Universidad de Colima

Programación y métodos computacionales

Eigenvalue Computing Side-by-Side

Autor:
Yair Antonio (Castillo)[2]

Profesor:
Roberto A. Saenz Casas

15 de noviembre de 2016

Índice

1. Introducción

La computación en paralelo es una forma de cálculo donde se realizan las operaciones de la computadora en forma simultánea, en lugar de secuencialmente.

La mayoría de los algoritmos de álgebra lineal de tipo numérico estándar se han hecho en paralelo. Pero nos enforcaremos en el algoritmo de Jacobi de autovalores para matrices simétricas, cuyo propósito es encontrar los valores propios de cualquier matriz simétrica $n \times n$.

En general, las matrices simétricas de 2×2 siempre tendrán valores propios reales por el Teorema Espectral, los cuales se pueden encontrar mediante el uso de la ecuación de segundo grado. Las matrices simétricas grandes no tienen ningún tipo de ecuación explícita para encontrar sus autovalores; así que el algoritmo de Jacobi fue pensado como un conjunto de pasos iterativos para encontrar los autovalores de cualquier matriz simétrica.

Se debe de empezar con una matriz simétrica en irla diagonalizando poco a poco con las iteraciones. De forma particular, si $A \in \mathbf{R}^{n \times n}$ es simétrica, entonces queremos que se reduzca

$$\text{Offdiags(A)} = \sqrt{\sum_{i=1}^{n} \sum_{j=1, j \neq i}^{n} a_{ij}^2} \tag{1}$$

que es una medida del "tamaño"de las entradas fuera de la diagonal de A. La cantidad en (1) se reduce con iteraciones a través de resolver subproblemas 2×2 hechos de A. Para hacer lo que se dijo anteriormente, en el algoritmo de Jacobi de autovalores para matrices simétricas, elegimos una entrada de la matriz (p, q), donde $0 < p, q \leq n$ y calculamos un par coseno-seno (c, s) de tal forma que

$$\begin{bmatrix} b_{pp} & b_{pq} \\ b_{qp} & b_{qq} \end{bmatrix} = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}^T \begin{bmatrix} a_{pp} & a_{pq} \\ a_{qp} & a_{qq} \end{bmatrix} \begin{bmatrix} c & s \\ -s & c \end{bmatrix}$$

es diagonal, o sea que $b_{qp} = b_{pq} = 0$ y como A es simétrica entonces $a_{pq} = a_{pq}$.

La matriz A es sobreescrito como $B = J^T A J$

$$J = \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & s & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & -s & \cdots & c & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix}$$

y c, s definidos arriba. Notemos que B coincide con A, excepto en la fila/columna p y fila/-columna q.

Definimos un -sweep- como la secuencia de N rotaciones donde $N = (n^2 - n)/2$ con n el tamaño de la matriz. Un solo -sweep- incluye un paso a través de todos los posibles entradas-(p, q). El algoritmo lleva a cabo -sweeps- que se repiten hasta que B converge a diagonal, esto quiere decir que (1) se va reduciendo en cada iteración.

El par elegido-(c, s) en J, puede ser elegido de manera eficiente. Hay una serie de formas de ordenar el ciclo a través de los pares-(p, q) [12].

2. Contenido

2.1. Algoritmo de Jacobi para matrices simétricas

La idea detrás del método de Jacobi es de forma sistemática, reducir (1), es decir el "tamaño"de lo que está afuera de la diagonal. Lo que usamos son "Jacobi rotations"que son de la forma J, eligiendo una entrada par-(p, q). Lo que queremos es diagonalizar matrices de 2×2 usando rotaciones, para eso tenemos "The 2-by-2 Symmetric Schur Decomposition". [2]

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix}^T \begin{bmatrix} a_{pp} & a_{pq} \\ a_{qp} & a_{qq} \end{bmatrix} \begin{bmatrix} c & s \\ -s & c \end{bmatrix} = \begin{bmatrix} b_{pp} & b_{pq} \\ b_{qp} & b_{qq} \end{bmatrix}$$

Donde queremos que

$$b_{pq} = a_{pq}(c^2 - s^2) + (a_{pp} - a_{qq})cs = 0 \tag{2}$$

la ecuación (2) la multiplicamos por $-1/2a_{pq}c^2$, suponiendo que $a_{pq} \neq 0$ y tenemos

$$0 = \frac{a_{pq}(c^2 - s^2)}{-2a_{pq}c^2} + \frac{(a_{pp} - a_{qq})cs}{-2a_{pq}c^2} = \frac{(s^2 - c^2)}{2c^2} + \frac{(a_{qq} - a_{pp})s}{2a_{pq}c} \tag{3}$$

si, $a_{pq} = 0$ solo tendríamos $(c, s) = (1, 0)$, de otra manera definimos

$$\tau = \frac{a_{qq} - a_{pp}}{2a_{pq}}, \qquad t = \frac{s}{c}$$

multiplicamos (3) por 2 y nos queda

$$0 = \frac{s^2}{c^2} - 1 + 2\frac{(a_{qq} - a_{pp})}{2a_{pq}}\frac{s}{c} = t^2 + 2\tau t - 1 \tag{4}$$

resolviendo (4) tenemos

$$t = -\tau \pm \sqrt{\tau^2 + 1}$$

para obtener los valores de c y s usamos

$$c^2 + s^2 = 1, \qquad t = \frac{s}{c}$$

y despejamos, obteniendo

$$c = \frac{1}{\sqrt{1 + t^2}}, \qquad s = tc$$

Entonces nuestro pseudocógido para sacar los valores de c, s queda de la siguiente forma

```
function sencos(A)
  if A[p,q]!=0, then
    tau=(A[q,q]-A[p,p])/(2A[p,q])
    if tau >=0, then
      t <- 1/(tau + sqrt(1+tau^2))
    else, then
      t <- -1/(-tau + sqrt(1+tau^2))
    end
    c <- 1/sqrt(1+t^2)
    s <- tc
  else then
    c <- 1
    s <- 0
  end
end
```

Como mencionamos arriba, solo las columnas/filas p y q son alteradas cuando el subproblema (p, q) es resuelto. Pero, ¿cómo elegimos nuestra entrada (p, q) de manera eficiente?. Hay dos maneras: una es eligiendo el elemento más grande fuera de la diagonal y otro es pasando por cada una de las entradas que están arriba de la diagonal de manera ordenada. El pseudocódigo para elegir el elemento más grande arriba de la diagonal es:

```
function maximum(A)
  n <- size(A,1)
  k,l,amax <- (0,0,0)
    for i, 1 to n-1, do
      for j, i+1 to n, do
        if abs(A[i,j])>=amax, then
           amax <- abs(A[i,j])
           k <- i
           l <- j
        end
      end
    end
   return [k,l]
end
```

El pesudocódigo para pasar por toda los índices arriba de la diagonal es trivial.

Antes de llegar al pseudocódigo del método de Jacobi ocupamos otros dos códigos, uno que me dé el "tamaño"que definimos arriba para la convergencia y otro que me genere J ya teniendo c, s. Para el tamaño es:

```
function fueradiagonal(A)
  n <- size(A,1)
  h <- 0
    for i, 1 to n-1, do
       for j,i+1 to  n, do
```

```
            h <- h+2*A[i,j]^2
        end
      end
    return sqrt(h)
end
```

Y para el otro es el siguiente, donde (n tamaño de la matriz, el vector de las matrices rotacionales, los valores de c y s)

```
function matrizrotacional(n,c,d)
  J=eye(n)
  J[c[1],c[1]] <- d[1]
  J[c[1],c[2]] <- d[2]
  J[c[2],c[1]] <- -d[2]
  J[c[2],c[2]] <- d[1]
  return J
end
```

Con lo antes expresado, podemos hacer nuestro algoritmo de Jacobi.

```
function Jacobi(A)
  n <- size(A,1)
  B <- eye(n) #Matriz identidad de n x n
  tol=10.0^(-10)
  while fueradiagonal(A) > tol, do
    for 1 to (((n-1)n)/2), do
      coo <- maximum(A)
      num <- sencos(A,coo)
      mat <- matrizrotacional(n,coo,num)
      B <- B*mat
      A <- transpose(mat)*A*mat
    end
  end
  return A,B  # A matriz de autovalores y B la matriz de autovectores
end
```

El algoritmo anterior se llama -Jacobi Classic-. La otra manera que dijimos arriba es ir por cada uno de los que están afuera de la diagonal de manera ordenada; su pseudocódigo no es muy diferente de éste último, por tanto queda así:

```
function Jacobi2(A)
  n <- size(A,1)
  B <- eye(n)
  tol <- 10.0^(-10)
  while fueradiagonal(A)>tol, do
    for i, 1 to n-1, do
      for j, i+1 to n, do
        num <- sencos(A,[i,j])
        mat <- matrizrotacional(n,[i,j],num)
        B <- B*mat
```

```
          A <- transpose(mat)*A*mat
        end
      end
    end
    return A,B
end
```

El algoritmo anterior se llama -Cyclic-by-row- y con esto ya podemos obtener los autovalores y autovectores de una matriz simétrica de cualquien tamaño, y, además, descompusimos la matriz de esta manera $A = BDB^T$, donde B es ortogonal, i.e. $BB^T = I$.

Ejemplo del algoritmo de Jacobi con los dos métodos de elección de la entrada (p, q). Con fueradiagonal(A)$< 10^{-10}$. Sea

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 5 & 6 & 7 \\ 3 & 6 & 6 & 9 \\ 4 & 7 & 9 & 10 \end{bmatrix}$$

aplicando el -Jacobi classic- obtenemos

$$D = \begin{bmatrix} -0.71853 & 1.292e-26 & -2.342e-22 & -3.050e-21 \\ 5.442e-17 & 0.556515 & 6.058e-28 & 1.193e-18 \\ 4.629e-16 & 2.327e-16 & -1.28015 & -1.972e-31 \\ 1.053e-16 & 2.085e-16 & 8.398e-16 & 23.4422 \end{bmatrix}$$

y

$$B = \begin{bmatrix} 0.777873 & -0.576317 & 0.0946697 & 0.231968 \\ 0.443828 & 0.757935 & -0.147228 & 0.454835 \\ -0.267994 & -0.0129932 & 0.79698 & 0.54114 \\ -0.355118 & -0.305326 & -0.578086 & 0.668194 \end{bmatrix}$$

donde D es la matriz de autovalores en la diagonal y B la matriz de autovectores respectivamente. Ahora con -Cyclic-by-row-

$$D = \begin{bmatrix} 0.556515 & -1.668e-14 & -1.373e-19 & -2.228e-22 \\ -1.694e-14 & -0.71853 & 8.122e-23 & 3.786e-37 \\ -3.204e-16 & 1.414e-16 & 23.4422 & 0 \\ 7.120e-16 & -9.394e-16 & -1.467e-15 & -1.28015 \end{bmatrix}$$

y

$$B = \begin{bmatrix} 0.576317 & 0.777873 & 0.231968 & -0.0946697 \\ -0.757935 & 0.443828 & 0.454835 & 0.147228 \\ 0.0129932 & -0.267994 & 0.54114 & -0.79698 \\ 0.305326 & -0.355118 & 0.668194 & 0.578086 \end{bmatrix}$$

Vemos que los dos algoritmos regresan resultados muy parecidos y defieren muy poco. Además de obtener los autovalores de la matriz con presición albitraria también tenemos una descomposición de la matriz A donde B es ortogonal.

La pregunta interesante que tenemos ahora es ¿Cuántos -sweeps- ocupamos para llegar a la convergencia? Hay un resultado en 1985 hecho por Brent and Luk, quienes han argumentado

de manera heurística que el número de -sweeps- es proporcional a $\log n$.

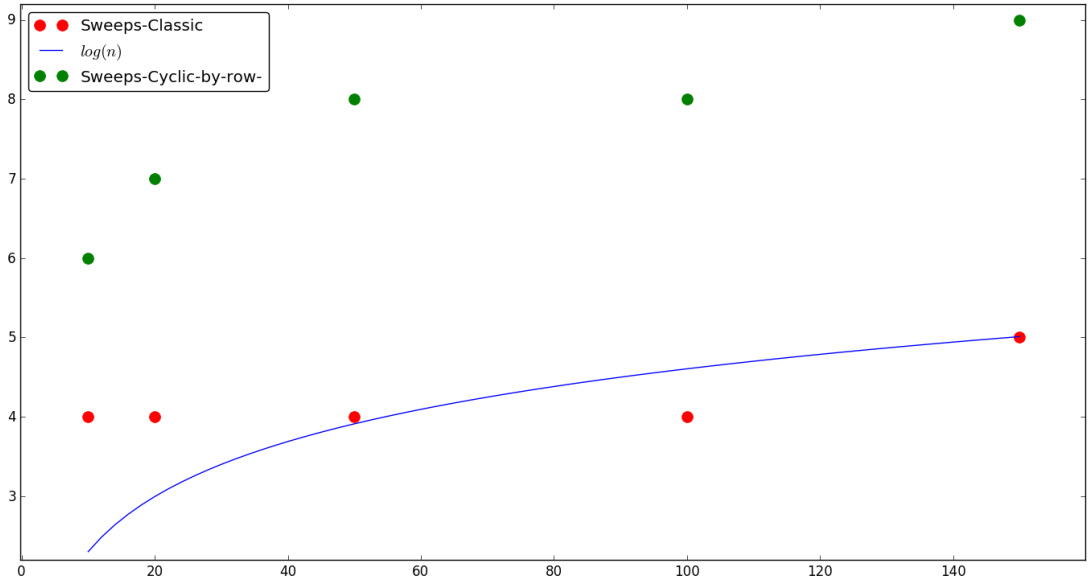
Mostraré una tabla del número de -sweeps- para llegar fueradiagonal(A)$< 10^{-10}$ hecho numéricamente. Para elegir la matriz A hice un programa que hiciera matrices simétricas de $n \times n$ con entradas de números al azar.

Jacobi classic	5	10	20	50	70	100	150
-sweeps-	3.04	3.84	4	4.02	4	4.04	4→5
$\log n$	≈ 2.303	≈ 2.99	≈ 3.91	≈ 4.60	≈ 5.01		

Todos las matrices donde aplicaba el algoritmo ocupaban 4 -sweeps- a excepción del último. Con -cyclic-by-row-

-cyclic-by-row	10	20	50	100	150
Promedio de -sweeps-	5.7 → 6	6.7→ 7	7.4→8	8	9

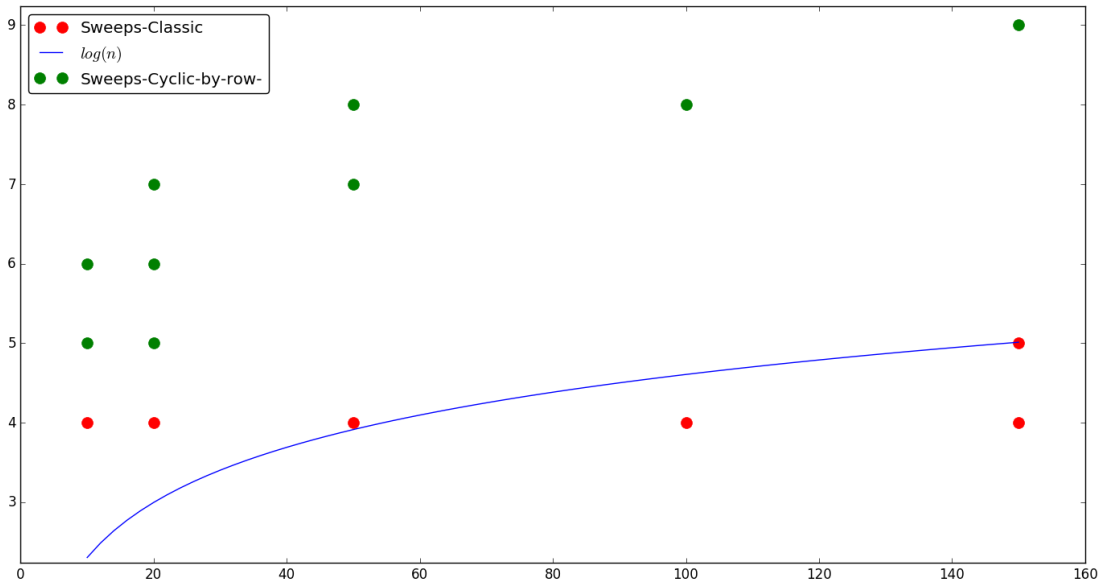
Vemos que es mas rápido el de Jacobo classic. Si lo graficamos



Fui bajando el número de veces de repeticiones cuando el tamaño de la matriz iba subiendo, como se muestra en la siguiente tabla

Tamaño de la matriz n	10	20	50	100	150
Número de repeticiones	10	10	5	2	2

así se ve si graficamos los resultados individuales



2.2. ¿Por qué programación en paralelo del algoritmo de Jacobi?

Al principio hablamos un poco de computación en paralelo, ahora hablaremos de que ventajas tiene la programación en paralelo con el algoritmo de Jacobi.
Primero, la ventajas de trabajar en paralelo son:

- Obtener resultados más rápido de lo que se puede con un solo procesador.
- Aprovechar el hardware que se encuentra en nuestra computadora o en la nube.

Lo que hace que el algoritmo de Jacobi se pueda utilizar en paralelo, es que muchas operaciones son independientes de cada uno. La pre o post-multipliación con la matriz J^k solo afecta a la fila y columna p y q en la matriz A^{k+1}. Esto nos permite realizar rotaciones para diferentes valores de p y q simultáneamente, siempre y cuando las operaciones no afecten la misma fila o columnas.

Otra ventaja del método de Jacobi es ahora sobre el algoritmo QR simétrico (la idea es la descomposición de una matriz $A = QR$ donde Q es ortogonal y R es una matriz triangular superior. [11]) es su paralelismo. Como cada actualización de Jacobi consiste en una rotación de filas que afecta sólo a las filas p y q, y una rotación de columnas que sólo afecta las columnas p y q, hasta $n/2$ las actualizaciones de Jacobi se pueden realizar en paralelo.

2.3. ¿Qué podemos implementar en paralelo del algoritmo de Jacobi?

Como el tamaño de la matriz, n, es generalmente mucho mayor que el número de procesadores, p, es común usar un enfoque de bloque, en el que cada actualización consiste en el cálculo de una descomposición de Schur simétrica $2r \times 2r$ para algún bloque elegido de tamaño r. Esto se logra aplicando otro algoritmo, como el algoritmo QR simétrico, en una escala más pequeña. Entonces, si $p \geq n/(2r)$, un bloque completo Jacobi -sweep- puede ser paralelizado. [1]

Por lo tanto, para almacenar un bloque de matriz A es necesario reservar una memoria de tamaño $(2sk + s) \times (2sk + s)$, y para almacenar un bloque de V es necesario un recuerdo de tamaño $(2sk + s) \times 2sk$.

El algoritmo base de casi todas las implementaciones secuenciales actuales (también para computadores paralelos con memoria compartida) es el algoritmo iterativo QR de Francis con doble desplazamiento implícito (método donde con una sencilla manipulación algebraica se factoriza una matriz M, en $M = QR$, y se obtiene $A_{k+1} = Q^T MQ$, con lo cual se evitaría un paso). Como paso previo a este algoritmo debemos reducir la matriz a la forma de Hessenberg (es una matriz triangular excepto por una subdiagonal adyacente a la diagonal principal). Existen otros métodos iterativos basados en algoritmos tipo Jacobi. Éstos sin embargo no son convergentes globalmente, dado que requieren un número más elevado de iteraciones, aunque son muy competitivos para matrices simétricas.

Recientemente se han propuesto nuevos algoritmos para la reducción de matrices simétricas a la forma tridiagonal y la posterior obtención del sistema de valores propios utilizando transformaciones tipo Jacobi aplicadas solo a un lado de la matriz.

Para diseñar un algoritmo paralelo, lo primero que debemos hacer es decidir la distribución de los datos a los procesadores. Esta distribución y el movimiento de datos en las matrices determinan las necesidades de la memoria y la transferencia de datos en un sistema distribuido.

Cada uno de los bloques de tamaño $2sk \times 2sk$ se considera como un proceso y tendrá una necesidad particular de memoria. Por lo menos necesita memoria para almacenar los bloques iniciales, pero se necesita alguna memoria adicional para almacenar datos en pasos sucesivos de la ejecución. Un esquema del método se muestra a continuación.

En cada proceso:
while no se alcanza la convergencia, do
    for cada conjunto de Jacobi en un barrido, do
        Realizar barridos en los bloques de tamaño $2s \times 2s$ correspondientes a índices asociados al procesador, acumulando las rotaciones
        Difundir las matrices de rotación
        Actualizar la parte de las matrices A y V asociadas al proceso, realizando multiplicaciones matriz-matriz
        Transferir filas y columnas de bloques de A y filas de bloques de V

end
end

Cada -sweep- se divide en un número de pasos que corresponden a cada paso a un conjunto de Jacobi.

Para cada conjunto de Jacobi se pueden calcular las matrices de rotación en paralelo, pero solo los procesos de trabajo asociados a los bloques $2sk \times 2sk$ de la diagonal principal de A. En estos procesos se realiza un -sweep- en cada uno de los bloques que contiene, correspondiente al conjunto de Jacobi en uso, y las rotaciones en cada bloque se acumulan en una matriz de rotaciones de tamaño $2s \times 2s$.

Después del cálculo de las matrices de rotanción, se envían a los otros procesos correspondientes a bloques en la misma fila y columna en la matriz A, y la misma fila en la matriz V. Y entonces los procesos puede actualizar la parte de A o V que contienen.

Para obtener el nuevo agrupamiento de datos de acuerdo con el siguiente conjunto de Jacobi es necesario realizar un movimiento de datos en la matriz, lo que implica una transferencia de datos entre procesos y necesidades adicionales de memoria. [4]

2.4. Problema SVD

Algo asombroso del método de Jacobi, es que también podemos extenderlo para resolver problemas de SVD. SVD es descomponer una matríz $A = U\Sigma V^T$ donde $A \in \mathbf{R}^{m\times n}$, $U \in \mathbf{R}^{m\times m}$($U$ ortogonal), $\Sigma \in \mathbf{R}^{m\times n}$(con autovalores en la diagonal), $V \in \mathbf{R}^{n\times n}$($V$ ortogonal). Pero nosotros solo haremos solo para matrices de $n \times n$

Ahora en vez de estar resolviendo una secuencia de problemas simétricos de autovalores 2×2, resolvemos una secuencia de problemas 2×2 SVD. Ahora para una entrada par-(p, q), calculamos un par de rotaciones de la forma:

$$\begin{bmatrix} c_1 & s_1 \\ -s_1 & c_1 \end{bmatrix}^T \begin{bmatrix} a_{pp} & a_{pq} \\ a_{qp} & a_{qq} \end{bmatrix} \begin{bmatrix} c_2 & s_2 \\ -s_2 & c_2 \end{bmatrix} = \begin{bmatrix} d_1 & 0 \\ 0 & d_2 \end{bmatrix}$$

Pero como es mas dificil buscar c_2, s_2, resolveremos primero algo mas fácil de la forma:

$$B = \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} a_{pp} & a_{pq} \\ a_{qp} & a_{qq} \end{bmatrix}$$

Donde $c^2 + s^2 = 1$ y B es de la forma

$$B = \begin{bmatrix} l & m \\ m & n \end{bmatrix}$$

o sea, B es simétrica. Para saber los valores de c, s tenemos que:

$$\begin{bmatrix} l & m \\ m & n \end{bmatrix} = \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} a_{pp} & a_{pq} \\ a_{qp} & a_{qq} \end{bmatrix}$$

Donde obtenemos que:

$$\begin{bmatrix} l & m \\ m & n \end{bmatrix} = \begin{bmatrix} a_{pp}c + a_{qp}s & a_{pq}c + a_{qq}s \\ a_{qp}c - a_{pp}s & a_{qq}c - a_{pq}s \end{bmatrix}$$

Aquí tenemos un sistema de ecuaciones que satisface que:

$$a_{pq}c + a_{qq}s = m, \qquad a_{qp}c - a_{pp}s = m, \qquad c^2 + s^2 = 1$$

Resolviendo para s, c queda

$$c = \frac{(a_{pp} + a_{qq})m}{a_{pp}a_{pq} + a_{qp}a_{qq}}, \qquad s = \frac{(a_{qp} - a_{pq})m}{a_{pp}a_{pq} + a_{qp}a_{qq}}$$

pero además para que se cumpla que $c^2 + s^2 = 1$, sustituimos y resolvemos para m y queda que es

$$m = \pm \frac{1}{\sqrt{\frac{(a_{pq}-a_{qp})^2+(a_{pp}+a_{qq})^2}{(a_{pp}a_{pq}+a_{qp}a_{qq})^2}}}$$

Pero tomaremos la parte positiva, por simplicidad; teniendo esto, podemos continuar resolviendo problemas SVD, entonces resolviendo c, s, nos quedará que B es simétrica entonces por el algoritmo de Jacobi podemos diagonalizarla

$$\begin{bmatrix} c_2 & s_2 \\ -s_2 & c_2 \end{bmatrix}^T \begin{bmatrix} l & m \\ m & n \end{bmatrix} \begin{bmatrix} c_2 & s_2 \\ -s_2 & c_2 \end{bmatrix} = \begin{bmatrix} d_1 & 0 \\ 0 & d_2 \end{bmatrix}$$

Y por la primera ecuación tenemos que

$$\begin{bmatrix} c_2 & s_2 \\ -s_2 & c_2 \end{bmatrix}^T \begin{bmatrix} c & s \\ -s & c \end{bmatrix}^T = \begin{bmatrix} c_1 & s_1 \\ -s_1 & c_1 \end{bmatrix}^T$$

Y obtenido esto, hacemos J con c_1, s_1 y c_2, s_2 y hacemos lo mismo que el algoritmo anterior y continuamos aplicando el algoritmo de Jacobi para llegar a tener una descomposición SVD. [7]

Nuestro algoritmo para matrices $A \in \mathbf{R}^{n\times n}$ no simétricas, se parece mucho a los pseudocódigos ya hechos anteriormente solo cambiando pequeños detalles. Primero tenemos que ver lo que está afuera de la diagonal, solo que ahora ya no podemos pasar solo por la parte superior sino por toda la matriz a excepción de la diagonal:

```
function fueradiagonalSVD(A)
  n <- size(A,1)
  h <- 0
    for i, 1 to n, do
        for j, 1 to n dp
          if j!=1, them
           h <- h+A[i,j]^2
          end
```

```
        end
      end
    return sqrt(h)
end
```

Para ver cuál es el elemento mas grande, ahora tenemos que pasar por toda la matriz a excepción de la diagonal, el pseudocódigo queda de la forma:

```
function maximumSVD(A)
  n <- size(A,1)
  k,l,amax <- (0,0,0)
    for i, 1 to n, do
      for j, 1 to n, do
        if j!=i, them
          if abs(A[i,j])>=amax, them
            amax <- abs(A[i,j])
            k <- i
            l <- j
          end
        end
      end
    end
  return [k,l]
end
```

Ahora, para obtener los índices estaremos pasando por toda la matriz, si llegamos a que $k > l$ y queremos tomar la matriz de 2×2 de esos índices utilizando el algoritmo ya definido; lo que va a pasar es que la matriz se volteará y nos perjudicará cuando estemos iterando, así que tenemos que checar bien cuando $k > l$ con el siguiente pseudocódigo, como pasa en este ejemplo

$$A = \begin{bmatrix} 4 & 11 & 5 \\ 14 & 8 & 6 \\ 7 & -2 & 5 \end{bmatrix}$$

Si aplicamos el algoritmo para ver cuales son lo índices de elemento mas grande fuera de la diagonal serán $[2, 1]$, teniendo eso queremos sacar la matriz de 2×2, pero no queremos que pase esto

$$A = \begin{bmatrix} 8 & 11 \\ 14 & 4 \end{bmatrix}$$

Para evitar esto ocupamos el siguiente pseudocódigo:

```
function CorreccionSVD(A,a) # A-matriz y a-vector de los indices
  M <- eye(2)
  if a[1]<a[2], them
    M[1,1] <- A[a[1],a[1]]
    M[1,2] <- A[a[1],a[2]]
    M[2,1] <- A[a[2],a[1]]
    M[2,2] <- A[a[2],a[2]]
  elseif a[1]>a[2], them
```

```
    M[1,1] <- A[a[2],a[2]]
    M[1,2] <- A[a[2],a[1]]
    M[2,1] <- A[a[1],a[2]]
    M[2,2] <- A[a[1],a[1]]
  end
  return M
end
```

Y con eso corregimos el error, pero nos falta como obtener c, s para así obtener s_1, c_1 y s_2, c_2, se obtiene con la fórmula que ya dimos arriba, solo es pasarlo a un pseudocódigo

```
function SimetricaSVD(A)
  M <- eye(2)
  c1 <- (A[1,1]+A[2,2])/(A[1,1]*A[1,2]+A[2,1]*A[2,2])
  s1 <- (-A[1,2]+A[2,1])/(A[1,1]*A[1,2]+A[2,1]*A[2,2])
  m= <- 1/sqrt(s1^2+c1^2)
  c <- c1*m
  s <- s1*m
  M[1,1] <- c
  M[1,2] <- s
  M[2,1] <-s
  M[2,2] <- c
  return M*A,M     # M*A-Matriz simetrica, M-Matriz de c,s
end
```

Este es el último pseudocódigo extra que ocupamos, dado que lo otros programas ya lo hemos hecho, ahora solo falta implementarlo para que sólo acepte una matriz de $n \times n$ y obtener la SVD.

```
function SVD(A)
  n <- size(A,1)
  U <- eye(n)
  V <- eye(n)
  tol <- 10^(-6)
  while fueradiagonalSVD(A)>tol, do
    for 1 to ((n^2-n)/2), do
      Coor <- maximumSVD(A)
      Corr <- CorreccionSVD(A,Coor)          # Hace la matriz de 2x2
      CS <- SimetricaSVD(Corr)               # Saca s c y d1 d2
      C2S2 <- Jacobi(CS[1])                  # Obtiene c2 y s2
      C1S1 <- CS[2]*transpose(C2S2[2])       # Obtiene c1 y s1
      mat1 <- matrizrotacion(n,Coor,C1S1[:,1])
      mat2 <- matrizrotacion(n,Coor,C2S2[2][:,1])
      U <- U*transpose(mat1)
      V <- V*mat2
      A <- mat1*A*mat2
    end
  end
```

```
    return (U,A,V)
end
```

En vez de usar Jacobi(A) también pudimos haber utilizado Jacobi2(A), o también podemos haber hecho que pase por cada entrada a excepción de la diagonal. Este es nuestro pseudocódigo de SVD para matriz de $n \times n$ no simétricas. [5]

Ejemplo: Tomamos la matriz

$$A = \begin{bmatrix} 4 & 11 & 5 \\ 14 & 8 & 6 \\ 7 & -2 & 5 \end{bmatrix}$$

Aplicando el algoritmo obtenemos

$$U = \begin{bmatrix} 0.519939 & 0.697381 & -0.493278 \\ 0.804605 & -0.205926 & 0.55696 \\ 0.286834 & -0.68648 & -0.668185 \end{bmatrix}, V = \begin{bmatrix} 0.725021 & -0.547344 & -0.418042 \\ 0.546998 & 0.826439 & -0.133388 \\ 0.418495 & -0.131959 & 0.898581 \end{bmatrix}$$

$$\Sigma = \begin{bmatrix} 21.1747 & 0 & 1.67433e-8 \\ -8.88178e-16 & 8.95013 & -1.8875e-13 \\ 2.38034e-7 & -3.79677e-12 & -2.74383 \end{bmatrix}$$

si multiplicamos $U\Sigma V^T$ obtemos de vuelta A.

Ejemplo 2;

$$A1 = \begin{bmatrix} 1 & 4 & 5 \\ 4 & 6 & 7 \\ 6 & 5 & 6 \end{bmatrix}$$

Aplicando al algoritmo obtenemos

$$U = \begin{bmatrix} 0.715978 & -0.404735 & -0.568828 \\ 0.203609 & -0.658312 & 0.724686 \\ -0.667772 & -0.634677 & -0.388929 \end{bmatrix}, V = \begin{bmatrix} 0.893164 & 0.449188 & 0.0220976 \\ -0.269334 & 0.5736 & -0.773591 \\ -0.360163 & 0.684992 & 0.6333 \end{bmatrix}$$

$$\Sigma = \begin{bmatrix} -2.77241 & -7.11727e-8 & -6.77626e-21 \\ -6.14851e-7 & -15.2409 & -1.16958e-11 \\ -4.23516e-20 & 8.03768e-12 & -0.165664 \end{bmatrix}$$

si multiplicamos $U\Sigma V^T$ obtemos de vuelta A.

2.5. Velocidad de convergencia

Sabemos que podemos elegir de diferente manera nuestro (p, q), pero ¿Se podrá llegar a atorar el método de Jacobi utilizando cualquier manera de tomar (p, q) para una matriz simétrica?. El siguiente teorema considera esto.

2.5.1. Teorema (Rate of diagonaliazation for the Jacobi Algorithm):

Sea A una matriz simétrica de $n \times n$. Sea $A^{(n)}$ que denota la matriz producida en el paso n del algoritmo de Jacobi. Entonces:

$$\text{Off}(A^{(n)}) \leq \left(1 - \frac{2}{n^2 - n}\right)^m \text{Off(B)}$$

Ya que

$$\left(1 - \frac{2}{n^2 - n}\right) < 1$$

y

$$\lim_{m \to \infty} \left(1 - \frac{2}{n^2 - n}\right)^m = 0$$

Entonces el teorema dice que

$$\lim_{m \to \infty} \text{Off}(A^{(m)}) = 0$$

Por lo tanto, la respuesta a la pregunta es que el método de Jacobi nunca se trabará.

Para el método de -Cyclic-by-row-, Wikilson(1962) dijo que converge de forma cuadrática. [2] Ejemplo, tomamos la siguiente matriz

$$A = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 \\ 1 & 3 & 6 & 10 \\ 1 & 4 & 10 & 20 \end{bmatrix}$$

al aplicar algoritmo con Offdiags(A)<10^{-10} ya definido arriba podemos hacer una tabla

Sweeps	Offdiags(A)
0	16
1	1.41193
2	0.16232
3	0.00041
4	4.48438e-16

otro ejemplo

$$A1 = \begin{bmatrix} 10 & 4 & 5 & 4 & 2 & 4 \\ 4 & 4 & 2 & 1 & 4 & 0 \\ 5 & 2 & 2 & 4 & 3 & 1 \\ 4 & 1 & 4 & 5 & 1 & 4 \\ 2 & 4 & 3 & 1 & 3 & 2 \\ 4 & 0 & 1 & 4 & 2 & 1 \end{bmatrix}$$

lo mismo que el anterior

Sweeps	Offdiags(A1)
0	17.02938
1	3.26136
2	0.54807
3	0.00822
4	1.16001e-7
5	1.00441e-19

Para el algoritmo Clásico de Jacobi, la velocidad de convergencia del algoritmo es considerablemente mejor que lineal. Además -Wilkinson J.H- En su libro menciona que $r > 2\log\frac{1}{\epsilon}$. Ahora haciendo lo mismo pero con el método Clásico de Jacobi tenemos

Sweeps	Offdiags(A)
0	16
1	0.29679
2	0.00201
3	1.953e-12

y

Sweeps	Offdiags(A1)
0	17.02938
1	1.831198
2	0.038380
3	2.212615e-6
4	4.338804e-17

Aquí también se puede ver que es mas rápido el algoritmo Clásico de Jacobi que -Cyclic-by-row-.

2.6. Block Jacobi Symmetric Eigenvalue

Dividir en $n \times n$ matrices A como la siguiente:

$$J = \begin{bmatrix} A_{11} & \cdots & A_{1N} \\ \vdots & \ddots & \vdots \\ A_{N1} & \cdots & A_{NN} \end{bmatrix}$$

En una "Block Jacobi procedure", el subproblema (p, q) implica resolver $2r - by - 2r$ Schur decomposition

$$\begin{bmatrix} V_{pp} & V_{pq} \\ V_{qp} & V_{qq} \end{bmatrix}^T \begin{bmatrix} A_{pp} & A_{pq} \\ A_{qp} & A_{qq} \end{bmatrix} \begin{bmatrix} V_{pp} & V_{pq} \\ V_{qp} & V_{qq} \end{bmatrix} = \begin{bmatrix} D_{pp} & 0 \\ 0 & D_{qq} \end{bmatrix}$$

Este ya se nos ha presentado antes, y ya tenemos un algoritmo para obtener $V_{pp}, V_{qp}, V_{pq}, V_{qq}$

2.7. Preguntas extras

¿Habrá otra forma de sacar la SVD de $A \in \mathbf{R}^{m\times n}$? ¿Qué pasa si multiplicamos AA^T? Es claro que obtenemos una matriz simétrica, pero sabemos que podemos descomponerla en $A = U\Sigma V^T$ donde U, V son ortogonales, si sustituimos esto en AA^T o A^TA

$$AA^T = (U\Sigma V^T)(U\Sigma V^T)^T = (U\Sigma V^T)(V\Sigma U^T) = U(\Sigma^2)U^T$$

$$A^TA = (U\Sigma V^T)^T(U\Sigma V^T) = (V\Sigma U^T)(U\Sigma V^T) = V(\Sigma^2)V^T$$

Y como tenemos un programa para calcular los autovalores y autovectores de una matriz simétrica. Entonces tenemos V y U, y solo sería sacar la raíz de$\sqrt{\Sigma}$ para obtenerlo, pero nos faltaría acomodar los valores singulares, de mayor a menor, haciendo eso obtemos la SVD de A. [6]

Otra manera de utilizar a Jacobi, es en la resolución de ecuaciones lineales. Podemos ver a $A = D + R$ donde D es la diagonal de A y R el resto de la matriz A

$$Ax = b$$

$$(D + R)x = b \Rightarrow Dx + Rx = b \Rightarrow Dx = b - Rx \Rightarrow x = D^{-1}(b - Rx)$$

Entonces el pseudocódigo queda así, donde x_0 es igual a un vector cualquiera y n el número de iteraciones

```
function JacobiLineal(A,b,x_0,n)
  D <- eye(length(b),length(b))
  R <- zeros(length(b))
  for i, 1 to length(b), do
    D[i,i]=A[i,i]
  end
  R <- A-D
  for 1 to n, do
    x_0 <- (D^(-1))*(b-R*x_0)
  end
  return x_0
end
```

3. Bibliografía

Referencias

[1] Jim Lambers.,*Lecture 7 Notes*, 2010-11. Recuperado de http://web.stanford.edu/class/cme335/lecture7.pdf

[2] Horn, R.A. y Johnson, C.R., *Matrix Analysis*, segunda edición, Cambridge University Press, págs. 87–88, 2013. Recuperado de http://web.mit.edu/ehliu/Public/sclark/Golub%20G.H.,%20Van%20Loan%20C.F.-%20Matrix%20Computations.pdf

[3] Martínez F. C. (2008), *Implicit Jacobi Algorithms for the Symmetric Eigenproblem*. Universidad Carlos III de Madrid. España. Recuperado de http://gauss.uc3m.es/web/personal_web/fdopico/talks/2008ilas.pdf

[4] Takahashi, Y. , Hirota, Y. y Yamamoto, Y.. (2012). *Performance of the block Jacobi method for the symmetric eigenvalue problem on a modern massively parallel computer. Proceedings of ALGORITMY* pp. 151–160. Recuperado de http://www.iam.fmph.uniba.sk/algoritmy2012/zbornik/16Takahashi.pdf

[5] Cline, A.K. y Dhillon, I.S.. *Computation of the Singular Value Decomposition*. The University of Texas at Austin. Recuperado de http://www.cs.utexas.edu/users/inderjit/public_papers/HLA_SVD.pdf

[6] MATH2071: LAB #9: *The Singular Value Decomposition*. Recuperado de http://www.math.pitt.edu/s̃ussmanm/2071Spring08/lab09/lab09.pdf

[7] Brent R.P., Luk F.T. y Van Loan Ch. *Computation of the Singular Value Decomposition using Mesh-Connexted Processors* Journal of VLSI and Computers Systems, Volume 1, Number 3. Recuperado de https://www.cs.cornell.edu/cv/ResearchPDF/Comp.Sing.Value.Decomp.Using.Mesh.Connect.Proc..pdf

[8] Kennet H. R. (2007) *HandBook of Linear Algebra* United States of America. Edit. Chapman & Hall/CRC. Recuperado de http://www2.fiit.stuba.sk/k̃vasnicka/QuantumComputing/Hogben-Handbook %20of %20Linear %20Algebra-(CRC %20press, %202007).pdf

[9] Van de Geijn R.A. (1988) *A novel storage scheme for parallel Jacobi Methods* The University of Texas. United States of America. Recuperado de ftp://www.cs.utexas.edu/pub/techreports/tr88-26.pdf

[10] Okša G. y Vajteršic M. (2007) *Parallel One-Sided Block Jacobi SVD Algorithm: I. Analysis and Design* Universität Salzburg. Austria. Recuperado de https://www.cosy.sbg.ac.at/research/tr/2007-02_Oksa_Vajtersic.pdf

[11] https://es.wikipedia.org/wiki/Algoritmo_QR

[12] Martin D.C. y Tongen A. (2011) *Keeping it R.E.A.L. Research Experiences for ALL Learnes* Mathematical Association of America, Inc.