor);

        // Calculo del error relativo
        error_relativo = error_absoluto / valor;

        printf("| %7d | %17.15lf | %15.15lf | %14.15lf |\n", k,
    exp_calculados[k - 1], error_absoluto, error_relativo);
    }

    printf("
    ----------------------------------------------------------------------\n");

   return 0;
}
```

Listing 1: Programa para calcular e

Se muestra la terminal y la manera de ejecutar el programa desde la terminal.

```
guillermo_sego@MacBook-Air Examen1 % gcc Problema1.1.c -o Problema1.1
guillermo_sego@MacBook-Air Examen1 % ./Problema1.1
----------------------------------------------------------------------
|    k    |     e calculado    |  Error absoluto | Error relativo |
----------------------------------------------------------------------
|       1 | 2.593742460100002 | 0.124539368359043 | 0.045815473235769 |
|       2 | 2.704813829421528 | 0.013467999037517 | 0.004954599959619 |
|       3 | 2.716923932235594 | 0.001357896223452 | 0.000499542103852 |
|       4 | 2.718145926824926 | 0.000135901634120 | 0.000049995417214 |
|       5 | 2.718268237192297 | 0.000013591266748 | 0.000004999947616 |
|       6 | 2.718280469095753 | 0.000001359363292 | 0.000000500081808 |
|       7 | 2.718281694132082 | 0.000000134326963 | 0.000000049416128 |
|       8 | 2.718281798347358 | 0.000000030111687 | 0.000000011077471 |
|       9 | 2.718282052011560 | 0.000000223552515 | 0.000000082240374 |
|      10 | 2.718281826560347 | 0.000000001898698 | 0.000000000698492 |
|      11 | 2.718281826560347 | 0.000000001898698 | 0.000000000698492 |
|      12 | 2.718281826560347 | 0.000000001898698 | 0.000000000698492 |
|      13 | 2.718281826560347 | 0.000000001898698 | 0.000000000698492 |
|      14 | 2.718281826560347 | 0.000000001898698 | 0.000000000698492 |
|      15 | 2.718281826560347 | 0.000000001898698 | 0.000000000698492 |
|      16 | 2.718281826560347 | 0.000000001898698 | 0.000000000698492 |
|      17 | 2.718281826560347 | 0.000000001898698 | 0.000000000698492 |
|      18 | 2.718281826560347 | 0.000000001898698 | 0.000000000698492 |
|      19 | 2.718281826560347 | 0.000000001898698 | 0.000000000698492 |
|      20 | 2.718281826560347 | 0.000000001898698 | 0.000000000698492 |
----------------------------------------------------------------------
```

Listing 2: Terminal de ejecucion

En resultados mostrados en la tabla impresa por la terminal de ejecución, se observa un error mayor en los valores de k mas bajos. A partir del valor de k = 10, parece que los valores del error convergen y después de k = 10 no tenemos cambios en el cálculo, ni en el

error absoluto ni relativo.

Esto puede ser atribuido a la precisión computacional del tipo de variables que se utilizan durante el programa. Utilizamos variables tipo **double**, la cual tiene una precisión de 8 bytes es decir 64 bits. La precisión que podemos alcanzar es el número máximo de dígitos que el tipo de dato puede representar de manera precisa. En la medida que incrementamos el valor de n, la contribución de $1/n$ al cálculo se vuelve mas pequeña y el cálculo final alcanza la máxima precisión que la variable tipo **duuble** puede ofrecer.

2 Problema 2

Se requiere resolver un problema de difusión del cual se obtiene un sistema de ecuaciones pentadiagonal (5 diagonales diferentes de cero y los demás términos de la matriz son cero). El sistema de ecuaciones tiene la siguiente forma:

$$\mathbf{Ax} = \mathbf{b} \tag{2}$$

con $-4x_{i-2} - 8x_{i-1} + 40x_i - 8x_{i+1} - 4x_{i+2} = 100, \quad$ para $i = 3, \ldots, 1998$.

En los extremos de la matriz A se eliminan los términos que quedan fuera (cuando el índice es negativo o mayor a 2000). Así,

- - La primer ecuación es: $40x_1 - 8x_2 - 4x_3 = 20$;
- - La segunda ecuación es: $-8x_1 + 40x_2 - 8x_3 - 4x_4 = 50$,
- - La penúltima ecuación es: $-4x_{1997} - 8x_{1998} + 40x_{1999} - 8x_{2000} = 50$,
- - La última ecuación es: $-4x_{1998} - 8x_{1999} + 40x_{2000} = 20$.

Con el sistema generado (1 pt.), obtener:

- - La solución del sistema de ecuaciones con dos métodos distintos. Decir por qué utilizaste esos métodos de solución. (3 pts.)
- - Los eigenvectores correspondientes a la matriz A. Presentar los 10 eigenvalores más pequeños y los 10 más grandes. (3 pts.)

2.1 Método del gradiente conjugado

Para la solución del sistema de ecuaciones utilizamos primero el método del gradiente conjugado. El método del gradiente conjugado es un método excelente para solución de este tipo de sistemas $Ax = b$ donde A es una matriz simétrica y definida positiva. Este método es ideal si se trata de matrices dispersas ya que las matrices dispersas al tener una gran cantidad de ceros permiten ahorro de operaciones matriciales. Además el método de gradiente positivo tiene la ventaja de que no necesita almacenar matrices adicionales, te da la solución aproximada utilizando como entrada las matrices A y B. A continuación se muestra el código del gradiente conjugado implementado para resolver el sistema:

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

```c
# include <math.h>

int main() {

    int size = 2000;
    // Inicializar la matriz A como un solo bloque de memoria
    double *A = malloc(size * size * sizeof(double));

    // Inicializar el vector B
    double *B = malloc(size * sizeof(double));

    // Iniclaizar la matriz
    Matrix_Initialize(A, B, size);

    /*

    Metodo de Gradiente conjugado

    */

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

    // Calcular el gradiente conjugado
    Conjugate_gradient(A, B, X, size, size);

    // Checar si tenemos solución
    double tolerance = 1e-3;  // Define umbral de tolerancia
    int control = isSolution(A, X, B, size, tolerance);

    if (control == 0 ) {
        printf("X es solución de AX = b.\n");
    } else {
        printf("X no es solución de AX = b.\n");
    }

    // Guardar valores propios en un archivo
    saveMatrixToFile( X, size, 1, "Solucion_Gradiente_Conjugado.txt" );

    // Liberar la memoria
    free(A);
    free(B);
    free(X);

    return 0;
}
```

Listing 3: Main para calcular el gradiente conjugado

Se indica a continuación la ejecución del método de gradiente conjugado. Para poder ejecutar el programa se utiliza un archivo tipo **makefile**. Se ejecuta en la terminal y el archivo tiene la instrucción de relacionar las bibliotecas y compilar el archivo. Utilizamos la librería de autoría propia llamada **matrix.h**, en ella se guardan las rutinas necesarias, por ejemplo la rutina **Matrix_Initialize**, inicializa la matriz como se indicó en el problema. La compilación del código se indica a continuación.

```
guillermo_sego@MacBook-Air Examen1 % make
gcc -g -o build/Debug/Problema2.1_GC.o -c Problema2.1_GC.c
gcc -g -o build/Problema2.1_GC build/Debug/matrix.o build/Debug/Problema2.1
    _GC.o
guillermo_sego@MacBook-Air Examen1 % ./build/Problema2.1_GC
X es solución de AX = b.
```

Listing 4: Terminal de ejecucion del metodo de gradiente conjugado

2.2 Método de Cholesky

El segundo método que se utilizó para resolver el sistema fue el método Cholesky. La descomposición de Cholesky es especialmente útil cuando se trata con matrices que son simétricas y definidas positivas, al igual que el código anterior. De manera similar al método del gradiente conjugado, el método de Cholesky es útil especialmente en matrices que son dispersas, pues la cantidad de ceros en estas matrices permite optimizar y acelerar los cálculos. Además, debido a la naturaleza de la descomposición, el guardado de datos es eficiente ya que sólo es necesario almacenar la matriz triangular inferior L. Por estas razones se utilizó el método de Choelsky para resolver el sistema. A continuación se presenta el código main de la solución del método de Cholesky

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

int main() {

    int size = 2000;
    // Inicializar la matriz A como un solo bloque de memoria
    double *A = malloc(size * size * sizeof(double));

    // Inicializar el vector B
    double *B = malloc(size * sizeof(double));

    // Iniclaizar la matriz
    Matrix_Initialize(A, B, size);

    /*

    Metodo de Cholesky

    */

    // Declaramos las matrices L y U para almacenar los resultados
    double *L = (double *) malloc(size * size * sizeof(double));
    // Las incializamos en 0
    Initialize(L, size * size);

    // Aplicamos la factorización LU
    // Manejo de estado devuelto por crout
    if (cholesky(A, L, size) == -1) {
        printf("Error durante la factorización\n");
        free(A);
        free(B);
```

```
        free(L);
        return -1;
    }

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

    solveCholesky(L, B, X, size);

    // Checar si tenemos solución
    double tolerance = 1e-3;  // Define umbral de tolerancia
    int control = isSolution(A, X, B, size, tolerance);

    if (control == 0 ) {
        printf("X es solución de AX = b.\n");
    } else {
        printf("X no es solución de AX = b.\n");
    }

    // Guardar valores propios en un archivo
    saveMatrixToFile( X, size, 1, "Solucion_Cholesky.txt" );

    // Liberar la memoria
    free(A);
    free(B);
    free(L);
    free(X);

    return 0;
}
```

Listing 5: Main para ejecutar el método de Cholesky

Al igual que el método de gradiente conjugado. Guardamos las funciones en la misma biblioteca **matrix.h**. Para poder compilar el programa se modifica el archivo make para relacionar los ejecutable con las librerías.

```
guillermo_sego@MacBook-Air Examen1 % make
gcc -g -o build/Debug/Problema2.1_C.o -c Problema2.1_C.c
gcc -g -o build/Problema2.1_GC build/Debug/matrix.o build/Debug/Problema2.1_C
    .o
guillermo_sego@MacBook-Air Examen1 % ./build/Problema2.1_C
X es solución de AX = b.
```

Listing 6: Terminal de ejecucion del metodo de Cholesky

2.3 Método de la potencia

Para poder calcular los eigenvalores de la matriz se utilizó el método de potencias. El método de potencias permite calcular los primeros n eigenvalores de una matriz. Este método tiene la particularidad de regresar los valores propios ordenados del mayor al menos. Para el cálculo de los 10 eigenvalores mas grandes y los 10 eigenvalores mas pequeños utilizamos el mismo método de potencias. Esto se logra calculando todos los eigenvalores de la matriz. Los primeros 10 valores propios serán los mayores, y los últimos serán los menores.

El siguiente código es el main para el método de potencias

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

int main() {
    int size = 200;
    // Inicializar la matriz A como un solo bloque de memoria
    double *A = malloc(size * size * sizeof(double));

    // Inicializar el vector B
    double *B = malloc(size * sizeof(double));

    // Iniclaizar la matriz
    Matrix_Initialize(A, B, size);

    /*

    Metodo de potencias

    */

    // Reservar memoria para almacenar los vectores propios y valores propios
    double *U = (double *)malloc(size * size * sizeof(double));
    double *l = (double *)malloc(size * sizeof(double));

    // Ejecutar el método de la potencia
    PowerMethod(A, U, l, size);

    // Imprimir valores propios
    int numValues = 10;  // Número de valores propios a mostrar
    printf("+---------------------------------------+\n");
    printf("| Valores propios mayores | Valores propios menores |\n");
    printf("+---------------------------------------+\n");

    for(int i = 0; i < numValues; i++) {
        printf("| %7.2f                 | %7.2f                 |\n", l[i],
     l[size-numValues+i]);
    }
    printf("+---------------------------------------+\n");

    // Guardar valores propios en un archivo
    saveMatrixToFile( l, size, 1, "Valores_propios.txt" );

    // Liberar la memoria
    free(A);
    free(U);
    free(l);
    free(B);
    return 0;
}
```

Listing 7: Main para ejecutar el método de Potencia

Para compilar el programa de igual forma utilizamos un archivo **makefile** que relaciona la librería **matrix** que es donde se encuentran muchas rutinas necesarias para la ejecución

del programa.

```
guillermo_sego@MacBook-Air Examen1 % make
gcc -g -o build/Debug/Problema2.1_PM.o -c Problema2.1_PM.c
gcc -g -o build/Problema2.1_PM build/Debug/matrix.o build/Debug/Problema2.1
    _PM.o
guillermo_sego@MacBook-Air Examen1 % ./build/Problema2.1_PM
+--------------------------------------+
| Valores propios mayores | Valores propios menores |
+--------------------------------------+
|   48.32                 |   15.95                 |
|   50.96                 |   15.98                 |
|   48.73                 |   16.07                 |
|   49.50                 |   16.04                 |
|   49.18                 |   16.04                 |
|   50.44                 |   16.07                 |
|   49.64                 |   15.71                 |
|   48.65                 |   15.96                 |
|   50.22                 |   16.01                 |
|   49.57                 |   15.89                 |
+--------------------------------------+
```

Listing 8: Terminal de ejecucion del metodo de potencias. Se muestran los valores propios mayores y menores.