Manual de instrucciones

Marco Antonio Garrido Rojo*

Introducción

En este documento se recoge una explicación del funcionamiento de los diferentes scripts de MATLAB elaborados para la asignatura de Métodos Numéricos. La información aquí plasmada podrá consultarse libremente en mi perfil de GitHub junto con el código al que se refiere.

Antes de concluir con esta breve introducción conviene aclarar el modo en que debe llamarse a las funciones de estos scripts. Dado que MATLAB (o el intento de hacer algo parecido de forma libre FreeMat) restringe la visibilidad de sus funciones al archivo en el que se encuentra, salvo las de igual nombre que este, me vi en la obligación de seguir el consejo de mis amigos de StackOverflow[1] que me recomendaron hacer la chapuza de lo que en este mundo de matemáticos con ordenador se llaman struct (ojalá se hubieran copiado un poco más de Stroustrup):

```
function funs = makefuns
  funs.fun1=@fun1;
  funs.fun2=@fun2;
end
function y=fun1(x)
  y=x;
end
function z=fun2
  z=1;
end
```

En la estructura $makefuns$ del ejemplo (de divertido tiene poco pese al nombre) se agrupan las dos funciones del archivo, $fun1$ y $fun2$. Lo que viene a ser un puntero a estas dos funciones, nos permite utilizarlas fuera del archivo llamándolas mediante las variables f1 y f2.

```
>> myfuns = makefuns;
>> f1 = makefuns.fun1();
>> f2 = makefuns.fun2();
```

*https://github.com/MaSteve/MN

[1]http://stackoverflow.com/questions/3569933/

```
>> f1(5)
ans =
     5
>> f2()
ans =
     1
```

A falta de una orientación a objetos decente (porque parece que es lo que piden a gritos desde MATLAB restringiendo la visibilidad de las funciones de un archivo) hay que conformarse con este sistema.

Ahora sí pasamos a ver las distintas implementaciones.

Eliminación gaussiana

La eliminación gaussiana es un método para resolver sistemas de ecuaciones lineales. La implementación de dicho método se encuentra en el fichero gauss.m del repositorio.

Llamando a la función $gauss(A, b)$, donde A es una matriz cuadrada de rango máximo y b es un vector, obtenemos una matriz triangular asociada sobre la que podremos aplicar el método de remonte, apoyándonos en la variable *punt* que almacena las permutaciones de filas realizadas a la matriz original. La función $gausssolver(A, b, punt)$ aplica el remonte una vez procesada la información requerida.

La razón por la que se divide el algoritmo en dos funciones es para facilitar la tarea de resolver varios sistemas con la misma matriz A de coeficientes.

Factorización LU.

Un método de resolución alternativo de sistemas lineales es la factorización LU, implementado en lufact.m en el repositorio. Cabe recordar que este método no es aplicable a toda matriz cuadrada invertible.

La función $lufact(A, b)$ transforma la matriz A en su descomposición como producto de una matriz triangular inferior con una matriz triangular superior. Dichas matriz triangulares se agrupan en una única matriz *Amod* dado que la diagonal de la inferior son todo unos. Posteriormente la función $lusolver(Amod, b)$ resuelve los dos sistemas triangulares resultantes mediante el método del remonte.

De nuevo se contemplan dos funciones para facilitar la resolución de nuevos sistemas con la misma matriz de coeficientes.

Factorización de Cholesky.

La factorización de Cholesky (cholesky.m) es un caso particular de factorización LU para matrices simétricas definidas positivas. En este caso particular la descomposición se hace en dos matrices, la una traspuesta de la otra, lo que reduce el número de cálculos considerablemente. La función $cholesky(A, b)$ se encarga de la pertinente descomposición y $choleskysolver(Amod, b)$ de aplicar los dos métodos de remonte.

Resolución de sistemas tridiagonales.

Para sistemas tridiagonales existe una formula alternativa para su resolución donde primero se procesan unos coeficientes que posteriormente sirven para construir la solución en tiempo lineal. Este método es empleado posteriormente para el cálculo de funciones spline cúbicas. Su implementación se encuentra en tridiag.m y se utiliza llamando a la función $tridiag(A, b)$ que devuelve la solución en forma de vector.

Método de Jacobi.

Entramos en la sección de métodos iterativos para la resolución de sistemas lineales. Jacobi es un método aproximado de obtener la solución a un sistema a partir de un vector inicial (en este caso el vector nulo) a partir del cual se intenta construir una sucesión convergente a la solución. Su implementación se encuentra en el archivo jacobi.m del repositorio.

Llamando a $jacobi(A, b, iter, prec)$, donde A es la matriz de coeficientes, b el vector de términos independientes, $iter$ el número máximo de iteraciones, y $prec$ el error que se quiere cometer en la aproximación, se obtiene la solución en forma de vector y un $flag$ donde un uno indica que el algoritmo ha alcanzado la solución con el margen de error indicado.

Método de relajación.

Este otro método es similar al de Jacobi, con la diferencia de que entre el vector de partida y el vector resultado en un iteración se aplica una media ponderada de donde sale el sucesor definitivo.

Llamando a la función $relajacion(A, b, iter, prec, w)$ del archivo relajacion.m obtenemos una solución aproximada y un flag de mismo significado que el del método de Jacobi. La función añade un parámetro de entrada adicional w que determina el factor a aplicar en la media ponderada. Este parámetro debe estar entre 0 y 2.

Cabe recordar que hay situaciones en las que no se podrá obtener una sucesión convergente mediante estos métodos iterativos.

Polinomio de interpolación.

El polinomio de interpolación de una tabla de puntos es aquel polinomio de grado mínimo que pasa por todos ellos. El método implementado es el conocido método de la fórmula de interpolación de Newton (newton.m).

Dada una tabla con dos columnas, la primera con los valores x y la segunda con sus imágenes, llamando a la función $newtonP(table)$ se obtiene una tabla con las diferencias divididas de x_n de distinto orden, es decir, $f[x_i, \ldots, x_n], i \in [0, n]$. Con estos valores es posible construir el polinomio de interpolación.

En el mismo archivo se cuenta con una función que representa graficamente el polinomio, $plotNewton(pol, table)$, y una función, $addPoint(val, img, pol, Pi, table)$, que permite añadir un punto más a la interpolación llamándola con los datos ya procesados de la tabla, a saber, val, valor x, img, su imagen, pol, polinomio de interpolación actual, Pi, función Π_n actual, y $table$, con los valores de las diferencias divididas antes comentadas.

Funciones spline cúbicas.

Por último, se recoge un script spline.m con una implementación del método de interpolación mediante funciones spline cúbicas.

Llamando a $splineT(table)$ con una tabla con dos columnas como la del método anterior, se calcula una función a trozos de clase 2 que la aproxima. Para ello, obtiene unos coeficientes o momentos asociados a la tabla de valores aportada, que sirven para el cálculo de la función en los distintos intervalos, resolviendo un sistema tridiagonal.

Se incorporan dos funciones adicionales, $plotSpline(pols, table)$, que permite plotear la función interpoladora, y $fromFunction(f, x0, xn, div)$, que interpola directamente una función f (en forma de string) en el intervalo $[x_0, x_n]$ con un paso de div. Además, esta última función representa sobre la spline la función f para poder hacer una comparación visual.