Computación de Altas Prestaciones

Sesión 11

1

Contenido

1. Librerías numéricas.
2. Núcleos computacionales: Blas y Lapack.
3. Vectorización en Matlab.
4. Vectorización en Matlab mediante la función find.
5. Enlaces sobre vectorización.
6. Preasignación de memoria.

DSIC
DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

2

1. Librerías numéricas

- Base para desarrollar aplicaciones de ingeniería.
- Problemas básicos de algebra lineal numérica:
  - Sistemas de ecuaciones lineales.
  - Mínimos cuadrados.
  - Valores propios.
  - Valores singulares.

1. Librerías numéricas

- Características deseables:
  - Eficiencia y flexibilidad.
  - Facilidad de uso.
  - Portabilidad.
  - Robustez.
- Algoritmo numéricamente efectivo:
  - Propósito general.
  - Fiable.
  - Estable.
  - Eficiente.

Dependiente de la máquina

2. Núcleos computacionales

OBJETIVO: Portabilidad y eficiencia

NÚCLEOS COMPUTACIONALES: Definición de una serie de rutinas estándar básicas de operaciones de álgebra lineal

BLAS: Basic Linear Algebra Subprograms.

LAPACK: Linear Algebra PACKage.

2.1. BLAS: Niveles

- Tres niveles:
  - Nivel 1: operaciones vector-vector $y \leftarrow \alpha x + y$
    - Operaciones O(n) - Datos O(n)
  - Nivel 2: operaciones matriz-vector $y \leftarrow \alpha A x + \beta y$
    - Operaciones $O(n^2)$ - Datos $O(n^2)$
  - Nivel 3: operaciones matriz-matriz $C \leftarrow \alpha A B + \beta C$
    - Operaciones $O(n^3)$ - Datos $O(n^2)$
- Inicialmente sólo BLAS-1 (1973):
  - El resto de las operaciones se definieron posteriormente (1984-88), por ser necesarias para el proyecto LAPACK.

2.1. BLAS: Nomenclatura

BLAS1: xaaaa

BLAS 2 y 3: xyyzz

x (Tipo de datos)	*yy (Tipo de matrices)*
S: Reales simple precisión D: Reales doble precisión C: Complejos simple precisión Z: Complejos doble precisión	GE: Rectangulares generales SY: Simétricas HE: Hermitianas TR: Triangulares (otros en BLAS 2)
zz (Operación)	
MV: producto matriz por vector MM: producto matriz por matriz SV, SM: resolución de sistemas triangulares R, R2, RK, R2K: actualizaciones de rango 1, rango 2, ...	

2.1. BLAS 1
Operaciones Vector-Vector

- _AXPY: $y = \alpha x + y$
- _SCAL: $x = \alpha x$
- _SWAP: intercambia x e y
- _COPY: $y = x$
- _DOT : producto escalar $x^T y$
- _NRM2: calcula la 2-norma de x
- _ROTG: genera rotación
- _ROT : aplica rotación

siendo x e y vectores y α un escalar

2.1. BLAS 2
Operaciones Matriz-Vector

- Productos matriz por vector:
  - _GEMV: $y = \alpha\ op(A)\ x + \beta\ y$
  - _TRMV: $x = op(A)\ x$ A triangular
- Actualizaciones de rango uno o rango dos:
  - _GER : $A = \alpha\ x\ y^T + A$
  - _SYR2: $A = \alpha\ x\ y^T + \alpha\ y\ x^T + A$ A simétrica
- Resolución de sistemas triangulares:
  - _TRSV: $x = op(A^{-1})\ x$ A triangular

siendo A una matriz, x e y vectores, α y β escalares y $op(A)=A, A^T, A^H$

9

2.1. BLAS 3
Operaciones Matriz-Matriz

- Productos matriz por matriz:
  - _GEMM: $C = \alpha\ op(A)\ op(B) + \beta C$
  - _TRMM: $B = \alpha\ op(A)\ B$, $B = \alpha\ B\ op(A)$ A triangular
- Actualizaciones de rango k y 2k de una matriz simétrica:
  - _SYRK : $C = \alpha AA^T + \beta C$, $C = \alpha A^TA + \beta C$ C simétrica.
  - _SYR2K: $C = \alpha AB^T + \alpha BA^T + \beta C$, $C = \alpha A^TB + \alpha B^TA + \beta C$
- Resolución de sistemas triangulares múltiples:
  - _TRSM: $B = \alpha\ op(A^{-1})\ B$, $B = \alpha\ B\ op(A^{-1})$ A triangular

siendo A, B, C matrices, α y β escalares y $op(A)=A, A^T, A^H$

10

2.1. BLAS: Llamadas

- Fortran:

```
CALL DGEMV(TRANS, M, N, ALPHA, A, LDA, X, INCX, BETA, Y,
           INCY)
```

- C:

```
dgemv_(&TRANS, &M, &N, &ALPHA, A, &LDA, X, &INCX, &BETA, Y,
       &INCY)
```

- Parámetros:
  - BLAS1: dim, escalar, vector, vector
  - BLAS2: opc, dim, ab, escalar, matriz, vector, escalar, vector, opc, dim, escalar, vector, vector, matriz
  - BLAS3: opc, dim, escalar, matriz, matriz, escalar, matriz

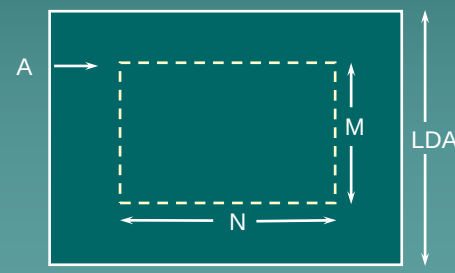
(sólo aparecen algunos de los parámetros, según la rutina)

2.1. BLAS: Parámetros

- Escalares:
  - ALPHA, BETA.
- Opciones:
  - TRANS = 'N', 'T', 'C' - Si la rutina opera con A, A^T, A^H.
  - UPLO = 'U', 'L' - Triángulo superior o inferior.
  - DIAG = 'U', 'N' - Si la matriz es triangular unitaria o no.
  - SIDE = 'L', 'R' - Si se multiplica por la izquierda o derecha.
- Ancho de banda:
  - KL, KU: Valores inferior y superior.
  - K: En matrices triangulares, simétricas o hermitianas.

2.1. BLAS: Parámetros

- Matrices:
  - A: Primera posición de memoria ocupada por la matriz.
  - LDA (Leading Dimension o dimensión máxima): Distancia entre dos elementos de la misma fila.



13

2.1. BLAS: Parámetros

- Dimensiones:
  - M, N: Número de filas y columnas.
- Vectores:
  - X: Primera posición de memoria ocupada por el vector.
  - INCX: Distancia entre dos elementos del vector.
    - INCX=1 Vector contiguo en memoria.
    - INCX>1 Posiciones no consecutivas.
    - INCX<1 Elementos en orden inverso.

Ejemplo: Norma de la fila 2 de una matriz de tamaño M x N:
dnrm2_(&N, &A[1], &M)

14

2.1. BLAS: disponibilidad

- Versiones disponibles:
  - Versión FORTRAN en código fuente:
    http://www.netlib.org/blas
  - Versiones optimizadas libres:
    - GEMM-based BLAS, GOTO BLAS, ATLAS.
  - MKL (Intel), ACML (AMD), …
  - Versiones para C y para Java.
- Proyectos relacionados:
  - Sparse BLAS: Para matrices dispersas.
  - PBLAS: BLAS en paralelo para máquinas de memoria distribuida.
  - BLACS: Rutinas de comunicaciones para máquinas de memoria distribuida.
  - cuBLAS: Para GPUs de NVIDIA.

2.1. BLAS: conclusiones

- Interfaz uniforme para todas las plataformas:
  - Portabilidad: El programador no considera detalles de la arquitectura.
  - Eficiencia: Permite disponer de implementaciones optimizadas.
- Eficiencia:
  - BLAS 1 - Iba bien en máquinas escalares.
  - BLAS 2 - Permite aprovechar arquitecturas vectoriales.
  - BLAS 3 - Mayor ratio flops / referencias a memoria.
    Ideal para máquinas con jerarquía de memorias.
- Conclusión:
  - Conviene usar BLAS 3: Operaciones matriz-matriz mediante algoritmos orientados a bloques.

2.2. LAPACK

- Librería para resolución de problemas comunes de álgebra lineal:
  - Portable: se basa en el uso de BLAS.
  - Eficiente: algoritmos orientados a bloques.
- Tipos de problemas:
  - Sistemas de ecuaciones lineales.
  - Problemas de mínimos cuadrados.
  - Cálculo de valores propios.
  - Cálculo de valores singulares.
- Incluye descomposiciones que permiten resolver los problemas anteriores (LU, Cholesky, QR, SVD, Schur), número de condición, etc.

2.2. LAPACK: Antecedentes

- Primera versión pública: 1991.
- Basada en BLAS 1, surge como versión mejorada de:
  - LINPACK: Sistemas de ecuaciones lineales.
  - EISPACK: Problemas de valores propios.
- Las librerías anteriores son ineficientes en las máquinas actuales: no tienen en cuenta la jerarquía de memorias.
- LAPACK rediseñado para minimizar el tráfico de datos:
  - Nuevos algoritmos, algunos mejorados para mayor precisión.
  - Diseño para máquinas paralelas (memoria distribuida).
  - Algunos diseños recientes para multicores (memoria compartida).

2.3. LAPACK: Características

- Principales características:
  - Rutinas de alto nivel para problemas complejos.
  - Especial atención al problema del movimiento de datos entre memorias.
  - Eficiencia y portabilidad mediante llamadas a BLAS.
  - Para matrices densas y banda, no dispersas.
  - Dominio público: http://www.netlib.org/lapack

2.3. LAPACK: Conclusiones

- Proyectos relacionados antiguos:
  - CLAPACK.
  - JLAPACK.
  - LAPACK++.
  - LAPACK95.
  - ScaLAPACK: Implementación en multiprocesadores de memoria distribuida.
- Proyectos relacionados actuales:
  - PLASMA, MKL o ACML (Multicores).
  - MAGMA o cuSolver (GPUs).

3. Vectorización en Matlab

- El concepto de "vectorización de código" en Matlab es diferente (aunque relacionado) del concepto de vectorización en programación general.
- Un bucle se ha vectorizado si se puede ejecutar en unidades vectoriales (SSE, AVX) aprovechando sus múltiples registros.
- En Matlab, se llama "vectorizar" a sustituir los bucles por expresiones vectoriales o matriciales equivalentes que al estar basadas en código compilado se ejecutan de forma más eficiente.

3. Vectorización en Matlab

- Ejemplo 1:

```
v=[1 2 3 4];
w=[5 6 7 8];
for i=1:length(v)
   z(i)=v(i)+w(i);
end

z=v+w
```

Suma de vectores por bucle

Suma de vectores "vectorizada"

3. Vectorización en Matlab

- Ejemplo 2:

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

← Producto de vectores por bucle, "componente a componente"

```
z=v.*w
```

← Producto de vectores "componente a componente", vectorizado

```
for i=1:length(v)
    z(i)=v(i)/w(i);
end
```

← División de vectores por bucle, "componente a componente"

```
z=v./w
```

← División de vectores "componente a componente", vectorizada

El punto modifica el comportamiento del operador, realizando la operación "componente a componente".

3. Vectorización en Matlab

- Ejemplo 3: Queremos calcular el volumen de 1000000 conos, a partir de sus diámetros (D) y alturas (H) generados aleatoriamente:

```
n=1000000;
D=rand(1,n);
H=rand(1,n);
% Versión con bucle
tic
for i=1:n
    V(i) = 1/12*pi*D(i)^2*H(i);
end
tiempo_bucle=toc

% Versión vectorizada
tic
V=(1/12)*pi*D.^2.*H;    % Observa el uso del punto antes del operador
tiempo_vectorizado=toc
```

3. Vectorización en Matlab

- Las funciones matemáticas elementales (sqrt, …) o trigonométricas (sin, cos, tan, …) de Matlab admiten como argumentos vectores o matrices, lo cual es útil para vectorizar código.
- Ejemplo 4:

```
t = 0:0.01:20*pi;
tic
for i=1:length(t)
    y(i) = sin(t(i));
end
tiempo_bucle=toc

% versión vectorizada
tic
y = sin(t);
tiempo_vectorizada=toc
```


4. Vectorización en Matlab mediante la función “find”

- Toma como dato de entrada expresiones relacionales basadas en vectores o matrices.
- Devuelve como resultado un vector con los índices de los vectores o matrices donde la expresión relacional es verdadera.
- En muchos casos puede ser útil para sustituir los bucles habitualmente compuestos por un “if”.

4. Vectorización en Matlab mediante la función "find"

- Ejemplo 1: Deseamos calcular las posiciones de los "máximos locales" de un vector v, es decir, los índices tales que v(i) > v(i-1) y v(i) > v(i+1).

Versión con bucle:

```
function ind = maximos_locales_bucle (v)
n=length(v);
ind=[];
for i=2:n-1
   if v(i)>v(i-1) && v(i)>v(i+1)
      ind=[ind i];
   end
end
```

27

4. Vectorización en Matlab mediante la función "find"

Versión vectorizada:

```
function ind = maximos_locales_vec (v)
n=length(v);
ind=find(v(2:n-1)>v(1:n-2)  &  v(2:n-1)>v(3:n));
ind=ind+1; % necesario para que dé el mismo resultado que la
           % versión con bucle.
```

28

4. Vectorización en Matlab mediante la función "find"

- Ejemplo 2: Deseamos recorrer una matriz A y sustituir los elementos negativos por ceros.

Con bucle:

```
[m,n]=size(A);
for i=1:m
  for j=1:n
    if A(i,j) <0
      A(i,j) = 0;
    end
  end
end
```

Vectorizando:

```
ind = find(A < 0);
A(ind)=0;
```

29

4. Vectorización en Matlab mediante la función "find"

- Ejercicio 1. Escribe una función que, dada una matriz M, calcule sin usar bucles cuántas columnas posee en las que la suma de sus elementos sea positiva.

Solución:

```
function ncols = columnas_suma_positiva (M)
suma = sum(M);
indices = find (suma > 0);
ncols = length (indices);
```

30

4. Vectorización en Matlab mediante la función "find"

- Ejercicio 2. Escribe, sin usar bucles, una función en Matlab a la que se le pase un vector y devuelva un 1 si el vector es capicúa o un 0 si no lo es.

  Ejemplo: [3 6 7 9 10 9 7 6 3] es capicúa.

  Solución:

```
function respuesta = capicua (x)
n = length (x);
indices = find (x == x(n:-1:1));
longitud = length (indices);
respuesta = longitud == n;
```

31

5. Enlaces sobre vectorización

- http://www.mathworks.es/es/help/matlab/matlab_prog/vectorization.html
- http://www.mathworks.es/es/help/matlab/matlab_prog/techniques-for-improving-performance.html
- http://www.ee.columbia.edu/~marios/matlab/Vectorization.pdf

32

6. Preasignación de memoria

- En Matlab no es necesario indicar a priori la dimensión de vectores ni matrices y, en su generación, automáticamente readaptan su dimensión (si es necesario):

```
for i=1:n
   V(i) = 1/12*pi*D(i)^2*H(i);
end
```

- Sin embargo, el proceso de redimensionar el vector V en cada iteración es costoso. Dimensionando previamente el vector V (rellenándolo de ceros o de cualquier otro modo) el bucle es considerablemente más rápido:

```
V=zeros(1,n)
for i=1:n
   V(i) = 1/12*pi*D(i)^2*H(i);
end
```

- El editor de Matlab detecta estos casos y nos avisa.