Trabajo Práctico 1
93.75 - Métodos Numéricos Avanzados
Brusselator

Grupo 1
di Tada, Teresa Celina - 52354
Gualino, Matías Sergio - 53344
Sakuda, María Eugenia - 53191
Varela, María de las Mercedes - 51332

Octubre 2015

Contenidos

Introducción

Brusselator es un modelo matemático de reacción-difusión para reacciones químicas en un reactor tubular. Este permite calcular una matriz A que representa la discretización del Jacobiano del problema. Nuestro objetivo es generar dicha matriz por medio de la recepción de parametros relacionados a la reacción química y el tamaño de la matriz a realizar para luego obtener sus autovalores por medio de dos métodos diferentes.

Para la verificación del correcto funcionamiento de los métodos se utilizarán dos matrices cuadradas obtenidas de `Matrix Market` [1], y estas serán compararadas con la generación de la discretización de nuestra autoría.

[1] http://math.nist.gov/MatrixMarket/data/NEP/

Capítulo 1

Modelo Brusselator

Las concentraciones $x(t,z)$ y $y(t,z)$ de dos componentes de difusión y reactivos estan modelados por el siguiente sistema:

$$\frac{\partial x}{\partial t} = \frac{\delta_1}{L^2}\left(\frac{\partial^2 x}{\partial^2 z}\right) + f(x,y) \tag{1.1}$$

$$\frac{\partial y}{\partial t} = \frac{\delta_2}{L^2}\left(\frac{\partial^2 y}{\partial^2 z}\right) + g(x,y) \tag{1.2}$$

con condiciones iniciales tales como $x(0,z) = x_0(z), y(0,z) = y_0(z)$ y las condiciones de borde de Dirichlet $x(t,0) = x(t,1) = x^*, y(t,0) = y(t,1) = y^*$ donde $0 \leq z \leq 1$ es el espacio de coordenadas a lo largo del tubo y t, el tiempo. En el modelo llamado Brusselator se consideran:

$$f(x,y) = \alpha - (\beta+1)x + x^2 y,\ g(x,y) = \beta x - x^2 y$$

Este sistema admite las soluciones estacionarias triviales $x^* = \alpha$, $y^* = \frac{\beta}{\alpha}$. En este problema en particular nos interesamos principalmente en las soluciones periodicas estables del sistema como la bifurcación del parámetro L varía. Esto ocurre cuando los autovalores de las partes reales del Jacobiano del lado derecho de (1.1) y (1.2) se evaluan en la solución de estación constante, que es puramente imaginaria. A los efectos de verificar esto numéricamente, primero hay que discretizar las ecuaciones respecto de la variable z y calcular los autovalores con partes reales de la Jacobiana discreta resultante.
Si discretizamos el intervalo [0,1] usando n puntos interiores con el tamaño de malla $h = \frac{1}{(m+1)}$, entonces el Jacobiano discretizado es una matriz de 2x2

con la forma:

$$\mathbf{A} = \begin{pmatrix} \tau_1\mathbf{T} + (\beta+1)\mathbf{I} & \alpha^2\mathbf{I} \\ -\beta\mathbf{I} & \tau_2\mathbf{T} - \alpha^2\mathbf{I} \end{pmatrix} \tag{1.3}$$

donde $\mathbf{T} = tridiag\{1, -2, 1\}$, $\tau_1 = \frac{1}{h^2}\frac{\delta_1}{\mathbf{L}^2}$ y $\tau_2 = \frac{1}{h^2}\frac{\delta_2}{\mathbf{L}^2}$. Donde:

- α es la constante en terminos de reacción para x.
- β es el coeficiente en terminos de reacción para y.
- δ_1 es el coeficiente de difusión para x.
- $\delta 2$ es el coeficiente de difusión para y.
- $\mathbf{L}$ es el parámetro de bifurcación ($\mathbf{L}^2$ divide a δ_1 y δ_2).

Capítulo 2

Generación matriz A

El modelo Brusslator define una opción analítica de cálculo para la matriz del Jacobiano del problema. La misma está programada en el siguinte código de Octave, donde se utiliza *calculate_tridiag* y *calculate_identity* que son métodos que construyen la matriz tridiagonal y la matriz identidad requeridas.

```
1  % Generate A matrix for Brusselator problem
2  % Parameters:
3  % m - order of the matrix / 2
4  % a - constant in reaction term for x
5  % b - coefficient in reaction term for y
6  % delt1 - diffusion coefficient for x
7  % delt2 - diffusion coefficient for y
8  % L - bifurcation parameter (L2 divides DELT1 and DELT2)
9  function A = generate_A_matrix (m, a, b, delt1, delt2, L)
10   h = 1 / (m+1);
11   tau1 = delt1/((h*L)**2);
12   tau2 = delt2/((h*L)**2);
13
14   T = calculate_tridiag(m,1,-2,1);
15   I = calculate_identity(m);
16
17   a11 = tau1*T+(b-1)*I;
18   a12 = (a**2)*I;
19   a21 = -b*I;
20   a22 = tau2*T-(a**2)*I;
21
22   A = [a11,a12;a21,a22];
23
24 endfunction;
```

Capítulo 3

Cálculo de los autovalores

3.1 Cálculo analítico

Para la resolución de los autovalores mediante el cálculo analítico se utilzó la fórmula presentada en el documento `Matrix Market` [1] mediante la siguiente implementación.

```
function Aeig = eigen_form(m, a, b, delta1, delta2, L)
    h = 1/(m+1);

    tau1 = delta1/(h*L)^2;
    tau2 = delta2/(h*L)^2;
    for j=1:m,
       eigofT(j) = -2*(1- cos(pi*j*h) );   % eigenvalues of T
    end
    for j=1:m,
       coeff(1) = 1;
       coeff(2) = a^2 - (b - 1) - (tau1+tau2)*eigofT(j);
       coeff(3) = b*a^2 + tau1*tau2*eigofT(j)^2 + tau2*(b-1)*eigofT(j) -
      a^2*tau1*eigofT(j) - a^2*(b-1);
       d = roots(coeff);
       Aeig(j) = d(1);
       Aeig(m+j) = d(2); % eigenvalues of A
    end
end
```

[1] http://math.nist.gov/MatrixMarket/data/NEP/mvmbwm/mvmbwm.html

3.2 Cálculo numérico

A fin de resolver el cálculo numérico de autovalores, se recurrieron a dos algoritmo principales: la descomposición QR como herramienta para obtener una matrz R*Q y el "implicit double shift QR" como técnica de deflación para procesar los autovalores complejos. A continuación se desarrollan dichos algoritmos.

3.2.1 Descomposición QR

Para llevar a cabo la desocmposición QR de la matriz, se utilizó el algoritmo de Householder. El mismo es el de menor orden dentro de los tres vistos, siendo el mismo: $2n^2(\frac{m-n}{3})$

La implementación realizada es la siguiente:

```
function [Q,R] = qr(A)
[m, n] = size(A);
Q = eye(m,m);
R = A;

for k = 1:n
  x = R(k:m,k);
  s = sign(x(1))*norm(x);
  u = x;
  u(1) = x(1)+s;
  H = eye(m-k+1)-u*u'/(s*u(1));

  R(k,k) = -s;
  R(k+1:m,k) = 0;
  R(k:m,k+1:n) = H*R(k:m,k+1:n);

  Q(k:m,:) = H*Q(k:m,:);
end

Q = Q';
```

La idea principal de este algoritmo es ir transformando A en una matriz triangular superior (R) a partir de las reflexiones de Householder. A partir de estas matrices de Householder que se van formando y dado que tienen la peculiaridad de ser ortogonales, se va formando la matriz Q.

3.2.2 Autovalores

Se determinó desarrollar la técnica de *implicit double shift QR* para poder hallar los autovalores de las matrices A. Inicialmente se aplica la descom-

pocisión QR planteada con antrioridad para ir a proximando la matriz hasta poder distinguir cuadrados de 2x2 en la diagonal. Además se sabe que la matriz converge desde el extremo inferior derecho, hacia el extremo superior izquierdo. Para poder distinguir esos cuadrados se plantea una cota. Esta cota, junto con los elementos a comparar determinan si un autovalor es complejo o real. En caso de ser real, la condición es:

$$abs(A(i, i-1)) < (abs(A(i, i)) + abs(A(i-1, i-1))) * cota \tag{3.1}$$

Si se tratan de autovalores complejos:

$$abs(A(i-1, i-2)) < (abs(A(i-1, i-1)) + abs(A(i-2, i-2))) * cota \tag{3.2}$$

En el caso de que el autovalor sea real, se considera el último elemento de la diagonal como el autovalor. Por otro lado, si se trata de autovalores complejos, se desarrolla el polinomio característico de la matriz de 2x2 y se sacan los autovalores a partir de la obtención de las raíces del mismo. La cota que se utiliza para determinar si un autovalor es real, complejo o se debe seguir aproximando la matriz con QR, varía de acuerdo al tamaño de la matriz. en la mayoría de los casos que se probaron una conta de e-10 era suficiente, pero para la de m = 50 se tuvo que bajar la cota a e-12.

Implementación:

```
1  % Calculate the eig values of the A matrix
2  % A is the A matrix
3  % cota should be updated depending of the sieze of the matrix.
4
5  function E = custom_eig (A ,s)
6    i = size(A)(1);
7    E = zeros(1, i);
8    cota = 1e-12;
9
10   while i > 2
11     [Q, R] = qr(A);
12     A = R*Q;
13
14     %Recupero los autovalores de esa matriz
15     if (abs(A(i, i-1)) < (abs(A(i, i)) + abs(A(i -1, i - 1)))*cota)
16       E(i) = A(i,i);
17       i = i - 1;
18       A = A(1:i, 1:i);
19     elseif abs(A(i-1,i-2)) < (abs(A(i-1,i-1)) + abs(A(i-2,i-2)))*cota
20       ANS = roots([1, - (A(i-1,i-1) + A(i,i)), A(i-1, i-1)*A(i,i) - A(i
       -1 , i)*A(i, i-1)]);
21       E(i-1) = ANS(1);
22       E(i) = ANS(2);
```

```
23       i = i - 2;
24       A = A(1:i, 1:i);
25     endif
26
27   endwhile
28   ANS = roots([1, - (A(i-1,i-1) + A(i,i)), A(i-1, i-1)*A(i,i) - A(i-1 ,
       i)*A(i, i-1)]);
29   E(i-1) = ANS(1);
30   E(i) = ANS(2);
31
32 endfunction
```

3.2.3 Implementación de la matriz

En principio el cómputo de los autovalores se realizó completamente con **matrices full**, que guardan todos los valores de cada matriz. Como Octave realiza multiplicaciones de matrices de manera muy eficiente, ésto no presentó un problema importante con las matrices A hasta que se intentó correr una con la variable m=1000.

Matrices Ralas

Para optimizar el método entonces se decidió utilizar **matrices ralas**, que poseen menor complejidad espacial ya que solo se almacenan los elementos distintos del valor nulo. Particularmente ésto es útil para la resolución de nustro problema ya que tanto A, como Q y R poseen gran cantidad de 0.

La implementación utilizada fue una propia, que constaba de tres vectores:

Vector	Contenido
v1	Filas de los elementos distintos de 0 y size de la matriz M
v2	Columnas de los elementos distintos de 0
v3	Valores de los elementos distintos de 0. El valor en el índice i se encuentra en la fila v1(i) y columna v2(i) de la matriz full

Table 3.1: Representacion matrices ralas.

v2 y v3 poseen el mismo tamaño, que representa la cantidad de elementos no nulos de la matriz. En cambio, v1 incluye un número más referido al tamaño de filas y columnas de la matriz representada.

La transformación de las matrices full a sparse se realizaba de la siguiente manera, aprovechando la función find de octave:

```
1 function [row col v] = full2sparse(A)
2
3 [row,col,v] = find(A);
4
5 row = row';
6 row(size(row)(2)+1)=size(A)(2);
7 col = col';
8 v = v';
9
10 end
```

Comparación matriz full y rala

Al calcular los tiempos de la multiplicación de matrices entre nuestra implementacion sparse con métodos propios y la full utilizando métodos de octave (hablaremos más de esto en el Análisis de Resultados), se llegó a la conclusión de que de la segunda forma era más veloz.

A futuro, si se deseara optimizar el algoritmo se podría implementar una matriz sparse con una implementación de Yale Sparse Matrix o utilizando listas de listas [2].

[2] https://en.wikipedia.org/wiki/Sparse_matrix

Capítulo 4

Análisis de Resultados

A continuación presentaremos una comparación de los tiempos y resultados obtenidos y analizaremos los mismos.

La Tabla 4.1 muestra la comparación de los tiempos en minutos para el cálculo de autovalores entre lo que realiza octave y nuestra propia implementación.

m	eig (m)	custom_eig (m)
10	0.0059503	0.045853
20	0.011867	0.14988
30	0.018568	0.66117
40	0.024468	1.7753
50	0.034035	4.0294
100	0.075321	42.097

Table 4.1: Comparación del cálculo de autovalores de Octave con el cálculo numérico implmentado

La Tabla 4.2 presenta la comparación entre los tiempos de las multiplicaciones de matrices full de octave con nuestra implementación de matrices ralas. Como puede verse, se obtuvieron mejores resultados de la primera manera.

m	A*A (s)	multiplySparseMatrices (s)
10	9.5367e-04	0.085449
50	0.0026703	5.0561
100	0.014328	33.159

Table 4.2: Comparación multiplicación de matrices implementada

En cuanto al límite de tamaño de la matriz que recibe el algoritmo, asumimos que es indeseado un tiempo mayor a cincuenta minutos, por lo que el tamaño máximo sugerido es m = 100.

A continuación se presetan los gráficos realizados a partir de 6 pruebas. Las misma consistieron en la comparación de los autovalores calculados a partir de nuestro algoritmo calculate_eig (representado por x magentas) y el eig de octave (representado por o azules). A lo largo de la secuencia, desde Figure 4.1 hasta 4.6 se puede observar como la forma se sigue manteniendo y se van agregando autovalores reales hacia el lado izquierdo. Si observamos los autovalores graficados, se puede notar cómo los calculados coinciden en la gran mayoría de los casos. Se fue corrigiendo la precisión de las iteraciones del método calculate_eig, para que esto se cumpliera. Originalmente se planteó una cota e-10, pero sucedió que al probar con matrices de m=50, había dos autovalores que no coincidían. Por ese motivo, se terminó aumentando la cota a e-12. Esto implica que puede que para determinados m, hubiese que aumentar la precisión y para otros bastase una cota más baja.

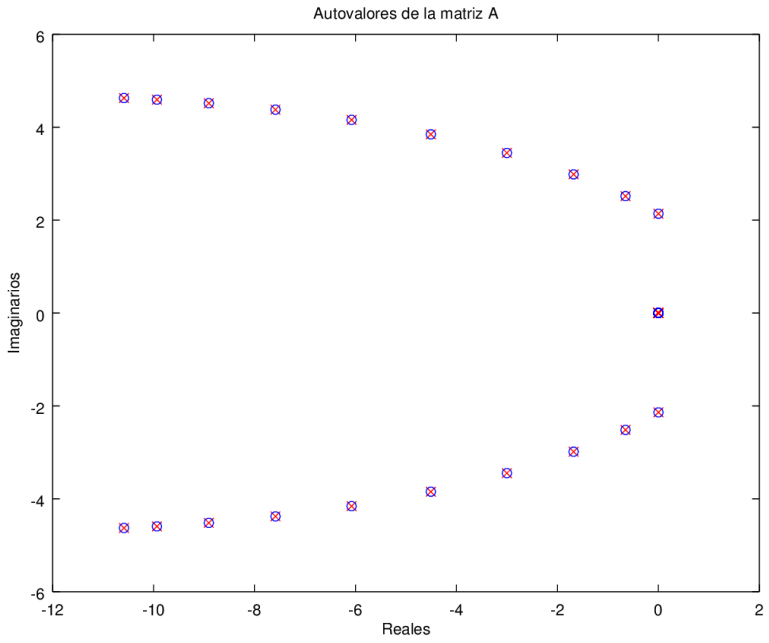

Figure 4.1: Gráfico de los autovalores de la matriz A para $n = 10$

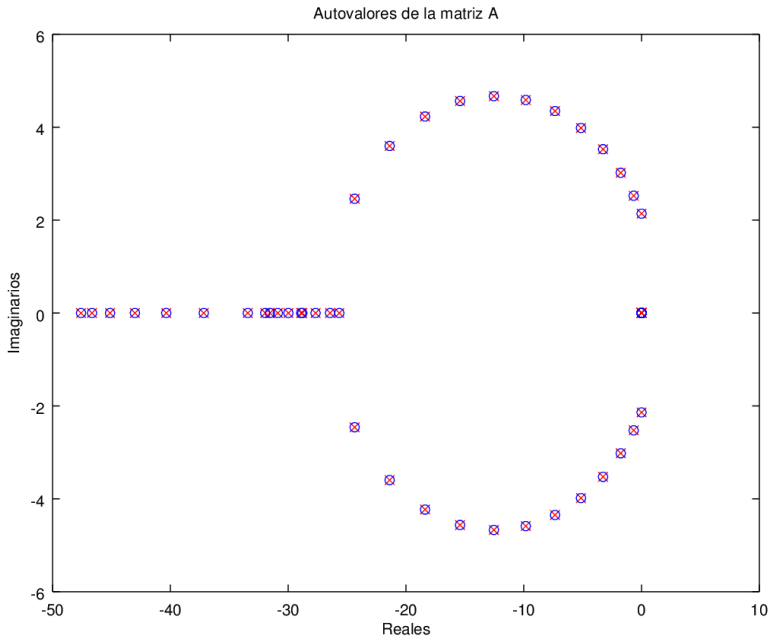

Figure 4.2: Gráfico de los autovalores de la matriz A para $n = 20$

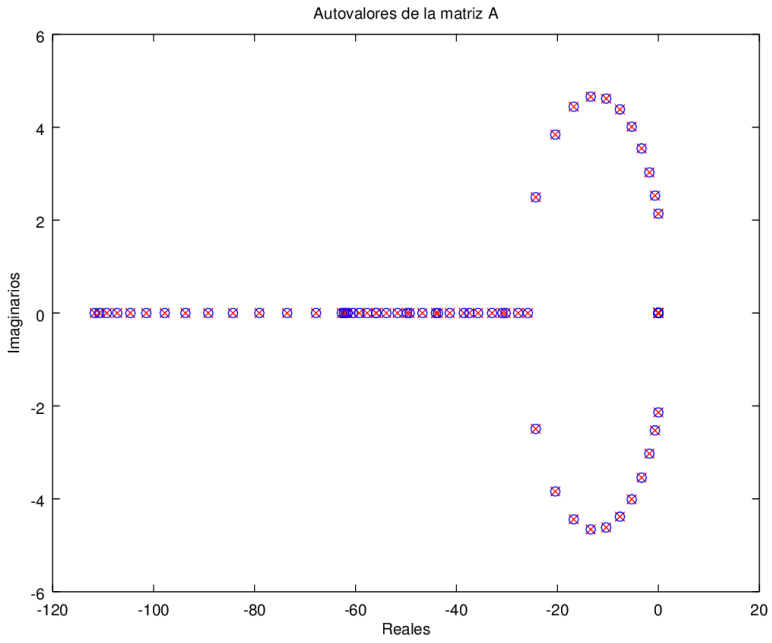

Figure 4.3: Gráfico de los autovalores de la matriz A para $n = 30$

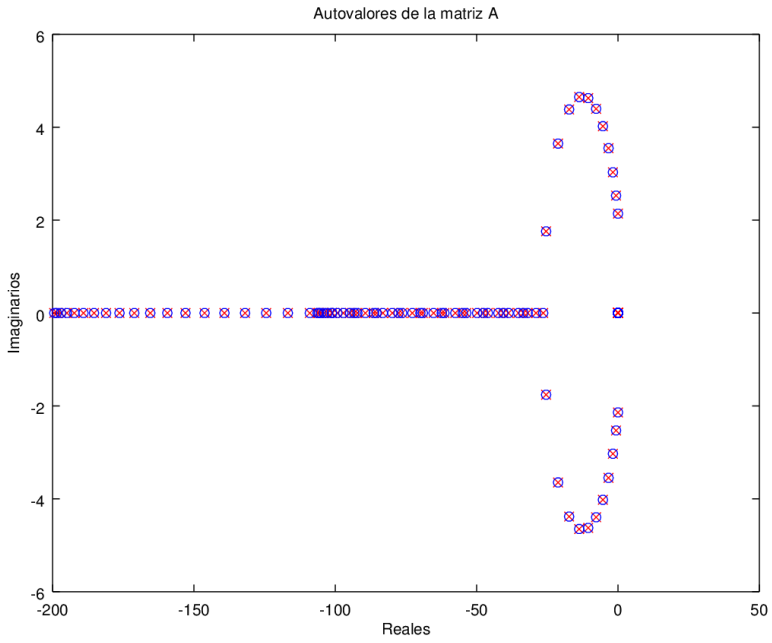

Figure 4.4: Gráfico de los autovalores de la matriz A para $n = 40$

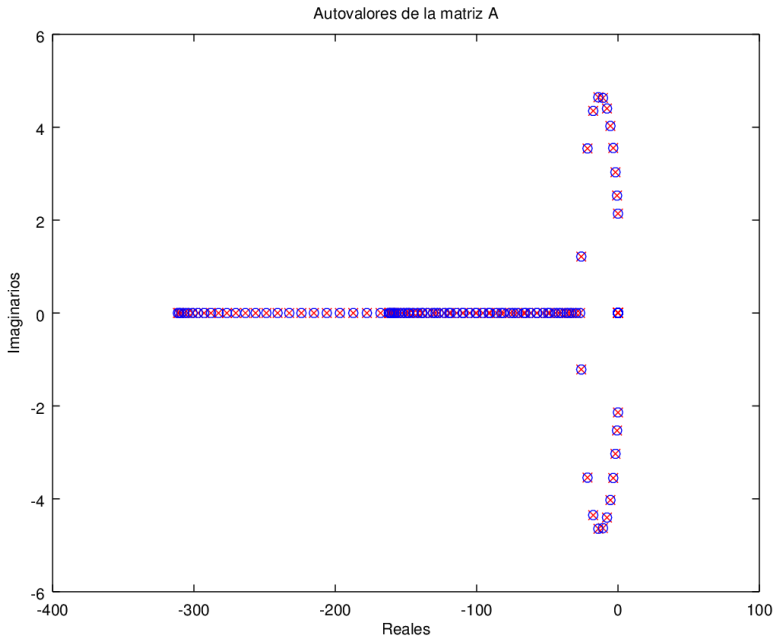

Figure 4.5: Gráfico de los autovalores de la matriz A para $n = 50$

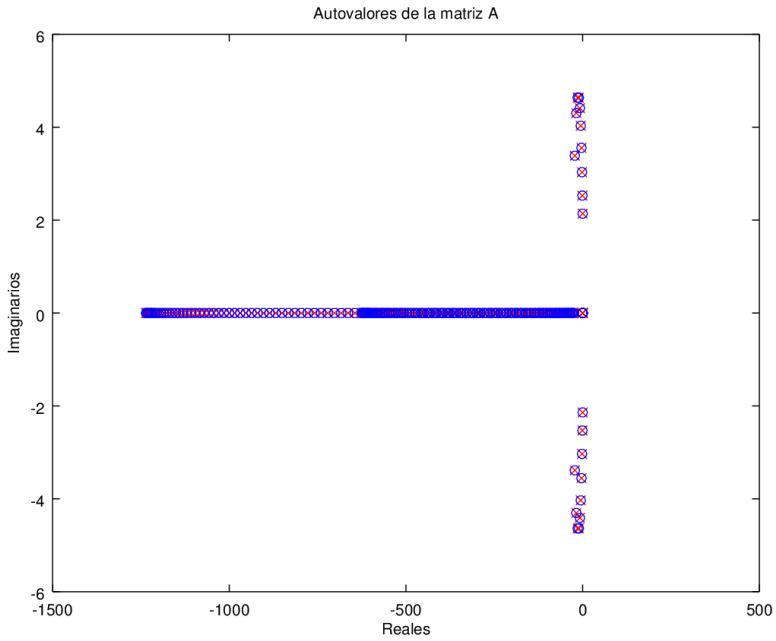

Figure 4.6: Gráfico de los autovalores de la matriz A para $n = 100$

Capítulo 5

Conclusiones

Luego de realizada nuestra implementación para calcular los autovalores complejos de una matriz dados ciertos parámetros y notar que la diferencia de tiempos entre el cálculo analítico y nuestras funciones se acrecentaba cada vez más cuando la matriz aumentaba su dimensión, decidimos investigar en profundidad sobre distintos métodos para realizar esta tarea. Nos dimos cuenta que nuestro problema en particular podía optimizarse con matrices ralas, por lo que se decidió realizar una implementación propia de las mismas. Luego de los análisis y comparaciones concluimos que la implementación que utilizamos no fue eficiente, por lo que se mantuvo el código con las matrices full. Aún así, se siguió intentando mejorar el código y se logró optimizarlo pero no al nivel de la función de cálculo y los métodos de Octave.

Podemos concluir entonces que logramos que nuestro método obtuviera los autovalores correspondientes y, como se mostró anteriormente con los gráficos, todos coincidentes con los autovalores calculados mediante el cálculo analítico. Por ello estamos satisfechos con el logro obtenido a pesar de que hubiesemos deseado poder hacerlo más eficiente.

Finalmente, el presente trabajo nos resultó una interesante manera de ver como los distintos temas de la materia se pueden aplicar en diferentes aspectos de ingeniería, como en el presente caso para la ingeniería química.

Bibliografía

[1] Apuntes tomados en las clases tanto prácticas como teóricas.

[2] Material de la cátedra proporcionado en *http://www.itba.edu.ar/iol*.

[3] Matrix Market, NEP Collection, http://math.nist.gov/MatrixMarket/data/NEP/

[4] Zhaoujun Bai, David Day, James Demmel and Jack Dongarra, *A Test Matrix Collection for Non-Hermitian Eigenvalue Problems*, Release 1.0, October 6, 1996. http://www.cs.ucdavis.edu/ bai/NEP/document/collection.ps.

[5] John H. Mathews, Kurtis D. Fink, *Métodos numéricos con MATLAB*, España, Pearson Prentice Hall, 2008, tercera edición, capítulo 11.

[6] Bindel, *Matrix Computations (CS 6210)*, Fall 2009. http://www.cs.cornell.edu/ bindel/class/cs6210-f09/lec29.pdf