

La matriz-quadtrees y el algoritmo de Strassen

Especificación algebraica en Nereus — material para alumnos

1. El problema

El **algoritmo de Strassen** multiplica dos matrices de $2^k \times 2^k$ aplicando *Divide y Conquista*: parte cada matriz en cuatro cuadrantes y, en lugar de los **8** productos de submatrices del método de bloques ingenuo, usa solo **7** (a cambio de más sumas y restas). Eso baja la complejidad de $O(n^3)$ a $O(n^{2.807})$.

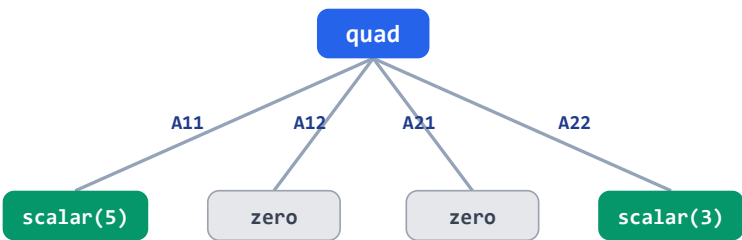
Como el problema es recursivo, conviene una **representación de datos recursiva**. Si además la matriz es **rala** (muchos ceros), queremos una estructura que ni siquiera almacene los bloques nulos: la **matriz-quadtrees**.

2. La matriz-quadtrees: tres tipos de nodo

Una matriz cuadrada se subdivide recursivamente en sus cuatro cuadrantes. Cada nodo del árbol es de uno de tres tipos — las tres *claves* del caso de estudio:

Clave	Constructor Nereus	Significado
= 0	<code>zero</code>	región nula : submatriz toda de ceros
< 0	<code>scalar(v)</code>	región homogénea : bloque base 1×1 con el valor <code>v</code>
> 0	<code>quad(A11,A12,A21,A22)</code>	región no homogénea : nodo con los 4 cuadrantes

Por ejemplo, la matriz rala `[[5,0],[0,3]]` se representa así — los dos ceros de la diagonal secundaria son un único nodo `zero` cada uno, sin almacenar nada:



`[[5,0],[0,3]]` como `quad(scalar(5), zero(), zero(), scalar(3))`.

Constructores y observadores en Nereus

BASIC CONSTRUCTORS zero, scalar, quad

```
zero:    -> MatrixQT;           -- región nula
scalar: Integer -> MatrixQT;    -- bloque base 1x1
quad:    MatrixQT * MatrixQT * MatrixQT * MatrixQT -> MatrixQT;
```

Los observadores responden preguntas sobre la estructura: `isZero`, `isScalar`, `isPartitioned`; `scalarValue` (valor de un bloque base); `nw/ne/sw/se` (los cuatro cuadrantes); y `numNonZero` (cuántos elementos no nulos tiene la matriz).

3. Suma y resta

`addQ` y `subQ` recorren los dos árboles en paralelo. La clave está en cómo tratan al `zero`:

```
addQ(zero(), m) = m;
addQ(m, zero()) = m;
addQ(scalar(x), scalar(y)) = scalar(x + y);
addQ(quad(a11,a12,a21,a22), quad(b11,b12,b21,b22)) =
    quad(addQ(a11,b11), addQ(a12,b12), addQ(a21,b21), addQ(a22,b22));
```

El cero poda la operación. El axioma `addQ(zero(), m) = m` devuelve `m` *sin recorrerlo*: sumar una región nula no cuesta nada. En una matriz rala, eso evita visitar ramas enteras del árbol. La resta es análoga, con un detalle: `subQ(zero(), m)` recorre `m` **negándolo** ($0 - M = -M$), sin necesidad de una operación de negación aparte.

4. El producto de Strassen

`strassen(A, B)` calcula $A \times B$ recursivamente, con tres casos base y el caso recursivo:

```
strassen(zero(), m) = zero();           -- 0 * M = 0 (poda la rama)
strassen(m, zero()) = zero();           -- M * 0 = 0 (poda la rama)
strassen(scalar(x), scalar(y)) = scalar(x * y); -- caso base: producto escalar
strassen(quad(a11,a12,a21,a22), quad(b11,b12,b21,b22)) =
  LET
    OPERATION p1: -> MatrixQT; ... p7: -> MatrixQT;
    AXIOMS
      p1() = strassen(addQ(a11,a22), addQ(b11,b22));
      p2() = strassen(addQ(a21,a22), b11);
      p3() = strassen(a11, subQ(b12,b22));
      p4() = strassen(a22, subQ(b21,b11));
      p5() = strassen(addQ(a11,a12), b22);
      p6() = strassen(subQ(a21,a11), addQ(b11,b12));
      p7() = strassen(subQ(a12,a22), addQ(b21,b22));
  IN
    quad( addQ(subQ(addQ(p1(),p4()),p5()),p7()), -- C11 = p1+p4-p5+p7
          addQ(p3(),p5()),                     -- C12 = p3+p5
          addQ(p2(),p4()),                     -- C21 = p2+p4
          addQ(subQ(addQ(p1(),p3()),p2()),p6()) ) -- C22 = p1+p3-p2+p6
  END-LET;
```

¿Por qué el LET? Los siete productos `p1..p7` se usan más de una vez al armar los cuatro cuadrantes (`p1` aparece en C11 y C22, etc.). El `LET` los liga **una sola vez**. Sin él, el axioma recalcularía productos y se harían más de 7 multiplicaciones recursivas por nivel — se perdería exactamente la ventaja que define al algoritmo de Strassen.

5. Un ejemplo trabajado

Multipliquemos la identidad por una matriz diagonal (el ejemplo del caso de estudio):

```
A = [[1,0],[0,1]] = quad(scalar(1), zero(), zero(), scalar(1))
B = [[2,0],[0,2]] = quad(scalar(2), zero(), zero(), scalar(2))
```

Los cuatro cuadrantes de cada una: `a11=scalar(1)`, `a12=a21=zero()`, `a22=scalar(1)`; e igual para B con el valor 2. Los siete productos:

Producto	Expresión	Se reduce a	Valor
p1	strassen(a11+a22, b11+b22)	strassen(scalar(2), scalar(4))	scalar(8)
p2	strassen(a21+a22, b11)	strassen(scalar(1), scalar(2))	scalar(2)
p3	strassen(a11, b12-b22)	strassen(scalar(1), scalar(-2))	scalar(-2)
p4	strassen(a22, b21-b11)	strassen(scalar(1), scalar(-2))	scalar(-2)
p5	strassen(a11+a12, b22)	strassen(scalar(1), scalar(2))	scalar(2)
p6	strassen(a21-a11, b11+b12)	strassen(scalar(-1), scalar(2))	scalar(-2)
p7	strassen(a12-a22, b21+b22)	strassen(scalar(-1), scalar(2))	scalar(-2)

Y los cuatro cuadrantes del resultado:

Cuadrante	Fórmula	Cuenta	Valor
C11	p1 + p4 - p5 + p7	8 - 2 - 2 - 2	scalar(2)
C12	p3 + p5	-2 + 2	scalar(0)
C21	p2 + p4	2 - 2	scalar(0)
C22	p1 + p3 - p2 + p6	8 - 2 - 2 - 2	scalar(2)

El resultado es `quad(scalar(2), scalar(0), scalar(0), scalar(2))`, es decir `[[2,0],[0,2]]` — correcto: identidad por diagonal da la diagonal.

Observá que las posiciones nulas quedaron como `scalar(0)`, no como `zero()`. El resultado es *correcto* pero no en su forma **rala canónica**: la aritmética produjo un cero "explícito". La operación de *normalización* de la sección 6 reemplaza esos `scalar(0)` por `zero()` para mantener la matriz lo más rala posible.

6. Normalización rala

El ejemplo anterior dejó `scalar(0)` donde, idealmente, debería haber `zero()`. Si se encadenan varios productos sin corregir esto, la matriz se va **densificando** y se pierde el beneficio de la representación rala. Dos operaciones lo resuelven:

- `isNormalized(m)` — ¿está `m` en su forma **rala canónica**? (sin `scalar(0)`, y sin nodos cuyos cuatro cuadrantes sean nulos);
- `normalize(m)` — devuelve esa forma canónica.

```
normalize(zero())    = zero();
normalize(scalar(v)) = if v = 0 then zero() else scalar(v) endif;
normalize(quad(a11,a12,a21,a22)) =
  joinQ(normalize(a11), normalize(a12), normalize(a21), normalize(a22));
```

`normalize` trabaja **de abajo hacia arriba**: primero normaliza los cuatro cuadrantes y después usa el auxiliar `joinQ`, que colapsa un nodo en `zero()` si sus cuatro cuadrantes ya normalizados resultaron nulos. Sobre el resultado del ejemplo anterior:

```
normalize( quad(scalar(2), scalar(0), scalar(0), scalar(2)) )
      = quad(scalar(2), zero(),    zero(),    scalar(2))
```

Por qué importa. La poda de Strassen (`strassen(zero, M) = zero`) solo funciona si los ceros están representados como `zero()` . Un `scalar(0)` "engaña" a la estructura: ocupa un nodo y no poda nada. Normalizar después de operar mantiene la matriz rala y conserva la eficiencia a lo largo de una cadena de cálculos.

7. Un ejemplo de 4×4

Una matriz de 4×4 es un `quad` cuyos cuatro cuadrantes son, a su vez, `quad`s de 2×2. Multipliquemos dos matrices ralas:

```
A = [ 1 2 0 0 ]      B = [ 5 0 0 0 ]
    [ 3 4 0 0 ]      [ 0 6 0 0 ]
    [ 0 0 0 0 ]      [ 0 0 0 0 ]
    [ 0 0 0 0 ]      [ 0 0 0 0 ]

A = quad( quad(scalar(1),scalar(2),scalar(3),scalar(4)), zero(), zero(), zero() )
B = quad( quad(scalar(5),zero(),zero(),scalar(6)),      zero(), zero(), zero() )
```

Nivel 1 — los 7 productos del tope

Con `a11 = A11`, `a12 = a21 = a22 = zero` y `b11 = B11`, `b12 = b21 = b22 = zero`:

Producto	Se reduce a	Resultado
<code>p1 = strassen(a11+a22, b11+b22)</code>	<code>strassen(A11, B11)</code>	recursión ↓
<code>p2 = strassen(a21+a22, b11)</code>	<code>strassen(zero, B11)</code>	zero (podado)
<code>p3 = strassen(a11, b12-b22)</code>	<code>strassen(A11, zero)</code>	zero (podado)
<code>p4 = strassen(a22, b21-b11)</code>	<code>strassen(zero, ...)</code>	zero (podado)
<code>p5 = strassen(a11+a12, b22)</code>	<code>strassen(A11, zero)</code>	zero (podado)
<code>p6 = strassen(a21-a11, b11+b12)</code>	<code>strassen(-A11, B11)</code>	recursión ↓
<code>p7 = strassen(a12-a22, b21+b22)</code>	<code>strassen(zero, zero)</code>	zero (podado)

De los 7 productos del nivel superior, **5 se resuelven sin recursión** gracias a `strassen(zero, M) = zero` y `strassen(M, zero) = zero`. Solo `p1` y `p6` bajan un nivel. Ésa es la poda de la representación rala en acción.

Nivel 2 — la recursión `p1 = strassen(A11, B11)`

Con `A11 = [[1,2],[3,4]]` y `B11 = [[5,0],[0,6]]`, los siete sub-productos y los cuatro cuadrantes:

q1	q2	q3	q4	q5	q6	q7
55	35	-6	-20	18	10	-12

Cuadrante	Fórmula	Cuenta	Valor
C11	$q_1 + q_4 - q_5 + q_7$	$55 - 20 - 18 - 12$	5
C12	$q_3 + q_5$	$-6 + 18$	12
C21	$q_2 + q_4$	$35 - 20$	15
C22	$q_1 + q_3 - q_2 + q_6$	$55 - 6 - 35 + 10$	24

Entonces `p1 = quad(scalar(5), scalar(12), scalar(15), scalar(24)) = [[5,12],[15,24]]`. Y, por linealidad, `p6 = strassen(-A11, B11) = -(A11·B11) = quad(scalar(-5), scalar(-12), scalar(-15), scalar(-24))`.

Nivel 1 — combinación final

Con `p2 = p3 = p4 = p5 = p7 = zero`, las cuatro fórmulas se simplifican:

Cuadrante	Fórmula	Queda
C11	$p_1 + p_4 - p_5 + p_7$	$p_1 = [[5,12],[15,24]]$
C12	$p_3 + p_5$	zero
C21	$p_2 + p_4$	zero
C22	$p_1 + p_3 - p_2 + p_6$	<code>addQ(p1, p6)</code>

`C22 = addQ(p1, p6) = p1 + (-p1)` se calcula cuadrante a cuadrante y da `quad(scalar(0), scalar(0), scalar(0), scalar(0))` — ceros **explícitos**: la matriz se densificó. Ahí entra `normalize`:

```
resultado = quad( quad(scalar(5), scalar(12), scalar(15), scalar(24)),
                  zero(), zero(),
                  quad(scalar(0), scalar(0), scalar(0), scalar(0)) )

normalize(resultado)
= quad( quad(scalar(5), scalar(12), scalar(15), scalar(24)),
        zero(), zero(), zero() )
```

que es la matriz producto, correcta y en forma rala canónica:

```
[ 5 12 0 0 ]
[15 24 0 0 ]
[ 0  0 0 0 ]
[ 0  0 0 0 ]
```

Este ejemplo recorre las tres ideas juntas: la **poda** (5 de los 7 productos del tope se resuelven sin recursión), la **recursión** de Strassen (`p1` baja un nivel y aplica de nuevo los 7 productos), y la **normalización**, que recupera la forma rala después de que la aritmética introdujo ceros explícitos.

8. Por qué quadtree + Strassen para matrices ralas

El árbol de recursión de Strassen tiene factor de ramificación **7**: para matrices $2^k \times 2^k$ hace 7^k productos escalares, frente a 8^k del método de bloques ingenuo. Ahí está el $O(n^{2.807})$.

La matriz-quadtree agrega una segunda poda, *ortogonal* a la anterior: los axiomas `strassen(zero(), m) = zero()` y `strassen(m, zero()) = zero()` cortan una rama recursiva entera en un solo paso. En una matriz rala — donde abundan los bloques nulos — muchísimos de los 7 subproductos de cada nivel se resuelven al instante sin recursión. La estructura recursiva de los datos acompaña a la estructura recursiva del algoritmo.

9. Para practicar

1. Escribí como `MatrixQT` la matriz `[[0,7],[0,0]]` y calculá `numNonZero`.
2. Calculá `strassen(zero(), B)` para cualquier `B`: ¿cuántas multiplicaciones escalares hace? ¿Por qué?
3. En el ejemplo de la sección 5, ¿cuál de los 7 productos se podría "podar" si `A` tuviera `a22 = zero()`? Seguí los axiomas.
4. Verificá a mano que `addQ` es conmutativa sobre el ejemplo: `addQ(A,B) = addQ(B,A)`.
5. ¿Por qué `strassen` no tiene axioma para `strassen(scalar(x), quad(...))`? ¿Qué tendría de malo esa multiplicación?