Programación: multiplicación de matrices por vectores usando operaciones lineales con las columnas (un tema del curso "Álgebra Lineal Numérica")

Egor Maximenko

http://www.egormaximenko.com

Instituto Politécnico Nacional
Escuela Superior de Física y Matemáticas
Ciudad de México

2 de marzo de 2021

Objetivo:

- programar el algoritmo de multiplicación de matrices por vectores, basado en las operaciones axpy (operaciones lineales con vectores);
- analizar la complejidad del algoritmo;
- medir el tiempo de ejecución de los programas.

Prerrequisitos:

- programación con funciones, arreglos y ciclos;
- construcción de matrices y vectores pseudoaleatorios;
- ciclos “for” anidados;
- el algoritmo axpy.

Problema de programación

Programar una función de dos argumentos A, b que calcule y devuelva el producto Ab, donde A es una matriz y b es un vector.

Entrada: una matriz A y un vector b.

Propiedades de la entrada: se supone que el número de columnas de A coincide con la longitud de b.

Salida: el producto Ab.

Ya habíamos resuelto esto problema usando el producto punto.
Ahora vamos a resolverlo usando la operación axpy.

Repaso: el producto de una matriz por un vector es una combinación lineal de las columnas de la matriz

$$Ax = \begin{bmatrix} A_{1,1}x_1 + A_{1,2}x_2 + A_{1,3}x_3 \\ A_{2,1}x_1 + A_{2,2}x_2 + A_{2,3}x_3 \end{bmatrix}$$

Repaso: el producto de una matriz por un vector es una combinación lineal de las columnas de la matriz

$$Ax = \begin{bmatrix} A_{1,1}x_1 + A_{1,2}x_2 + A_{1,3}x_3 \\ A_{2,1}x_1 + A_{2,2}x_2 + A_{2,3}x_3 \end{bmatrix} = \begin{bmatrix} A_{1,1}x_1 \\ A_{2,1}x_1 \end{bmatrix} + \begin{bmatrix} A_{1,2}x_2 \\ A_{2,2}x_2 \end{bmatrix} + \begin{bmatrix} A_{1,3}x_3 \\ A_{2,3}x_3 \end{bmatrix}$$

Repaso: el producto de una matriz por un vector es una combinación lineal de las columnas de la matriz

$$Ax = \begin{bmatrix} A_{1,1}x_1 + A_{1,2}x_2 + A_{1,3}x_3 \\ A_{2,1}x_1 + A_{2,2}x_2 + A_{2,3}x_3 \end{bmatrix} = \begin{bmatrix} A_{1,1}x_1 \\ A_{2,1}x_1 \end{bmatrix} + \begin{bmatrix} A_{1,2}x_2 \\ A_{2,2}x_2 \end{bmatrix} + \begin{bmatrix} A_{1,3}x_3 \\ A_{2,3}x_3 \end{bmatrix}$$

$$= x_1 \begin{bmatrix} A_{1,1} \\ A_{2,1} \end{bmatrix} + x_2 \begin{bmatrix} A_{1,2} \\ A_{2,2} \end{bmatrix} + x_3 \begin{bmatrix} A_{1,3} \\ A_{2,3} \end{bmatrix}$$

Repaso: el producto de una matriz por un vector es una combinación lineal de las columnas de la matriz

$$Ax = \begin{bmatrix} A_{1,1}x_1 + A_{1,2}x_2 + A_{1,3}x_3 \\ A_{2,1}x_1 + A_{2,2}x_2 + A_{2,3}x_3 \end{bmatrix} = \begin{bmatrix} A_{1,1}x_1 \\ A_{2,1}x_1 \end{bmatrix} + \begin{bmatrix} A_{1,2}x_2 \\ A_{2,2}x_2 \end{bmatrix} + \begin{bmatrix} A_{1,3}x_3 \\ A_{2,3}x_3 \end{bmatrix}$$

$$= x_1 \begin{bmatrix} A_{1,1} \\ A_{2,1} \end{bmatrix} + x_2 \begin{bmatrix} A_{1,2} \\ A_{2,2} \end{bmatrix} + x_3 \begin{bmatrix} A_{1,3} \\ A_{2,3} \end{bmatrix} = x_1 A_{*,1} + x_2 A_{*,2} + x_3 A_{*,3}.$$

Repaso de algunas definiciones

Para demostrar la regla en su forma general, necesitamos definiciones formales.

Definición del producto de una matriz por un vector.
Sea $A \in \mathcal{M}_{m\times n}(\mathbb{R})$ y sea $b \in \mathbb{R}^n$. Entonces, por definición, $Ab \in \mathbb{R}^m$ y

$$\forall j \in \{1, \ldots, m\} \qquad (Ab)_j = \sum_{k=1}^{n} A_{j,k} b_k.$$

Repaso de algunas definiciones

Para demostrar la regla en su forma general, necesitamos definiciones formales.

Definición del producto de una matriz por un vector.
Sea $A \in \mathcal{M}_{m\times n}(\mathbb{R})$ y sea $b \in \mathbb{R}^n$. Entonces, por definición, $Ab \in \mathbb{R}^m$ y

$$\forall j \in \{1, \dots, m\} \qquad (Ab)_j = \sum_{k=1}^{n} A_{j,k} b_k.$$

Definición de la columna de una matriz.
Sea $A \in \mathcal{M}_{m\times n}(\mathbb{R})$ y sea $k \in \{1, \dots, n\}$. Entonces, por definición, $A_{*,k} \in \mathbb{R}^m$ y

$$\forall j \in \{1, \dots, m\} \qquad (A_{*,k})_j = A_{j,k}.$$

Repaso de algunas definiciones

Definición de la suma de dos vectores.

Sean $x, y \in \mathbb{R}^n$. Entonces $x + y \in \mathbb{R}^n$ y

$$\forall j \in \{1, \ldots, m\} \qquad (x + y)_j = x_j + y_j.$$

Definición del producto de un escalar por un vector.

Sea $\lambda \in \mathbb{R}$ y sea $x \in \mathbb{R}^n$. Entonces $\lambda x \in \mathbb{R}^n$ y

$$\forall j \in \{1, \ldots, m\} \qquad (\lambda x)_j = \lambda x_j.$$

El producto de una matriz por un vector como una combinación lineal de las columnas de la matriz

Proposición

Sea $A \in \mathcal{M}_{m\times n}(\mathbb{R})$ y sea $b \in \mathbb{R}^n$. Entonces

$$Ab = \sum_{k=1}^{n} b_k A_{*,k}.$$

Demostración, inicio

Queremos demostrar la fórmula

$$Ab = \sum_{k=1}^{n} b_k A_{*,k}.$$

Demostración, inicio

Queremos demostrar la fórmula

$$Ab = \sum_{k=1}^{n} b_k A_{*,k}.$$

Ambos lados de la fórmula son elementos de $\mathbb{R}^m$.

Demostración, inicio

Queremos demostrar la fórmula

$$Ab = \sum_{k=1}^{n} b_k A_{*,k}.$$

Ambos lados de la fórmula son elementos de $\mathbb{R}^m$.

Demostremos que sus j-ésimas componentes son iguales, para cualquier j en $\{1, \ldots, m\}$.

Demostración, inicio

Queremos demostrar la fórmula

$$Ab = \sum_{k=1}^{n} b_k A_{*,k}.$$

Ambos lados de la fórmula son elementos de $\mathbb{R}^m$.

Demostremos que sus j-ésimas componentes son iguales, para cualquier j en $\{1, \ldots, m\}$.

$$\left(\sum_{k=1}^{n} b_k A_{*,k} \right)_j$$

Demostración, inicio

Queremos demostrar la fórmula

$$Ab = \sum_{k=1}^{n} b_k A_{*,k}.$$

Ambos lados de la fórmula son elementos de $\mathbb{R}^m$.

Demostremos que sus j-ésimas componentes son iguales, para cualquier j en $\{1, \ldots, m\}$.

$$\left(\sum_{k=1}^{n} b_k A_{*,k} \right)_j \overset{(1)}{=\!=} \sum_{k=1}^{n} \left(b_k A_{*,k} \right)_j$$

Demostración, inicio

Queremos demostrar la fórmula

$$Ab = \sum_{k=1}^{n} b_k A_{*,k}.$$

Ambos lados de la fórmula son elementos de $\mathbb{R}^m$.

Demostremos que sus j-ésimas componentes son iguales, para cualquier j en $\{1, \ldots, m\}$.

$$\left(\sum_{k=1}^{n} b_k A_{*,k} \right)_j \overset{(1)}{=\!=} \sum_{k=1}^{n} \left(b_k A_{*,k} \right)_j \overset{(2)}{=\!=} \sum_{k=1}^{n} b_k A_{j,k}$$

Demostración, inicio

Queremos demostrar la fórmula

$$Ab = \sum_{k=1}^{n} b_k A_{*,k}.$$

Ambos lados de la fórmula son elementos de $\mathbb{R}^m$.

Demostremos que sus j-ésimas componentes son iguales, para cualquier j en $\{1, \ldots, m\}$.

$$\left(\sum_{k=1}^{n} b_k A_{*,k}\right)_j \overset{(1)}{=\!=} \sum_{k=1}^{n} \left(b_k A_{*,k}\right)_j \overset{(2)}{=\!=} \sum_{k=1}^{n} b_k A_{j,k} \overset{(3)}{=\!=} \sum_{k=1}^{n} A_{j,k} b_k$$

Demostración, inicio

Queremos demostrar la fórmula

$$Ab = \sum_{k=1}^{n} b_k A_{*,k}.$$

Ambos lados de la fórmula son elementos de $\mathbb{R}^m$.

Demostremos que sus j-ésimas componentes son iguales, para cualquier j en $\{1, \ldots, m\}$.

$$\left(\sum_{k=1}^{n} b_k A_{*,k}\right)_j \overset{(1)}{=\!=} \sum_{k=1}^{n} \left(b_k A_{*,k}\right)_j \overset{(2)}{=\!=} \sum_{k=1}^{n} b_k A_{j,k} \overset{(3)}{=\!=} \sum_{k=1}^{n} A_{j,k} b_k \overset{(4)}{=\!=} (Ab)_j.$$

Demostración, final

$$\left(\sum_{k=1}^{n} b_k A_{*,k}\right)_j \overset{(1)}{=\!=} \sum_{k=1}^{n} \left(b_k A_{*,k}\right)_j \overset{(2)}{=\!=} \sum_{k=1}^{n} b_k A_{j,k} \overset{(3)}{=\!=} \sum_{k=1}^{n} A_{j,k} b_k \overset{(4)}{=\!=} (Ab)_j.$$

Justificación.

Demostración, final

$$\left(\sum_{k=1}^{n} b_k A_{*,k}\right)_j \overset{(1)}{=\!=} \sum_{k=1}^{n} \left(b_k A_{*,k}\right)_j \overset{(2)}{=\!=} \sum_{k=1}^{n} b_k A_{j,k} \overset{(3)}{=\!=} \sum_{k=1}^{n} A_{j,k} b_k \overset{(4)}{=\!=} (Ab)_j.$$

Justificación.

1) La definición de la adición en $\mathbb{R}^n$.

Demostración, final

$$\left(\sum_{k=1}^{n} b_k A_{*,k}\right)_j \overset{(1)}{=\!=} \sum_{k=1}^{n} \left(b_k A_{*,k}\right)_j \overset{(2)}{=\!=} \sum_{k=1}^{n} b_k A_{j,k} \overset{(3)}{=\!=} \sum_{k=1}^{n} A_{j,k} b_k \overset{(4)}{=\!=} (Ab)_j.$$

Justificación.

1) La definición de la adición en $\mathbb{R}^n$.
2) La definicion de la multiplicación por escalares en $\mathbb{R}^n$.

Demostración, final

$$\left(\sum_{k=1}^{n} b_k A_{*,k}\right)_j \overset{(1)}{=\!=} \sum_{k=1}^{n} \left(b_k A_{*,k}\right)_j \overset{(2)}{=\!=} \sum_{k=1}^{n} b_k A_{j,k} \overset{(3)}{=\!=} \sum_{k=1}^{n} A_{j,k} b_k \overset{(4)}{=\!=} (Ab)_j.$$

Justificación.

1) La definición de la adición en $\mathbb{R}^n$.
2) La definicion de la multiplicación por escalares en $\mathbb{R}^n$.
3) La propiedad conmutativa de la multiplicación en $\mathbb{R}$.

Demostración, final

$$\left(\sum_{k=1}^{n} b_k A_{*,k}\right)_j \overset{(1)}{=\!=} \sum_{k=1}^{n} \left(b_k A_{*,k}\right)_j \overset{(2)}{=\!=} \sum_{k=1}^{n} b_k A_{j,k} \overset{(3)}{=\!=} \sum_{k=1}^{n} A_{j,k} b_k \overset{(4)}{=\!=} (Ab)_j.$$

Justificación.

1) La definición de la adición en $\mathbb{R}^n$.
2) La definicion de la multiplicación por escalares en $\mathbb{R}^n$.
3) La propiedad conmutativa de la multiplicación en $\mathbb{R}$.
4) La definición del producto de una matriz por un vector.

Función axpy (repaso)

Dados $a \in \mathbb{R}$, $x, y \in \mathbb{R}^n$, la función calcula y devuelve el vector $ax + y$.

```
from numpy import *

def axpy(a, x, y):
    q = len(x)
    z = zeros(q)
    for j in range(q):
        z[j] = a * x[j] + y[j]
    return z
```

Función axpy (repaso)

Dados $a \in \mathbb{R}$, $x, y \in \mathbb{R}^n$, la función calcula y devuelve el vector $ax + y$.

```
from numpy import *

def axpy(a, x, y):
    q = len(x)
    z = zeros(q)
    for j in range(q):
        z[j] = a * x[j] + y[j]
    return z
```

Decimos que esta función es de nivel 1 porque la complejidad es de orden $\Theta(q^1)$.

Idea del algoritmo para $n = 3$

$$Ab = b_1 A_{*,1} + b_2 A_{*,2} + b_3 A_{*,3}$$

Idea del algoritmo para $n = 3$

$$Ab = b_1 A_{*,1} + b_2 A_{*,2} + b_3 A_{*,3} = 0_m + b_1 A_{*,1} + b_2 A_{*,2} + b_3 A_{*,3}.$$

Idea del algoritmo para $n = 3$

$$Ab = b_1 A_{*,1} + b_2 A_{*,2} + b_3 A_{*,3} = \underbrace{0_m}_{c,\ \text{paso } 0} + b_1 A_{*,1} + b_2 A_{*,2} + b_3 A_{*,3}.$$

Vamos a guardar las sumas parciales en un vector c:

$c \coloneqq 0_m$.

Idea del algoritmo para $n = 3$

$$Ab = b_1 A_{*,1} + b_2 A_{*,2} + b_3 A_{*,3} = \underbrace{0_m + b_1 A_{*,1}}_{c,\ \text{paso } 1} + b_2 A_{*,2} + b_3 A_{*,3}.$$

Vamos a guardar las sumas parciales en un vector c:

$c := 0_m$.

Primer paso:

$c := c + b_1 A_{*,1}$.

Idea del algoritmo para $n = 3$

$$Ab = b_1 A_{*,1} + b_2 A_{*,2} + b_3 A_{*,3} = \underbrace{0_m + b_1 A_{*,1} + b_2 A_{*,2}}_{c,\ \text{paso 2}} + b_3 A_{*,3}.$$

Vamos a guardar las sumas parciales en un vector c:

$c := 0_m$.

Primer paso:

$c := c + b_1 A_{*,1}$.

Segundo paso:

$c := c + b_2 A_{*,2}$.

Idea del algoritmo para $n = 3$

$$Ab = b_1 A_{*,1} + b_2 A_{*,2} + b_3 A_{*,3} = \underbrace{0_m + b_1 A_{*,1} + b_2 A_{*,2} + b_3 A_{*,3}}_{c,\ \text{paso } 3}.$$

Vamos a guardar las sumas parciales en un vector c:

$c := 0_m$.

Primer paso:

$c := c + b_1 A_{*,1}$.

Segundo paso:

$c := c + b_2 A_{*,2}$.

Tercer paso:

$c := c + b_3 A_{*,3}$.

Programación (multiplicar una matriz por un vector con axpy)

```
def mulmatvec1c(A, b):
    (m, n) = A.shape
    c = zeros(m)
    for k in range(n):
        c = axpy(b[k], A[:, k], c)
    return c
```

Programación (multiplicar una matriz por un vector con axpy)

```
def mulmatvec1c(A, b):
    (m, n) = A.shape
    c = zeros(m)
    for k in range(n):
        c = axpy(b[k], A[:, k], c)
    return c
```

El número 1 en el título de la función significa que usamos otra función de nivel 1 (axpy).

Programación (multiplicar una matriz por un vector con axpy)

```
def mulmatvec1c(A, b):
    (m, n) = A.shape
    c = zeros(m)
    for k in range(n):
        c = axpy(b[k], A[:, k], c)
    return c
```

El número 1 en el título de la función significa que usamos otra función de nivel 1 (axpy).

La letra “c” en el título de la función significa que trabajamos con la matriz A por columnas.

Solución que utiliza las operaciones lineales de la librería

Ahora sustituimos nuestra función `axpy` por las operaciones lineales $+$ y $*$ del paquete `numpy`.

```
def mulmatvec1c_addmul(A, b):
    (m, n) = A.shape
    c = zeros(m)
    for k in range(n):
        c = b[k] * A[:, k] + c
    return c
```

También se puede escribir `c += b[k] * A[:, k]`.

Solución con dos ciclos anidados

En vez de llamar la función axpy, sustituimos su código (con ciertos cambios pequeños).

```
def mulmatvec0c(A, b):
    (m, n) = A.shape
    c = zeros(m)
    for k in range(n):
        for j in range(m):
            c[j] += A[j, k] * b[k]
    return c
```

El número 0 en el título de la función significa que no utilizamos ningunas funciones auxiliares; trabajamos solamente con operaciones de nivel 0.

Análisis de la complejidad

Todas las versiones del algoritmo `mulmatvec` tienen la misma complejidad computacional.

Es fácil ver que el número de las operaciones de multiplicación (de números reales) es

$$mn\,.$$

La complejidad total es de orden $\Theta(mn)$.

Análisis de complejidad

En particular, si $m = n$, entonces el número de las multiplicaciones es

$$n^2,$$

y el orden de complejidad es $\Theta(n^2)$.

En nuestro contexto (algoritmos con matrices) es cómodo decir que el algoritmo es de nivel 2.

Pruebas de la constistencia para datos pequeños

Calculamos Ab usando varias versiones del algoritmo.

```
def test0_mulmatvecc():
    A = array([[4, 0.5, -1], [2, 1, -1.5]])
    b = array([0.5, 2, 4])
    c0 = A @ b
    c1 = mulmatvec1c(A, b)
    c2 = mulmatvec1c_addmul(A, b)
    c3 = mulmatvec0c(A, b)
    return column_stack([c0, c1, c2, c3])

print(test0_mulmatvecc())
```

Pruebas de la consistencia para tamaños arbitrarios

```
def maxerrorvec(vecs):
    return max(linalg.norm(u - vecs[0]) for u in vecs)

def test1_mulmatvecc(m, n):
    A = random.rand(m, n); b = random.rand(n)
    c0 = A @ b
    c1 = mulmatvec1c(A, b)
    c2 = mulmatvec1c_addmul(A, b)
    c3 = mulmatvec0c(A, b)
    return maxerrorvec([c0, c1, c2, c3])

print(test1_mulmatvecc(5, 3))
```

Pruebas de velocidad

```
import time

def test2_mulmatvecc(n):
    A = random.rand(n, n); b = random.rand(n)
    t0 = time.time(); c0 = A @ b
    t1 = time.time(); c1 = mulmatvec1c(A, b)
    t2 = time.time(); c2 = mulmatvec1c_addmul(A, b)
    t3 = time.time(); c3 = mulmatvec0c(A, b)
    t4 = time.time()
    return [t1 - t0, t2 - t1, t3 - t2, t4 - t3]

print(test2_mulmatvecc(1024))
```

Resultados de las pruebas de velocidad

	$n = 1024$	$n = 2048$	$n = 4096$
`A @ b`	$4.8 \cdot 10^{-4}$	$2.2 \cdot 10^{-3}$	$8.3 \cdot 10^{-3}$
`mulmatvec1c`	0.32	1.5	6.0
`mulmatvec1c_addmul`	$3.5 \cdot 10^{-3}$	$3.1 \cdot 10^{-2}$	$1.2 \cdot 10^{-1}$
`mulmatvec0c`	0.43	1.9	7.7

Resultados de las pruebas de velocidad

	$n = 1024$	$n = 2048$	$n = 4096$
`A @ b`	$4.8 \cdot 10^{-4}$	$2.2 \cdot 10^{-3}$	$8.3 \cdot 10^{-3}$
`mulmatvec1c`	0.32	1.5	6.0
`mulmatvec1c_addmul`	$3.5 \cdot 10^{-3}$	$3.1 \cdot 10^{-2}$	$1.2 \cdot 10^{-1}$
`mulmatvec0c`	0.43	1.9	7.7

Conclusión 1: las funciones de la librería son mucho más rápidas que las nuestras, porque las librerías están escritas en lenguajes compilables (C, Fortran), mientras que Python es un lenguaje interpretado.

Conclusión 2: al multiplicar n por 2, el tiempo se multiplica por 4.

“Por renglones” versus “por columnas” (en Python+numpy)

En clases anteriores programamos la multiplicación de matrices por vectores “por renglones”. Comparamos la velocidad con las versiones “por columnas”.

	$n = 1024$	$n = 2048$	$n = 4096$
`mulmatvec1r`	0.29	1.3	4.5
`mulmatvec1r_dot`	$1.8 \cdot 10^{-3}$	$5.9 \cdot 10^{-3}$	$2.0 \cdot 10^{-2}$
`mulmatvec0r`	0.43	1.7	6.7
`mulmatvec1c`	0.32	1.5	6.0
`mulmatvec1c_addmul`	$3.5 \cdot 10^{-3}$	$3.1 \cdot 10^{-2}$	$1.2 \cdot 10^{-1}$
`mulmatvec0c`	0.43	1.9	7.7