

# Ising Model

## MVMISG

### Métodos Numéricos Avanzados

Enzo Altamiranda, Mauricio Minestrelli, Cristian Ontivero, Valeria Serber

3 de octubre de 2014

#### Resumen

Se presentan los resultados que se obtuvieron al implementar diferentes estructuras de datos y algoritmos en C para, dado a partir de un parámetro que define el tamaño de una matriz con orígenes en el modelo de Ising, generarla. Posteriormente, para evitar los detalles de C se mueve a Octave, y se muestra un método para calcular los autovalores de la misma.

**Palabras clave:** Matrices ralas, Matrices dispersas, Compressed Column storage, Compressed sparse column, Modelo de Ising

## 1. Introducción

El *Modelo de Ising* es un modelo físico utilizado en el estudio del comportamiento de materiales ferromagnéticos, propuesto por el físico *Wilhelm Lenz* en 1920 a su estudiante *Ernst Ising*, quién demostró que el modelo unidimensional no tenía transición de fase para su tesis en 1924. La matriz tratada en el presente trabajo tiene sus orígenes en este modelo, y surge de un producto de matrices que se explicará más adelante.

En este informe se describen las distintas instancias sucedidas en el desarrollo de un programa que calcula la matriz del *Modelo de Ising*, junto con sus autovalores. Se mencionan los algoritmos elegidos y cómo se fue optimizando los mismos, así como también las estructuras utilizadas, con el objetivo de lograr una menor complejidad en el código y una mayor rapidez para generar resultados. También se muestran las pruebas realizadas y gráficos que comparan las distintas iteraciones del desarrollo.

## 2. Metodología

### 2.1. Cálculo de la matriz del Modelo de Ising

La matriz del Modelo de Ising, llamada  $\mathbf{A}$  a partir de ahora, puede calcularse como el producto de dos matrices  $\mathbf{K}$  y  $\mathbf{L}$ , ambas de dimensión  $2m \times 2m$ , con  $m$  perteneciente a los naturales.

$$\mathbf{A} = \mathbf{KL} \quad (1)$$

donde

$$\mathbf{K} = \begin{pmatrix} E & & & \\ & E & & \\ & & \ddots & \\ & & & E \\ & & & & E \end{pmatrix}, E = \begin{pmatrix} \cos \alpha & \sin \alpha \\ -\sin \alpha & \cos \alpha \end{pmatrix} \quad (2)$$

$$\mathbf{L} = \begin{pmatrix} \cos \beta & & & -\sin \beta \\ & F & & \\ & & \ddots & \\ & & & F \\ \sin \beta & & & & \cos \beta \end{pmatrix}, F = \begin{pmatrix} \cos \beta & \sin \beta \\ -\sin \beta & \cos \beta \end{pmatrix} \quad (3)$$

#### 2.1.1. Primera Iteración

En la primera iteración se optó por almacenar cada matriz en un arreglo de arreglos. Se comenzó realizando una implementación del algoritmo estándar de multiplicación de matrices, debido a la simpleza del código correspondiente. Se probó esta versión creando las matrices  $\mathbf{K}$  y  $\mathbf{L}$  y multiplicándolas. El orden temporal de este algoritmo es cúbico, es decir,  $O(n^3)$ . Debido a esto, al calcular matrices pequeñas el algoritmo termina de forma rápida, sin embargo, al probar matrices de mayor tamaño, el tiempo crece en gran medida.

#### 2.1.2. Segunda Iteración

Una vez visto los límites de la implementación densa, se optó por aprovechar el hecho de que las matrices necesarias para la construcción de la matriz  $\mathbf{A}$ , incluida esta última, eran todas ralas. Para optimizar el algoritmo, se decidió por cambiar la estructura de datos que almacena las matrices, de modo que los ceros no se almacenen, y el algoritmo de multiplicación pueda los productos de valores nulos.

La estructura utilizada en esta iteración se conoce como “*Compressed column storage*” (*CCS*), o alternativamente “*Compressed sparse column*”, y es la representación tradicional utilizada en *MATLAB* al usar la función “*sparse*”.

Esta representación cuenta con tres arreglos. El primero, al cual llamaremos **values**, contiene los valores no nulos de la matriz, de tamaño **nnz** (del inglés “*number of nonzeros*”). El segundo, **ri**, indica el índice de la fila del elemento que se encuentra en el primer arreglo y en la misma posición. Dicho arreglo también tiene tamaño **nnz**. El tercero, **cp**, tiene como tamaño la cantidad de columnas más uno, donde en la posición  $j$  del arreglo se guarda el índice en el arreglo de valores, en el que se encuentra el primer elemento no nulo de la columna  $j$ . El último elemento de **cp** es el valor **nnz**. Si alguna columna tuviera todos ceros, en el índice de esa columna se coloca el mismo valor que en la próxima columna.

### Ejemplo de almacenamiento de una matriz utilizando Compressed column storage

$$M = \begin{pmatrix} 0 & 3 & 0 & 5 & 7 \\ 0 & 0 & 0 & 3 & 8 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 5 \end{pmatrix}$$

Notar que la tercer columna no tiene valores. Esto se representa usando el mismo valor en  $cp$  que en la próxima columna. Esto es consistente con el hecho de que la cantidad de elementos no nulos presentes en la columna  $j$  se puede conocer con la resta

$$cp[j+1] - cp[j].$$

$$values = [ 1, 3, 2, 5, 3, 7, 8, 5 ]$$

$$ri = [ 2, 0, 3, 0, 1, 0, 1, 3 ]$$

$$cp = [ 0, 1, 3, 3, 5, 8 ]$$

### Complejidad espacial de la representación CCS

Para una **matriz densa**, es decir, que no tiene elementos nulos, de  $m \times n$ , se necesitan  $m$  punteros en memoria, cada uno apuntando a los  $m$  arreglos de  $n$  elementos. En comparación con una matriz en representación *CCS*, se necesita espacio para  $nnz \cdot \text{sizeof}(\text{elementos}) + (nnz + cols) \cdot \text{sizeof}(\text{índices})$ . Para una aproximación más fácil de entender, si suponemos que el tamaño de índices, elementos, y punteros en memoria usan todos a misma cantidad de bytes, se tiene que la complejidad espacial de la matriz densa es  $m+m \cdot n$ , es decir,  $O(m \cdot n)$ . Como en el caso del Modelo de Ising las matrices son cuadradas, el orden espacial resulta  $O(n^2)$ . En cambio, la complejidad espacial de la **matriz rara** es  $2 \cdot nnz + n$ , es decir orden  $O(nnz + n)$ , el cual es *lineal*.

### Algoritmo de multiplicación CCS

Al utilizar la representación *CCS*, se logró un algoritmo de multiplicación significativamente más eficiente (como se verá a continuación), ideal para matrices ralas, ya que bastó con multiplicar y sumar solamente los valores no nulos de las matrices, ignorando los ceros. A continuación se observa en pseudocódigo, el algoritmo implementado para multiplicar utilizando la estructura *CCS*, que difiere del algoritmo estándar utilizado anteriormente. Cabe destacar que tanto este algoritmo como la representación *CCS* de matrices, requieren que las matrices sean efectivamente dispersas para presentar ventajas ante el algoritmo y estructura tradicional de matrices densas.

---

```

CCSMatrix matriz = nueva matriz de m x n;

Por cada columna de la 2da matriz {
    Si la columna no tiene elementos entonces saltarla;

    Por cada fila de la 1ra matriz {

        currentVal = suma de los productos de los valores no nulos de la
            columna actual, con los correspondientes de la fila actual;

        Si currentVal es cero entonces saltar la fila;

        Si aun no se registro el valor que comienza la columna actual {

            Registrar;

            Si habia columnas saltadas por no tener elementos entonces
                propagar el valor actual hacia atras;
        }

        Almacenar el valor no nulo actual;

        Almacenar el indice de la fila del valor no nulo actual;
    }
}

Almacenar el numero de valores no nulos en la matriz;

Almacenar el numero de valores no nulos en la posicion final del arreglo de
    punteros de columnas;

Retornar matriz;

```

---

*Algoritmo de multiplicación CCS: Se muestra en pseudocódigo el algoritmo que calcula la multiplicación de las matrices  $K$  y  $L$  aprovechando la estructura de datos CCS.*

### 3. Resultados

#### 3.1. Comparación entre ambas iteraciones

Se realizaron pruebas en ambas iteraciones para descubrir las diferencias entre ellas. Se utilizaron matrices de diferentes tamaños, y se utilizó el comando “*time*” de *Linux* para medir el tiempo en cada caso. Para consistencia y coherencia en los resultados, todas las pruebas se hicieron en una misma computadora, de procesador Intel Core i7 3612QM 2.1 GHz y 8 GB DDR3 de memoria RAM. A continuación se muestra una tabla con los valores que se utilizaron en las pruebas, y luego los gráficos que se realizaron a partir de ellas.

Multiplicación estándar		Multiplicación CCS	
m	tiempo (seg)	m	tiempo (seg)
100	0.02	100	0
150	0.05	150	0
200	0.13	200	0
250	0.25	250	0.01
300	1.32	300	0.01
350	2.88	350	0.01
400	4.24	400	0.01
450	6.51	450	0.02
500	8.90	500	0.02
550	12.05	550	0.02
600	15.81	600	0.03
650	20.22	650	0.03
700	25.25	700	0.04
750	32.36	750	0.04
800	36.33	800	0.05
850	45.21	850	0.06
900	46.49	900	0.06
950	54.95	950	0.06
1000	62.82	1000	0.07
		10000	3.19
		20000	7.14
		25000	19.59
		30000	28.22
		35000	39.30
		40000	52.50
		45000	63.79

Tabla 1: Muestra el tiempo en segundos, que tardaron los dos algoritmos implementados respectivamente, a partir de distintos valores de  $m$ .

##### 3.1.1. Primera iteración

En el siguiente gráfico se representa la complejidad del algoritmo de multiplicación estándar de matrices. El eje  $x$  corresponde al valor de  $m$ , y el eje  $y$  al *tiempo* en segundos, que tarda el algoritmo en resolver.

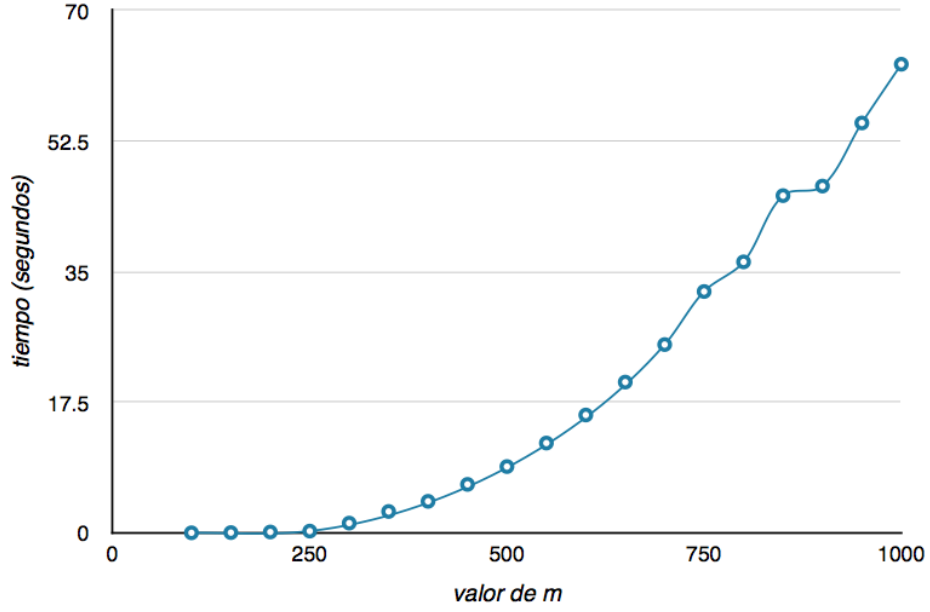


Figura 1: Multiplicación estándar de matrices

### 3.1.2. Segunda iteración

En el siguiente gráfico se representa la complejidad del algoritmo de multiplicación de matrices utilizando la estructura de datos *CCS*. Los ejes representan lo mismo que el gráfico anterior.

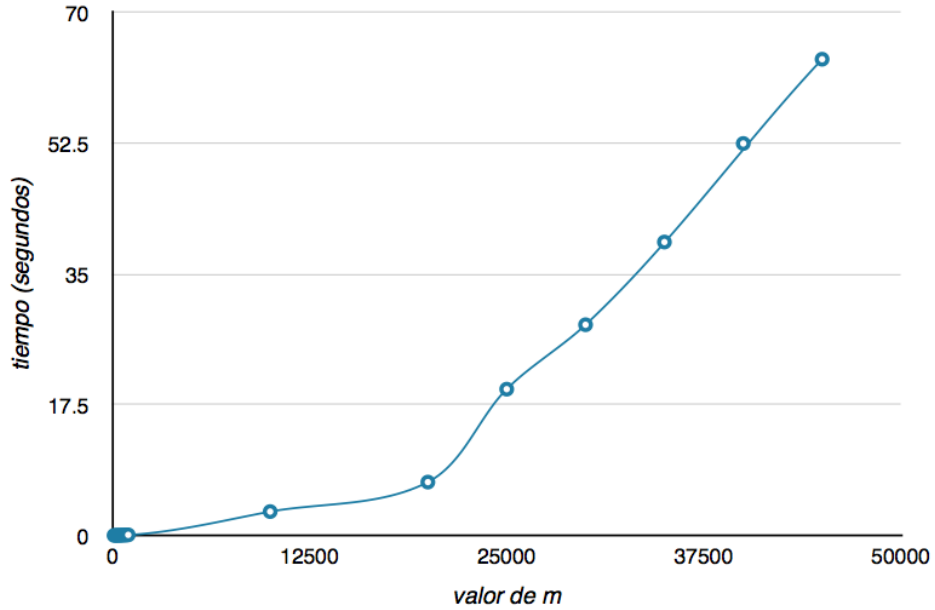


Figura 2: Multiplicación CCS

## 3.2. Descripción de los resultados

El resultado más importante que se puede obtener de los gráficos anteriores es que en el intervalo de hasta 70 segundos, el primer algoritmo puede resolver hasta matrices de  $m = 1000$ , mientras que el segundo puede resolver hasta matrices de  $m = 50000$  aproximadamente. Además, se puede observar que la curva del primer algoritmo crece mucho más rápido que la segunda. Esto puede corresponderse con el hecho de que el primer algoritmo es de  $O(n^3)$ , mientras que el otro es de orden menor.

### 3.2.1. Tercera iteración

Si bien se logró optimizar bastante el algoritmo que calcula la matriz  $A$ , mientras se realizaban pruebas se pudo observar un patrón en la construcción de  $A$ . Se notó que a partir de  $m = 3$ , la matriz  $A$  puede escribirse genéricamente, ya que pequeños bloques de elementos se repiten a lo largo de la estructura, aumentando la cantidad de bloques predeciblemente a medida que  $m$  aumenta. Por lo tanto, se puede evitar tener que calcular  $A$  a partir del producto entre  $K$  y  $L$ , y en vez de eso, seguir la siguiente regla para representarla:

Cuando  $m = 1$ ,

$$A = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \quad (4)$$

Cuando  $m = 2$ ,

$$A = \begin{pmatrix} T & Q & R & S \\ R & S & T & Q \end{pmatrix} \quad (5)$$

Cuando  $m = 3$ , la matriz comienza a expandirse dejando ceros en varias posiciones. Además puede verse que el bloque del medio compuesto por  $Q$ ,  $R$ ,  $S$  y  $T$  comienza a repetirse.

$$A = \begin{pmatrix} T & Q & R & & & S \\ & S & T & Q & R & \\ & & R & & S & T & Q \end{pmatrix} \quad (6)$$

Entonces, para  $m \geq 3$ , la matriz puede escribirse de forma genérica de la siguiente forma:

$$A = \begin{pmatrix} T & Q & R & & & & & & S \\ & S & T & Q & R & & & & \\ & & S & T & \ddots & & & & \\ & & & & \ddots & & & & \\ & & & & & Q & R & & \\ & & & & & S & T & Q & R & Q \\ R & & & & & & & S & T & Q \end{pmatrix} \quad (7)$$

Siendo

$$Q = \begin{pmatrix} \sin \alpha \cos \beta \\ \cos \alpha \cos \beta \end{pmatrix} \quad (8)$$

$$R = \begin{pmatrix} \sin \alpha \sin \beta \\ \cos \alpha \sin \beta \end{pmatrix} \quad (9)$$

$$S = \begin{pmatrix} -\cos \alpha \sin \beta \\ \sin \alpha \sin \beta \end{pmatrix} \quad (10)$$

$$T = \begin{pmatrix} \cos \alpha \cos \beta \\ -\sin \alpha \cos \beta \end{pmatrix} \quad (11)$$

Utilizando esta forma de construcción, se logra generar la matriz  $A$  de  $m = 100000000$  (cien millones) en al rededor de 10 segundos. Antes de comenzar a tardar por encima del minuto, la computadora se queda sin memoria RAM (por ejemplo, al probar con 300 millones, ya el sistema operativo retornó NULL al llamar a *malloc* para reservar la memoria necesaria).

## 4. Búsqueda de autovalores

Para calcular los autovalores de la matriz se utilizó el procedimiento recomendado en “Matrix Market”. En el mismo, se obtienen los autovalores de la matriz de  $2m \times 2m$ ,  $A = K \cdot L$ , calculando los autovalores de las  $m$  matrices de  $2 \times 2$ .

Mediante la función `compareEigenvalueMethods` se puede comprobar que los  $2m$  autovalores de las  $m$  matrices de  $2 \times 2$  corresponden a una buena aproximación de los autovalores de la matriz  $A$ .

Para obtener los autovalores de las matrices de  $2 \times 2$  se resuelve el polinomio característico a partir de la fórmula cuadrática:

$$\lambda_{1,2} = \frac{\text{tr}(A) \pm \sqrt{\text{tr}(A)^2 - 4 \cdot \det(A)}}{2} \quad (12)$$

Como comparación, se obtuvieron los autovalores de la matriz  $A$  a partir de la función de octave `eig`, y se comprobó que nuestros resultados eran correctos.

Utilizando la función `getEigenvalues`, se calculan los autovalores. Debido a que sólo se deben crear y multiplicar matrices de  $2 \times 2$ , para luego obtener los autovalores a partir de un método algebraico, se pueden calcular un gran número de autovalores en tiempos razonable, como se puede apreciar en la siguiente tabla:

Multiplicación estándar	
<b>m</b>	<b>tiempo (seg)</b>
10	0.006
100	0.050
1000	0.304
10000	2.541
100000	39.905
1000000	4400.470

Tabla 2: Muestra el tiempo en segundos, que tardó octave al usar distintos valores de  $m$  en la función `getEigenvalues`

En un principio se intentó utilizar el algoritmo  $QR$  para obtener los autovalores de  $A$ . Este acercamiento no dió resultados, ya que se notó que la matriz no convergía a una forma triangular. En un principio se pensó que esto se debía a la existencia de autovalores complejos, pero luego se descubrió que la razón era que la matriz  $A$  es ortogonal, por lo tanto al descomponer en matrices  $QR$ , resulta que  $Q$  equivale a  $A$ , y  $R$  a la matriz identidad. Por lo tanto, se optó por calcular los autovalores de las matrices de  $2 \times 2$ .

La matriz  $A$  que se obtiene de multiplicar  $K$  por  $L$  es rala, especialmente para valores grandes de  $m$ . Debido a esto, el algoritmo  $QR$ , de haber funcionado, habría sido muy ineficiente para calcular los autovalores. En un principio, se pensó en utilizar el Método de Arnoldi. Este procedimiento iterativo para matrices ralas, va generando a partir de  $A$  una matriz de Hessenberg. Una fracción de los autovalores de esta matriz son una buena aproximación a los autovalores de  $A$ . Adicionalmente, el algoritmo  $QR$  corre mucho más eficientemente sobre matrices de Hessenberg, por lo tanto, el Método de Arnoldi, da la posibilidad de conseguir un conjunto de autovalores de  $A$  de manera eficiente.

## 5. Conclusiones

La matriz  $A$  del modelo de Ising representa un buen ejemplo de la importancia de la correcta elección de una estructura de datos adecuada, y algoritmos eficientes a la hora de resolver un problema.

No solo esto, sino cómo, probando propiedades o características del problema en cuestión matemáticamente, es posible evitar cálculos redundantes y optimizar aun más un programa, dando tres diferentes



aristas de gran relevancia por las cuales uno puede aproximarse a una solución óptima. Esto también se pudo apreciar en el cálculo de los autovalores, donde el algoritmo final utilizado ni siquiera necesitó de la construcción de  $A$  para poder calcular sus autovalores.

## 6. Bibliografía

Watkins, David S., Fundamentals of Matrix Computations. New York: John Wiley & Sons, Inc., 2002

Yousef, Saad, Numerical Methods For Large Eigenvalue Problems. Society for Industrial and Applied Mathematics, 2011

Gene H. Golub & Charles F. Van Loan, Matrix computations, The Johns Hopkins University Press, Baltimore, 1996

[http://en.wikipedia.org/wiki/Sparse\\_matrix](http://en.wikipedia.org/wiki/Sparse_matrix)

[http://netlib.org/linalg/html\\_templates/node91.html#SECTION00931100000000000000](http://netlib.org/linalg/html_templates/node91.html#SECTION00931100000000000000)

[http://netlib.org/linalg/html\\_templates/node92.html#SECTION00931200000000000000](http://netlib.org/linalg/html_templates/node92.html#SECTION00931200000000000000)