Clase No. 5:

Repaso sobre lectura de datos desde la línea de comandos, manejo de memoria dinámica e introducción a la librería GSL

MAT–251

Dr. Alonso Ramírez Manzanares
CIMAT, A.C.
e-mail: alram@cimat.mx
web: http://www.cimat.mx/~alram/met_num/

Dr. Joaquín Peña Acevedo
CIMAT A.C.
e-mail: joaquin@cimat.mx

Parte 1:
Lectura de datos desde la línea de comandos

Lectura de datos desde la línea de comandos (I)

- En cada programa queremos realizar un conjunto de pruebas, lo que implica cambiar los valores de entrada de los algoritmos y no queremos recompilarlos en cada prueba.
- Eso requiere poder leer los datos de entrada de manera independiente.
- Usamos los argumentos de la función `main()` para obtener la información. Ejemplo (código *argumentos.cpp*)

```
int main(int argc, char **argv)     {
    int     i, n;
    double  dval;

    printf("\nLista de argumentos:\n");
    for(i=0; i<argc; ++i) {
        printf("\tArgumento %d: %s\n", i, argv[i]);
    }
    if(argc>0) n    = atoi(argv[1]);
    if(argc>1) dval = atof(argv[2]);

    printf("\nEntero    = %d\n", n);
    printf("Flotante  = %lf\n", dval);

    return(0);
}
```

Lectura de datos desde la línea de comandos (II)

Ejemplo de una llamada del programa desde la línea de comandos:

```
./argumentos -5 0.12345 y el resto de argumentos
```

Otro ejemplo se muestra en el código `evaluacionFnc.cpp`, el cual evalua tres funciones en intervalo definido por los valores que lee desde la línea de comandos:

```
./evaluacionFnc 10 1e11
```

En el código se puede cambiar tipo de las variables para ver como cambian los resultados.

Parte 2:
Introducción a la librería GSL

Librería GSL

La librería GSL comprende una amplia gama de métodos numéricos robustos.

- En la documentación se puede ver la cantidad y variedad de funciones que tiene.
- Hay que consultar la página del curso para las instrucciones de instalación, en el apartado "Extra".

Revisar los ejemplos

- `representacionIEEE.c` para ver la representación de punto flotante de un número.
- `serie.c` para modificar la forma en que se hace el redondeo (funciona en linux)
- `pruebaGsl.c` muestra que las funciones de la librería son robustas y eficientes.

En linux, para compilar:

```
gcc -o pruebaGsl pruebaGsl.c -lgsl -lgslcblas -lm
```

Parte 3:
Memoria para arreglos 1D y 2D

Memoria para arreglos 1D

- Revisar el código *memoria1D.cpp* para ver como crear arreglos 1D.
- En el programa se calcula el tiempo que demora hacer la suma de los elementos dos arreglos usando dos formas diferentes de accesar a los elementos del arreglo.

Memoria para arreglos 2D

Para asignar memoria a un arreglo 2D de tamaño para almacenar la información de una matriz $m \times n$ hacemos lo siguiente:

1. Creamos de manera dinámica un arreglo de apuntadores. Tantos apuntadores como filas tenga el arreglo (m).
2. Creamos un bloque de memoria de tamaño mn para almacenar todas las entradas.
3. Hacemos que el primer apuntador apunte a esta posición de memoria.
4. Hacemos que el resto de los apuntadores apunten a las posiciones del bloque de memoria reservada en donde empiezan cada fila de la matriz.

Paso 1: Creación del arreglo de apuntadores

```
double **createMatrix(int nr, int nc) {
    int i;
    double **mat;

    // Reservamos memoria
    mat = (double **) malloc( (nr)*sizeof(double *));
    if(mat==NULL) return(NULL);
    mat[0] = (double *) malloc( nr*nc*sizeof(double));
    if(mat[0]==NULL) return(NULL);
    for(i=1; i<nr; ++i)
        mat[i] = mat[i-1] + nc;
    return(mat);
}
```

Ahora queremos que `mat[i]` apunte a la fila correspondiente. Primero debemos reservar toda la memoria que vamos a necesitar y usar hacer que `mat[0]` apunte a ésta.

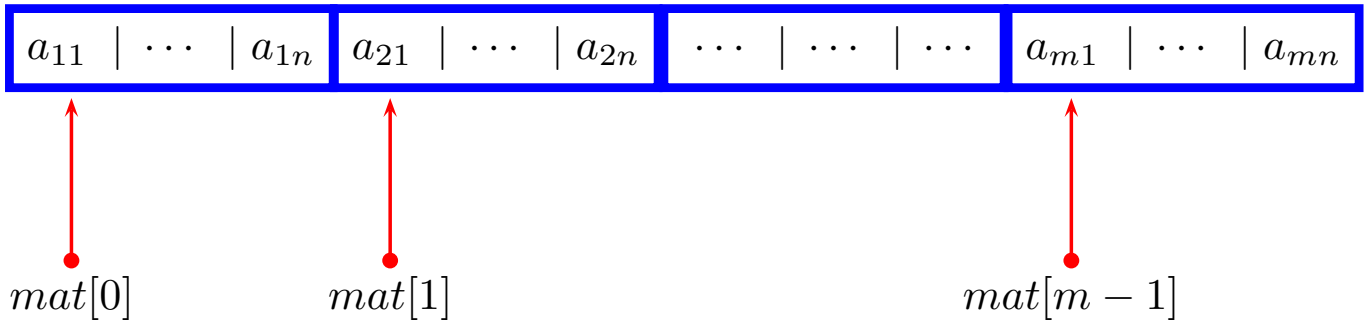
Pasos 2,3 y 4: Inicialización de los apuntadores

```
double **createMatrix(int nr, int nc) {
    int i;
    double **mat;

    // Reservamos memoria
    mat = (double **) malloc( (nr)*sizeof(double *));
    if(mat==NULL) return(NULL);
    mat[0] = (double *) malloc( nr*nc*sizeof(double));
    if(mat[0]==NULL) return(NULL);
    for(i=1; i<nr; ++i)
        mat[i] = mat[i-1] + nc;
    return(mat);
}
```

`mat[0]` apunta al inicio del bloque de memoria de tama no `nr*nc`. Cada apuntador `mat[i]` hace referencia a una localidad de memoria diferente.

Pasos 2, 3 y 4: Inicialización de los apuntadores



En el código, la primera línea crea el bloque de memoria para almacenar todos los elementos de la matriz (el espacio representado en azul en la figura) y la asigna a `mat[0]`.
El ciclo los inicializa el resto de los apuntadores a las posiciones en donde debe empezar cada fila.

Para $i = 0, \ldots, nr-1$ y $j = 0, \ldots, nc-1$, cada variable `mat[i][j]` sirve para almacenar el elemento en la columna $j+1$ de la fila $i+1$ de la matriz $\boldsymbol{A}$.

Liberación de memoria

Para liberar la memoria asignada a la matriz, primero liberamos la memoria del bloque creado para almacenar todos los elementos, y luego se libera la memoria que solicitamos para el arreglo de apuntadores:

```
void freeMatrix(double **mat) {
    free(mat[0]);
    free(mat);
}
```

Ejemplos

- Revisar el código *productoMatrizVector.cpp* para ver como crear arreglos 2D y 1D, y realizar operaciones entre ellos.
- En el programa se calcula el tiempo que demora hacer el producto de matriz y un vector, usando formas formas diferentes de accesar a los elementos del arreglo y crearlos.
- Revisar el programa *productoMatrices.cpp* para crear arreglos 2D y realizar el producto matricial entre ellos. También reporta el tiempo de la operación dependiendo de la forma en que se crean los arreglos y de como se accesan los elementos.

Parte 4:
Lectura y escritura de archivos binarios

Motivación

- Para probar los algoritmos que operan con matrices y vectores conviene leer esta información de archivos en vez de tener que capturarla manualmente.
- Para no introducir más errores y usar menos espacio en disco, conviene usar archivos binarios en la que las entradas de vectores y matrices son del tipo `double`.
- Para que todos puedan leer la información de los archivos, hay que establecer un formato.

Almacenamiento en memoria de matrices

Supongamos que tenemos una matriz $m \times n$,

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

- Queremos reservar memoria para almacenar las entradas de la matriz, de manera que sea compatible con la forma en que la librería GSL almacena la información.
- También queremos que se pueda accesar mediante un arreglo bidimensional, para facilitar la forma en que accesan a las entradas de la matriz.

$$a_{ij} \qquad \rightleftarrows \qquad a[i][j]$$

Convención para almacenamiento de datos

Para matrices:

- El archivo debe comenzar con dos enteros, m y n. El primero, m, debe indicar el número de filas de la matriz, y el segundo, n, debe indicar el número de columnas.
- El resto del archivo debe contener mn valores, que corresponden a las entradas de la matriz, almacenadas por filas: los primeros n valores corresponden a la primer fila,
  los siguientes n valores corresponden a la segunda fila, etc.

Para vectores:

- El archivo debe comenzar con un entero, m, que indica la dimensión del vector.
- El resto del archivo debe contener m valores, que corresponden a las entradas del vector.

Creación de la matriz a partir de un archivo

Para conseguir lo anterior, el procedimiento sería el siguiente:

1. Abrir el archivo y leer las dimensiones de la matriz.
2. Creamos de manera dinámica un arreglo de apuntadores, tantos como filas tenga la matriz.
3. Creamos un bloque de memoria de tamaño *mn* para almacenar todas las entradas.
4. Hacemos que el primer apuntador apunte a esta posición de memoria.
5. Hacemos que el resto de los apuntadores apunten a las posiciones del bloque de memoria reservada en donde empiezan cada fila de la matriz.
6. Leemos los elementos de la matriz del archivo.

Los pasos 2–5 son realizados por la función *createMatrix()*, vista en la sección anterior. De modo que solo se explican el resto de los pasos.

Paso 1: Lectura de las dimensiones de la matriz

La función que lee los datos de un archivo binario es:

```
double **readMatrix(char *cfile, int *nr, int *nc) {
    double **mat;
    FILE *f1 = fopen(cfile, "rb");

    if(!f1) return(NULL);
    // Lectura del tamaño de la matriz
    fread(nr, sizeof(int), 1, f1);
    fread(nc, sizeof(int), 1, f1);
    // Reservamos memoria
    mat = createMatrix(*nr, *nc);
    // Lectura de los datos
    fread(mat[0], sizeof(double), (*nr)*(*nc), f1);
    fclose(f1);
    return(mat);
}
```

La opcion "rb" hace que se habra el archivo de nombre cfile para lectura y especifica que el archivo es binario.

Paso 6: Lectura de los elementos de la matriz

```
double **readMatrix(char *cfile, int *nr, int *nc) {
    double **mat;
    FILE *f1 = fopen(cfile, "rb");

    if(!f1) return(NULL);
    // Lectura del tama no de la matriz
    fread(nr, sizeof(int), 1, f1);
    fread(nc, sizeof(int), 1, f1);
    // Reservamos memoria
    mat = createMatrix(*nr, *nc);
    // Lectura de los datos
    fread(mat[0], sizeof(double), (*nr)*(*nc), f1);
    fclose(f1);
    return(mat);
}
```

Se lee todos los elementos de la matriz y los almacena en el bloque de memoria al que hace referencia `mat[0]`.
Lo que resta de la función es la parte que cierra el archivo y devuelve el apuntador `mat`.

Escritura de datos binarios

La función siguiente escribe las entradas de una matriz en el formato acordado:

```
int writeMatrix(double **mat, int nr, int nc, char *cfile) {
   FILE *f1 = fopen(cfile, "wb");

   if(!f1) return(1);
   fwrite(&nr, sizeof(int), 1, f1);
   fwrite(&nc, sizeof(int), 1, f1);
   fwrite(mat[0], sizeof(double), nr*nc, f1);
   fclose(f1);
   return(0);
}
```

Lectura de vectores

```
double *readVector(char *cfile, int *nr) {
    double *vec;
    FILE *f1 = fopen(cfile, "rb");

    if(!f1) return(NULL);
    fread(nr, sizeof(int), 1, f1);
    vec = (double *) malloc( (*nr)*sizeof(double));
    if(vec==NULL) return(NULL);
    fread(vec, sizeof(double), *nr, f1);
    fclose(f1);
    return(vec);
}
```

Para usarla:

```
int nr;
double *vec = readVector(nombre_archivo, &nr);
```

Ejemplo

Hay que compilar el programa `lecturaBinarios.c`.
En la línea de comandos se especifica el nombre del archivo de la matriz y el nombre del archivo de un vector (en ese orden). Ejemplo:

```
./lecturaBinarios matA1.bin vecb1.bin
```

Como el programa imprime el contenido de los archivos en la pantalla, hay que probarlo con matrices pequeñas.

Ejemplo

Hay que compilar el programa `lecturaBinarios.c`.
En la línea de comandos se especifica el nombre del archivo de la matriz y el nombre del archivo de un vector (en ese orden). Ejemplo:

```
./lecturaBinarios matA1.bin vecb1.bin
```

Como el programa imprime el contenido de los archivos en la pantalla, hay que probarlo con matrices pequeñas.

Conversión de archivos de texto a binario

- Para probar los programas, se les proporciona datos (arreglos 1D y 2D) en archivos binarios para no perder precisión.
- Para generar estos archivos binarios a partir de un archivo de texto, puede usar como referencia el código:

`vecTxt2Bin.c` Convertir un archivo de texto que contiene un arreglo 1D a un archivo binario

`matTxt2Bin.c` Convertir un archivo de texto que contiene un arreglo 2D a un archivo binario