Técnicas eficientes de programación en MATLAB para instrumentación

María del Mar Sanz Lluch

Borja Bordel Sánchez

Marina Pérez Jiménez

MATLAB aplicado a la instrumentación electrónica

Departamento de Electrónica Física (UPM)

PROGRAMA

- Procesado de datos en instrumentación
- Vectorización
- Tipos de funciones
- Funciones in-place
- Paquetes en MATLAB
- Introducción a la programación concurrente en MATLAB

PROCESADO DE DATOS EN INSTRUMENTACIÓN

- Una vez construidas las estructuras de datos que albergan los datos adquiridos desde el exterior, es probable que sea necesario realizar un procesado de datos antes de mostrar resultados
- La algorítmica en MATLAB es similar a JAVA o C/C++

PROCESADO DE DATOS EN INSTRUMENTACIÓN

- En general, el tratamiento se realiza en plataformas de amplios recursos que no precisan optimizaciones en código, memoria y/o procesador
- Sin embargo, en muchas aplicaciones está aumentando la cantidad de datos, su complejidad, y se incluyen requisitos de tiempo real

PROCESADO DE DATOS EN INSTRUMENTACIÓN

- Estos hecho obligan, en muchas plataformas, a plantearse una programación más eficiente
- En MATLAB, las metodologías de programación eficiente son algo diferentes de las que se encuentran en otros lenguajes, dado que el lenguaje M permite programación multiparadigma

VECTORIZACIÓN

- Una de las acciones más comunes en programación es recorrer arrays o matrices
- En la mayoría de los lenguajes este tipo de algoritmos se resuelven mediante bucles iterativos
- MATLAB dispone de mecanismos que permiten realizar estas operaciones en una sola sentencia, sin acceder uno a uno a todos los elementos
  - **VECTORIZACIÓN**

VECTORIZACIÓN

- El caso más simple de vectorización es la aplicación de una operación matemática, elemento a elemento, a dos vectores (o matrices) de las mismas dimensiones
- En MATLAB dichas operaciones pueden realizarse de la misma manera que si fueran escalares
- Debe anteponerse un punto (.) al operador correspondiente

VECTORIZACIÓN

- Ejemplo: multiplicar los elementos de dos arrays

```
for i = 1:length(v)
       z(i) = v(i)*u(i)
end
```

Sin vectorización

```
z = v.*u
```

Con vectorización

VECTORIZACIÓN

- Muchas operaciones definidas mediante funciones de MATLAB (de librería) también admiten vectorización
  - Pueden aplicarse indistintamente a escalares o arrays
  - Ejemplos
    - *floor* Para calcular la parte entera de un decimal
    - *ceil* Para redondear al entero inmediatamente superior

VECTORIZACIÓN

- En un caso más avanzado, se quiere relacionar mediante operaciones dos vectores (o matrices) que no tienen las mismas dimensiones
- Por ejemplo, restar a cada columna de una matriz un vector columna dado
- En este caso existen tres alternativas (a parte de la clásica solución iterativa)

VECTORIZACIÓN

- Solución iterativa (clásica)

```
for i = 1:size(v,1)
        for j = 1:length(u)
                z(i,j) = v(i,j)- u(j)
        end
end
```

- Se emplea poco en MATLAB

VECTORIZACIÓN

- Solución semi-vectorizada
- Se basa en el operador (:) que permite extraer todos los elementos de una de las dimensiones de un array
- Ejemplo, todos los elementos de la primera fila
  - Se entendería como “todas las columnas” de la fila primera

```
v(1,:)
```

VECTORIZACIÓN

- El resultado final, con técnicas de semi-vectorización en nuestro ejemplo sería

```
for j = 1:size(v,2)
       z(:,j) = v(:,j)- u
end
```

- Es el tipo de algoritmo más habitual en usuarios de tipo “medio”

VECTORIZACIÓN

- La tercera alternativa es expandir las matrices hasta que coincidan sus dimensiones, y luego operar con vectorización
- Para ello se puede emplear la función *repmat*
- La función *repmat* expande una matriz replicándola en filas y columnas tanta veces como se indique

VECTORIZACIÓN

```
repmat(matriz, replicas_fil, replicas_col);
```

- Ejemplo de uso

```
 repmat([5 6; 4 7], 3, 2)

ans =

     5     6     5     6
     4     7     4     7
     5     6     5     6
     4     7     4     7
     5     6     5     6
     4     7     4     7
```

VECTORIZACIÓN

- El resultado final, con técnicas de expansión en nuestro ejemplo sería

```
z = v - repmat(u, size(v,2), 1)
```

- Es el tipo de algoritmo más habitual en usuarios de tipo “medio”

VECTORIZACIÓN

- La última alternativa es emplear la función *bsxfun*, diseñada para solucionar este problema específicamente
- *bsxfun* aplica una función que recibe como manejador (ver Tema 3) a dos matrices por columnas, expandiéndolas si es preciso

```
result = bsxfun(manejador, array1, array2);
```

VECTORIZACIÓN

- Nuestro ejemplo, aplicando la función *bsxfun* quedaría como

```
z = bsxfun(@minus, v, u);
```

- Es la notación más compacta, pero requiere crear la función que se pasa como manejador

VECTORIZACIÓN

- Una última operación que se puede realizar con técnicas de vectorización en MATLAB es la selección de elementos
- Por ejemplo: obtener los elementos de array cuyo valor sea mayor que cero
- En MATLAB este tipo de condiciones pueden utilizarse como un índice más

VECTORIZACIÓN

```
for i = 1:length(v)
        if(v(i)>0)
                z(j) = v(i);
                j++;
        end
end
```

Sin vectorización

```
z = v(v > 0);
```

Con vectorización

- En estos casos la solución vectorizada es la preferida

TIPOS DE FUNCIONES

- Hay cinco tipos de funciones en MATLAB
  - Principales
  - Locales
  - Anidadas
  - Privadas
  - Anónimas
- Cuando se realiza programación con objetos, además, se pueden emplear métodos (ver Tema 3)

TIPOS DE FUNCIONES

- Las funciones principales deben estar escritas en un fichero nombrado igual que la función

```
function [sal1, sal2,…, saln] = myfunction(en1, en2,…, enn)
        …
end
```

TIPOS DE FUNCIONES

- Si no se desea explicitar las variables de entrada y/o salida se pueden emplear palabras reservadas en su lugar
  - *varargout* para indicar cualquier tipo y número de variables de salida
  - *varargin* para indicar cualquier tipo y número de variables de entrada
  - *nargout* para indicar ninguna variable de salida
  - *nargin* para indicar ninguna variable de entrada

TIPOS DE FUNCIONES

- Las funciones locales son funciones que se codifican en el mismo fichero que una función principal, pero fuera de la misma. Son sólo visibles por las funciones escritas en ese mismo fichero

```
function [sal1, sal2,…, saln] = myfunction(en1, en2,…, enn)
        …
end
function [sal1, sal2,…, saln] = local(en1, en2,…, enn)
        …
end
```

TIPOS DE FUNCIONES

- Las funciones anidadas son funciones que se codifican dentro de otras funciones (habitualmente justo antes del cierre). Sólo son vistas por la función “padre”. Pueden hacer uso de todas las variables del “padre”

```
function [sal1, sal2,…, saln] = myfunction(en1, en2,…, enn)
        function [sal1, sal2,…, saln] = anidada(en1, en2,…, enn)
                …
        end
end
```

TIPOS DE FUNCIONES

- Las funciones privadas son funciones habituales cuyos ficheros se almacenan en una carpeta llamada “private”. Estas funciones sólo pueden ser vistas desde los ficheros situados en el directorio inmediatamente superior.

TIPOS DE FUNCIONES

- Las funciones anónimas son funciones que no quedan almacenadas en un fichero. Se define una expresión matemática que se mapea en memoria volátil y de la que se obtiene un puntero para referenciarla
- Una función puede cambiar de forma dinámica

```
myfunctionAnonima = @(en1) en1^2;
```

FUNCIONES IN-PLACE

- Todas las funciones pueden ser escritas siguiendo el modelo “in-place”
- Se emplea para modificar de forma permanente una de las variables de entrada sin que haya copia vaga
  - Ver Tema 3
- Basta colocar como variable de salida la variable de entrada que se desea modificar

FUNCIONES IN-PLACE

```
function [en1] = myfunction(en1)
        en1 = … ;
end
```

- No es recomendable salvo que se esté seguro de su necesidad
- Puede dar lugar a errores si se emplean nombres de variables comunes (x, y, etc.)

PAQUETES EN MATLAB

- En MATLAB los paquetes se crean tomando como referencia el sistema de ficheros
- No es necesario incluir ninguna sentencia en los scripts, clases o funciones para explicitar el paquete al que pertenecen
  - Un fichero pertenece al paquete que se llama como el directorio que lo contiene
- En MATLAB un mismo paquete puede contener clases, funciones, scripts…

INTRODUCCIÓN A LA PROGRAMACIÓN CONCURRENTE EN MATLAB

- Se distinguen dos tipos de concurrencia en MATLAB
  - Implícita: Aquella que se soporta en mecanismos propios del lenguaje.
    - P. ej. Operaciones especiales que se ejecutan en varias hebras de trabajo paralelas

INTRODUCCIÓN A LA PROGRAMACIÓN CONCURRENTE EN MATLAB

- Explícita: Aquella que se soporta en mecanismos del entorno de ejecución.
  - Para lograr concurrencia explícita hay que añadir sentencias especiales destinadas a modificar el comportamiento habitual del motor de ejecución

INTRODUCCIÓN A LA PROGRAMACIÓN CONCURRENTE EN MATLAB

- El modelo de concurrencia en MATLAB sigue el paradigma MIMD
  - Múltiples instrucciones, múltiples datos
  - Cada procesador ejecuta código de forma asíncrona e independiente
- Admite también SIMD, pero su uso es complejo

INTRODUCCIÓN A LA PROGRAMACIÓN CONCURRENTE EN MATLAB

- Aunque sólo es una recomendación, la concurrencia en MATLAB solo es eficiente en arquitecturas multinúcleo

INTRODUCCIÓN A LA PROGRAMACIÓN CONCURRENTE EN MATLAB

- El entorno donde se arranca la ejecución (llamado cliente) orquesta la distribución y ejecución el código en un conjunto de procesos independientes llamados (trabajadores o *workers*)
- Cada trabajador ejecuta un “trozo” de código que se llama “tarea”

INTRODUCCIÓN A LA PROGRAMACIÓN CONCURRENTE EN MATLAB

- La programación concurrente debe emplearse, básicamente, cuando enfrentemos alguno de estos problemas
  - Bucles de gran número de iteraciones
  - Bucles con iteraciones muy pesadas y largas
  - Scripts donde el código pueda separarse en varias tareas independientes de forma natural

INTRODUCCIÓN A LA PROGRAMACIÓN CONCURRENTE EN MATLAB

  - Cuando se evalúen conjuntos de datos muy grandes
  - Cuando queramos que una ejecución no bloquee el entorno MATLAB
    - Ejecución en segundo plano
- En los tres primeros casos, se empleará concurrencia implícita. En los dos últimos explícita

INTRODUCCIÓN A LA PROGRAMACIÓN CONCURRENTE EN MATLAB

- La utilización más extendida de la concurrencia implícita en MATLAB son los bucles *for* paralelos

```
parfor i=1:n
        …
end
```

INTRODUCCIÓN A LA PROGRAMACIÓN CONCURRENTE EN MATLAB

- Basta cambiar la palabra reservada *for* por *parfor* para que el motor de ejecución organice de forma automática el paralelismo
- Se crea una *piscina* de tareas (siendo cada tarea una iteración), de donde una serie de *trabajadores* van extrayendo el código que van a ejecutar

INTRODUCCIÓN A LA PROGRAMACIÓN CONCURRENTE EN MATLAB

- No hay ninguna garantía de que el orden de iteración se respete
  - Es decir, la iteración $i = 200$, puede ejecutarse antes que $i = 1$
- Por tanto, es imprescindible que las iteraciones sean independientes

INTRODUCCIÓN A LA PROGRAMACIÓN CONCURRENTE EN MATLAB

- Al finalizar la ejecución los resultados serán visibles en el cliente como si de un bucle *for* tradicional se tratarse

INTRODUCCIÓN A LA PROGRAMACIÓN CONCURRENTE EN MATLAB

- La concurrencia explícita sólo estará disponible si se dispone del *Parallel Processing Toolbox* instalado
- Aunque permite organizar sistemas concurrentes a gran escala y de forma compleja, aquí sólo vamos a revisar un uso inicial
  - Pero suficiente para la mayoría de los casos

INTRODUCCIÓN A LA PROGRAMACIÓN CONCURRENTE EN MATLAB

- Para solicitar que un determinado script se ejecute por un trabajador distinto del cliente se emplea el comando *batch*

```
trabajador = batch ('myScript')
```

- La función *batch* devuelve el ID del trabajador al que se le ha asignado la tarea

INTRODUCCIÓN A LA PROGRAMACIÓN CONCURRENTE EN MATLAB

- Si se desea que el script se distribuya entre más de un trabajador, basta indicarlo al invocar el comando

```
trabajador = batch ('myScript', 'pool', 3);
```

- En este caso 3 trabajadores ejecutarán el script

INTRODUCCIÓN A LA PROGRAMACIÓN CONCURRENTE EN MATLAB

- Cuando se arrancan varios trabajadores con el comando *batch* siempre se arranca uno adicional que coordina la ejecución
  - *batch* es no bloqueante
- Al contrario que en la concurrencia implícita, cuando *batch* termina no devuelve los resultados de ejecución, es preciso recuperarlos

INTRODUCCIÓN A LA PROGRAMACIÓN CONCURRENTE EN MATLAB

```
load (trabajador, ‘nombre_variable’);
```

- Si se desea, se puede bloquear el cliente hasta que termine de ejecutar sus tareas un determinado trabajador

```
wait (trabajador);
```

INTRODUCCIÓN A LA PROGRAMACIÓN CONCURRENTE EN MATLAB

- Finalmente, es preciso vaciar la zona de memoria ocupada por los trabajadores, una vez se hayan recuperado todos los resultados

```
delete (trabajador);
```