Tarea 2: Método de Bairstow y factorización de Cholesky

Sebastián Valencia Calderón
201111578

Proyecto 1

Ciertas propiedades de una matriz $A \in \mathbb{R}$, pueden ser usadas para mejorar la eficiencia del método de eliminación de Gauss para resolver un sistema del tipo $Ax = b$. En particular, si $A \in \mathbb{R}^{n\times n}$ es una matriz simétrica y positiva definida, el trabajo computacional se reduce a la mitad, lo que representa una mejora de eficiencia dramática. El objetivo de este proyecto, es el de explorar la naturaleza de un algoritmo que explota ambas propiedades de una matriz: que sea simétrica y positiva definida. a continuación, se definen las matrices a usar: una matriz $A \in \mathbb{R}^{n\times n}$ es simétrica si $a_{ij} = a_{ji}\ \forall\ i,\ j\ \in \{1,\ 2,\ \ldots,\ n\}$, es decir $A = A^T$. El conjunto de todas las matrices simétricas es $\mathbb{R}^{n\times n}_{\text{sym}}$. Una matriz es positiva definida si $x^T Ax > 0$ para todo vector $x \in \mathbb{R}^n \setminus \{0\}$. Las matrices positivas definidas, poseen ciertas características que se pueden explotar para resolver sistemas lineales donde ellas estén involucradas. En particular, si $n \geq 2$ y $A \in \mathbb{R}^{n\times n}_{\text{sym}}$, entonces:

- Todos los elementos de la diagonal son positivos.
- Todos los valores propios de A son reales y positivos.
- El determinante de A es positivo.
- $a_{ij}^2 < a_{ii}a_{jj}\ \forall\ i,\ j\ \in \{1,\ 2,\ \ldots,\ n\} \land i \neq j$.

Una matriz simétrica, tiene un conjunto ortonormal de vectores propios, cada uno distinto de cero, además, los valores propios correspondientes son todos reales, por lo que un vector $x \in \mathbb{R}^n \setminus \{0\}$, puede expresarse como una combinación lineal de los vectores propios de A. Si A es simétrica y positiva definida, se puede obtener una factorización $A = LU$, donde $U = L^T$. (estas propiedades, pueden estudiarse más a fondo en la referencia [4]) La deducción del algoritmo, está incluida en las referencias: . Por motivos prácticos, se incluye una generalización del mismo a partir del cálculo de la factorización para una matriz $A \in \mathbb{R}^{4\times 4}$.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} u_{11} & 0 & 0 & 0 \\ u_{21} & u_{22} & 0 & 0 \\ u_{31} & u_{32} & u_{33} & 0 \\ u_{41} & u_{42} & u_{43} & u_{44} \end{bmatrix} \times \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{bmatrix}$$

$$= \begin{bmatrix} u_{11}^2 & u_{11}u_{12} & u_{11}u_{13} & u_{11}u_{14} \\ u_{12}u_{11} & u_{12}^2 + u_{22}^2 & u_{12}u_{13} + u_{22}u_{23} & u_{12}u_{14} + u_{22}u_{24} \\ u_{13}u_{11} & u_{13}u_{12} + u_{23}u_{22} & u_{13}^2 + u_{23}^3 + u_{33}^2 & u_{13}u_{14} + u_{23}u_{24} + u_{33}u_{34} \\ u_{13}u_{11} & u_{14}u_{12} + u_{24}u_{22} & u_{14}u_{13} + u_{24}u_{23} + u_{34}u_{33} & u_{14}^2 + u_{24}^2 + u_{34}^2 + u_{44}^2 \end{bmatrix}$$

- **Algoritmo.** Cálculos algebraicos basados en la igualdad de matrices y la multiplicación matricial, y en los cálculos realizados para la matriz de dimensión 4×4 (ilustración original en la referencia [3]), sugieren lo siguiente:

$$u_{kk} = \sqrt{a_{kk} - \sum_{i=1}^{k-1} (u_{ik}^2)} \quad u_{km} = \frac{\left[a_{km} - \sum_{i=1}^{k-1} (u_{im} u_{ik})\right]}{u_{kk}} \quad \begin{cases} k \in \{1, \dots, N\} \\ m \in \{k+1, \dots, N\} \end{cases}$$

Una primera aproximación sencilla al algoritmo dadas las fórmulas obtenidas por medio de la generalización, se muestra a continuación:

Algorithm 1 Cholesky factorization based on formulas above

Require: $A \in \mathbb{R}^{n\times n}_{\text{sym}} \wedge A$ is positive definite
Ensure: L is lower triangular $\wedge\, L^T L = A$

```
procedure CHOLESKY
    L ∈ R^{n×n}
    for k ← 1 ... n do
        L_kk ← sqrt(a_kk − Σ_{s=1}^{k−1} (l_ks^2))
        for i ← (k+1) ... n do
            L_ik ← [a_ik − Σ_{s=1}^{k−1} (u_is u_ks)] / l_kk
    return L
```

Para aproximar el algoritmo 1, a una implementación sobre una máquina digital usando un lenguaje de programación real, se puede reescribir de esta forma (considerando error en la forma de la matriz).

Algorithm 2 Cholesky factorization for real implementation

Require: $A \in \mathbb{R}^{n\times n}$
Ensure: L is lower triangular $\wedge\, L^T L = A$ iff A is positive definite.

```
procedure CHOLESKY
    for i ← 1 ... n do
        t ← a_ii − Σ_{j=1}^{i−1} l_ji^2
        if t ≤ 0 then
            A is not positive definite  return error
        l_ii ← sqrt(t)
        for j ← (i+1) ... n do
            l_ij ← (a_ij − Σ_{k=1}^{i−1} l_ki l_kj) / l_ii
    return L
```

El último algoritmo, puede representarse fácilmente en un lenguaje de programación como MATLAB, basta con representar las sumatorias como productos puntos y ya está. El manejo de error, también hace propicio la implementación del algoritmo 2, en el lenguaje de programación, además, constituye una buena prueba para verificar si una matriz es positiva definida. A continuación, se incluye el código implementado en MATLAB, y la validación del mismo mediante el uso de algunas matrices de dimensión variable.

- **Implementación.** La implementación del algoritmo 1, en una máquina digital por medio de un lenguaje de programación de alto nivel como MATLAB, requiere contemplar la verificación de la positividad de la matriz A. Esta verificación, surge de manera natural con el diseño del algoritmo 2, el cual materializa la verificación, por medio de una sencilla manipulación a las fórmulas derivadas anteriormente. La única función pues necesaria para la realización de la factorización de Cholesky de una matriz es la siguiente:

Script 1: Factorizacion de Cholesky, desarrollo del algoritmo 2.

```
function L = cholesky(A)
% given a matrix, it tries hard to find the Cholesky decomposition of a
% matrix, iff the matrix is positive definite, otherwise, it fails and
% return an error.

%   Arguments:
%       A: a square matrix

%   USAGE:
%   L = cholesky(A);
%   assert L' * L = A

% Written by: Sebastian Valencia, Universidad de los Andes, 2016

    n = size(A, 1);         % matrix dimension
    L = zeros(n, n);        % allocating space for the lower triangular matrix

    for i=1:n
        % test if the matrix is positive definite
        tmp = A(i, i) - L(1:(i - 1), i)'* L(1:(i - 1), i);
        % if fail the test, report it immediately
        if(tmp <= 0)
            error('A should be a positive definite matrix');
        end;
        L(i, i) = sqrt(tmp); % the diagonal, as computed in the derivation
        % popoulate L(i, j) where i <> j
        for j = (i+1):n
            sum = L(1:(i - 1), i)' * L(1:(i - 1), j);
            L(i, j) = (A(i, j) - sum) / L(i, i);
        end;
    end;
end
```

- **Validación.** Para validar la implementación, se propone además de lo pedido, tomar ejemplos disponibles en la bibliografía (es enspecífico, las referencias [5], [7], y [8]), comparar el resultado expuesta en la misma con la ejecución de la factorización programada. A continuación, se muestra un ejemplo de dicha factorización presente en la referencia [7], además, se contrasta esto con la ejecución del código escrito sobre la representación de la misma matriz.

  Ejemplo:

$$A = \begin{bmatrix} 4 & -2 & 4 & 2 \\ -2 & 10 & -2 & -7 \\ 4 & -2 & 8 & 4 \\ 2 & -7 & 4 & 7 \end{bmatrix} \Rightarrow L = \begin{bmatrix} 2 & -1 & 2 & 1 \\ 0 & 3 & 0 & -2 \\ 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

  Le ejecución de estas lineas:

```
A = [4 -2 4 2 ; -2 10 -2 -7 ; 4 -2 8 4 ; 2 -7 4 7];
L = cholesky(A);
```

  Da como resultado:

$$L = \begin{bmatrix} 2 & -1 & 2 & 1 \\ 0 & 3 & 0 & -2 \\ 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

  Ahora, se valida haciendo uso de una matriz tridiagonal con valor 4 en la diagonal principal, y -1 en las diagonales adyacentes. Con esta descripción, se generan matrices de tamaño 10, 20 y 30, se calcula la factorización de Cholesky con el método implementado `cholesky`, y con la función de MATLAB `chol`, se comparan ambos resultados ($\left\| L^T \times T - A \right\|_\infty$) a partir de su efectividad. Para esto, se tiene el siguiente script que genera la matriz según su tamaño:

```
function A = genmatrix(n)
    A = diag(4*ones(1,n),0)+diag(-1*ones(1,n-1),1)+diag(-1*ones(1,n-1)
        ,-1);
end
```

  Por ejemplo, `genmatrix(5)` da como resultado:

$$A = \begin{bmatrix} 4 & -1 & 0 & 0 & 0 \\ -1 & 4 & -1 & 0 & 0 \\ 0 & -1 & 4 & -1 & 0 \\ 0 & 0 & -1 & 4 & -1 \\ 0 & 0 & 0 & -1 & 4 \end{bmatrix}$$

Ahora, se ejecuta el siguiente script para validar la ejecución de la función implementada y contrastar sus resultados con los de la librería estándar de MATLAB. De esta manera, se obtiene una comparación entre el comportamiento y desempeño numérico de ambas funciones.

```
% Compare cholesky() vs chol() on matrices of dimension 10, 20, 30

ns = [10 20 30]; iter = size(ns, 2);
merrors = zeros(1, iter); oerrors = zeros(1, iter);

i = 1;
while i <= iter
    A = genmatrix(ns(i)); LM = chol(A); LO = cholesky(A);
    em = norm(LM' * LM - A); om = norm(LO' * LO - A);
    merrors(i) = em; oerrors(i) = om;
    i = i + 1;
end;
```

Al graficar los resultados obtenidos con el anterior script, se obtiene lo siguiente:

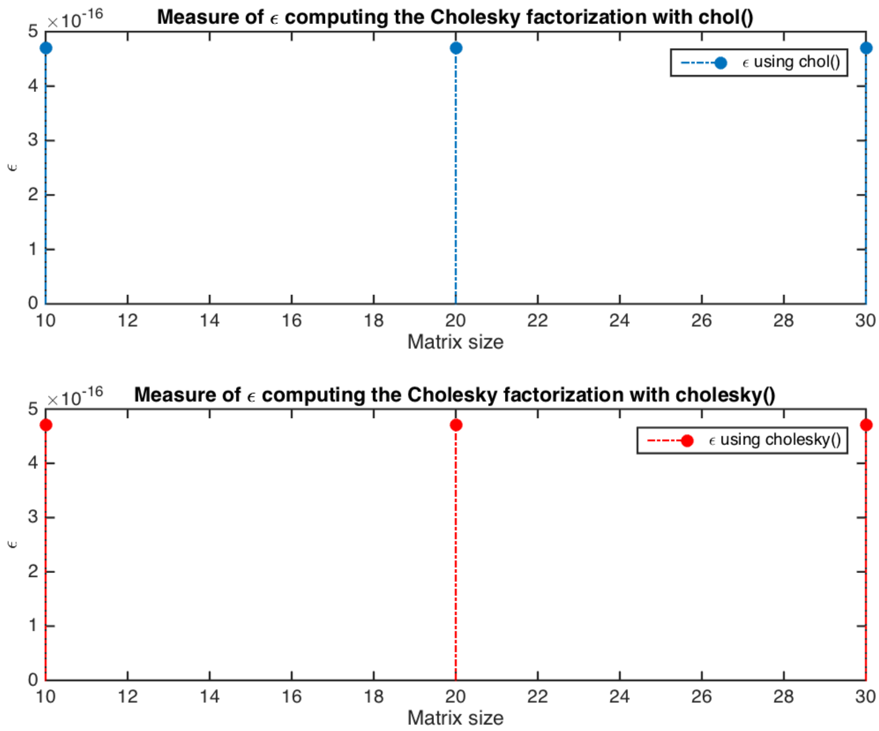

Figura. 1: Comparación de las dos funciones para factorizar una matriz positiva definida, la métrica de error, está dada por la expresión $\left\|L^T \times L - A\right\|$. El desempeño y comportamiento de ambas funciones es el mismo.

Proyecto 2

Un polinomio, es una estructura algebraica o ecuación de la forma:

$$P_n(x) \ : \ a_0x^n + a_1x^{n-1} + a_2x^{n-2} + \cdots + a_kx^{n-k} + \cdots + a_{n-1}x + a_n = 0$$

Donde los coeficientes a_i, son números específicos en $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, o $\mathbb{C}$. Para resolver este tipo de ecuaciones, es necesario hallar otro número puntual x que satisfaga la ecuación. Es decir, dado un polinomio $P_n(x)$, solucionar el polinomio, es encontrar el conjunto X:

$$X = \{x \in \mathbb{C} \mid P_n(x) = 0\}$$

El número n, es denominado orden del polinomio, este determina la cardinalidad de X, de manera más específica, n, o $|X|$, es el número de raíces del polinomio, es decir, cada uno de los elementos pertenecientes a X, los cuales, por definición satisfacen $P_n(X) = 0$. Conocer de antemano la cantidad de raíces de un polinomio, es un hecho matemático de fundamental importancia para la solución computacional del polinomio. Otra deducción matemática fundamental para esta tarea, es el teorema fundamental del álgebra, el cual, se presta para deducir que si un numero complejo $a+ib$ pertenece a X, entonces el numero complejo $a - ib$, también lo hace.

Los polinomios, son una estructura fundamental en la computación científica, al igual que las matrices, muchos problemas de computación científica, pueden reducirse a ecuaciones no lineales polinomiales. Por esta razón, es importante el diseño de algoritmos numéricos para la solución de este tipo de ecuaciones. Si $n \leq 3$, el problema es trivial, en el caso de $n = 1$, el problema se resuelve con dos multiplicaciones y una división. Si $n = 2$, existe la famosa fórmula para hallar las dos raíces. Si $n = 3$, se puede recurrir a las fórmulas de Cardano. Para grados mayores a 3, existe una dificultad inherente a las estructuras matemáticas comprometidas en la solución y estructura de los polinomios (los artilugios del álgebra abstracta). Para mitigar esto, se presenta un método para la solución numérica de un polinomio $P_n(x)$. A continuación, se presenta un método para la solución no analítica de un polinomio (La exposición referida, es presentada de manera formal en la referencia [2]. Una derivación alternativa y más cercana al cálculo matricial, es presentada en la referencia [1]).

Un polinomio de la forma $P_n(x) = \sum_{i=0}^{n} a_ix^{n-i}$, $(a_0 = 1)$, puede escribirse se la forma:

$$(x^2 + px + q)(x^{n-2} + b_1x^{n-3} + b_2x^{n-4} + \cdots + b_kx^{n-2-k} + \ldots b_{n-3}x + b_{n-2}) + Rx + S = 0$$

En la anterior expresión, $Rx + S$, es el residuo lineal, el cual, se debe iterativamente reducir para lograr una igualdad entre esta última expresión y el polinomio original. En este caso, el polinomio original $P_n(x)$, es divisible perfectamente con $(x^2 + px + q)$. La manipulación algebraica, es suficiente para relacionar los coeficientes p, q, $b1$, $b2$, b_3, $\ldots$, b_{n-2}, R, y S con los coeficientes del polinomio normalizado (aquel polinomio donde $a_0 = 1$) . A continuación, se muestra la relación:

$$
\begin{aligned}
b_1 &= a_1 - p \\
b_2 &= a_2 - pb_1 - q \\
b_i &= a_i - pb_{i-1} - qb_{i-2} \quad i = 1 \ldots (n-2) \\
R &= a_{n-1} - pb_{n-2} - qb_{n-3} \\
S &= a_n - qb_{n-2}
\end{aligned}
$$

Si $R = 0 \ \wedge \ S = 0$, las raíces del término cuadrático $(x^2 + px + q)$, son también raíces del polinomio original $P_n(x)$. Estas raíces, son fácilmente obtenidas mediante el uso de la fórmula cuadrática. Ahora, es necesario hallar coeficientes $p,\ q,\ b1,\ b2,\ b_3,\ \ldots,\ b_{n-2}$ para satisfacer la proposición $R = 0 \ \wedge \ S = 0$, de esta manera, a partir de la expresión cuadrática obtenida, se pueden hallar dos raíces de $P_n(x)$. El polinomio, en la reducción de los términos hasta lograr $R = 0 \wedge S = 0$, queda agrupado así $P_n(x) = (x^2+px+q)P_{n-2}(x)$. Es necesario, reducir ahora $P_{n-2}(x)$ de la misma forma hasta lograr que el polinomio que acompaña el término cuadrático sea fácil de resolver (usando la fórmula de Cardano o la fórmula cuadrática).

El método de Bairstow, usa aproximaciones de p y de q haciendo uso del método de Newton:

$$
\begin{aligned}
p^{(n+1)} &= p^{(n)} - \frac{\left(R \times \frac{\partial S}{\partial q} - S \times \frac{\partial R}{\partial q}\right)}{J} \\
q^{(n+1)} &= q^{(n)} + \frac{\left(R \times \frac{\partial S}{\partial p} - S \times \frac{\partial R}{\partial p}\right)}{J} \\
\textbf{Donde } J &= \left[\frac{\partial R}{\partial p}\frac{\partial S}{\partial q} - \frac{\partial S}{\partial p}\frac{\partial R}{\partial q}\right]
\end{aligned}
$$

Dado que R y S son funciones de $b1,\ b2,\ b_3,\ \ldots,\ b_{n-2}$, los cuales a su vez dependen de p y de q, es necesario hallar las derivaras parciales respecto a p y a q de los coeficientes $b1,\ b2,\ b_3,\ \ldots,\ b_{n-2},\ R,$ y S. A continuación, se halla cada derivada parcial:

$$
\begin{aligned}
\nabla b_1(p,q) &= [c_1, d_1] = [-1, 0] \\
\nabla b_2(p,q) &= [c_2, d_2] = \left[-b_1 - p\left(\frac{\partial b_1}{\partial p}\right), -p\left(\frac{\partial b_1}{\partial q}\right) - 1\right] = [-b_1 + p, -1] \\
\nabla b_i(p,q) &= [c_i, d_i] = \left[-b_{i-1} - p\left(\frac{\partial b_{i-1}}{\partial p}\right) - q\left(\frac{\partial b_{i-2}}{\partial p}\right), -p\left(\frac{\partial b_{i-1}}{\partial q}\right) - b_{i-2} - q\left(\frac{\partial b_{i-2}}{\partial q}\right)\right] \\
&= [-b_1 + p, -1] \quad i = 3 \ldots (n-2)
\end{aligned}
$$

$$
\begin{aligned}
\nabla R(p,q) &= \left[-b_{n-2} - p\left(\frac{\partial b_{n-2}}{\partial p}\right) - q\left(\frac{\partial b_{n-3}}{\partial p}\right), -p\left(\frac{\partial b_{n-2}}{\partial q}\right) - b_{n-3} - q\left(\frac{\partial b_{n-3}}{\partial q}\right)\right] \\
\nabla S(p,q) &= \left[-q\left(\frac{\partial b_{n-2}}{\partial p}\right), -b_{n-2} - q\left(\frac{\partial b_{n-2}}{\partial q}\right)\right]
\end{aligned}
$$

Haciendo uso de los coeficientes c_i y d_i, es posible reescribir los términos $\nabla R(p,q)$, $\nabla S(p,q)$, $\nabla b_{i=1\ldots(n-2)}(p,q)$ de la siguiente manera:

$$
\begin{aligned}
\nabla b_1(p,q) &= [c_1, d_1] = [-1, 0] \\
\nabla b_2(p,q) &= [c_2, d_2] = [-b_1 + p, -1] \\
\nabla b_i(p,q) &= [c_i, d_i] = [-b_{i-1} - pc_{i-1} - qc_{i-2}, -b_{i-2} - pd_{i-1} - qd_{i-2}] \quad i = 3\ldots(n-2) \\
\nabla R(p,q) &= [-b_{n-2} - pc_{n-2} - qc_{n-3}, -b_{n-3} - pd_{n-2} - qd_{n-3}] \\
\nabla S(p,q) &= [-qc_{n-2}, -b_{n-2} - qd_{n-2}]
\end{aligned}
$$

Las formulas escritas anteriormente, aplican para ecuaciones polinomiales de grados estrictamente mayores a 3. Para ecuaciones de grados menores, se recurre a la solución trivial.

Algorithm 3

Require: $n \geq 4 \ \wedge \ P_n(x) : \sum_{i=0}^{n} a_i x^{n-i}$, $a_0 = 1$
Ensure: $P_n(x) = (x^2 + px + q)(x^{n-2} + b_1 x^{n-3} + \cdots + b_k x^{n-2-k} + \ldots b_{n-3}x + b_{n-2})$

procedure BAIRSTOW
  $p, \ q \leftarrow 0, \ 0$
  $n \leftarrow$ Degree of $P_n(x)$
  $R, \ S \leftarrow \infty, \ \infty$
  $b, \ c, \ d$: Are polynomials of degree $(n-2)$
  while $\epsilon < |R + S|$ **do**
    $b_1, \ b_2 \leftarrow a_1 - p, \ a_2 - pb_1 - q$
    $c_1, \ c_2 \leftarrow -1, \ -b_1 + p$
    $d_1, \ d_2 \leftarrow 0, -1$
    for $i \in \{3, \ \ldots \ , \ n-2\}$ **do**
      $b_i \leftarrow a_i - pb_{i-1} - qb_{i-2}$
      $c_i, \ d_i \leftarrow -b_{i-1} - pc_{i-1} - qc_{i-2}, -b_{i-2} - pd_{i-1} - qd_{i-2}$
    $R, \ S \leftarrow a_1 - p, \ a_2 - pb_1 - q$
    $\nabla R(p,q) \leftarrow [-b_{n-2} - pc_{n-2} - qc_{n-3}, \ -b_{n-3} - pd_{n-2} - qd_{n-3}]$
    $\nabla S(p,q) \leftarrow [-qc_{n-2}, \ -b_{n-2} - qd_{n-2}]$
    $J \leftarrow \left[\frac{\partial R}{\partial p}\frac{\partial S}{\partial q} - \frac{\partial S}{\partial p}\frac{\partial R}{\partial q}\right]$
    $p, \ q \leftarrow p - \frac{\left(R \times \frac{\partial S}{\partial q} - S \times \frac{\partial R}{\partial q}\right)}{J}, \ q + \frac{\left(R \times \frac{\partial S}{\partial p} - S \times \frac{\partial R}{\partial p}\right)}{J}$
  return $p, \ q, \ x^{n-2} + b_1 x^{n-3} + b_2 x^{n-4} + \cdots + b_k x^{n-2-k} + \ldots b_{n-3}x + b_{n-2}$

- **Algoritmo.** El aparato teórico expuesto anteriormente, permite plantear un algoritmo para hallar la factorización de un polinomio $P_n(x)$ donde $n \geq 4$. Es decir, pado un polinomio $P_n(x)$, se pretende diseñar un algoritmo que calcule p, q, términos necesarios para el término cuadrático, y b_i, que garanticen que R y S, se encuentren en un radio muy pequeño cuyo centro es cero. El algoritmo de Bairstow, es usado para hallar la factorización cuadrática de un polinomio $P_n(x)$. Dicha factorización es: $(x^2+px+q)(x^{n-2}+b_1x^{n-3}+b_2x^{n-4}+\cdots+b_kx^{n-2-k}+\ldots b_{n-3}x+b_{n-2})+Rx+S$, tal que R y S sean tan pequeños como se requiera (computacionalmente hablando), dependiendo de un error ϵ.

El algoritmo 3, asigna los valores pertinentes (los cuales fueron deducidos anteriormente) a los coeficientes $b_1,\ b_2,\ \dots\ ,\ b_i$, y finalmente a p y a q según las igualdades expuestas en la deducción matemática. Este proceso, se repite hasta que $\epsilon \geq |R+S|$. Cuando esto se cumpla, se cuenta con valores de p, q, b_i, para formar la expresión $(x^2 + px + q)(x^{n-2} + b_1x^{n-3} + \cdots + b_kx^{n-2-k} + \dots b_{n-3}x + b_{n-2})$, que sea igual a $P_n(x)$. Con esta factorización, se obtienen las dos primeras raíces del polinomio original $P_n(x)$:

$$x^{(1)} = \frac{-p + \sqrt{p^2 - 4q}}{2} \qquad x^{(2)} = \frac{-p - \sqrt{p^2 - 4q}}{2}$$

Se puede aplicar de nuevo el algoritmo 3, sobre el polinomio $x^{n-2} + b_1x^{n-3} + \cdots + b_kx^{n-2-k} + \dots b_{n-3}x + b_{n-2}$, para obtener las raíces $x^{(3)}$, y $x^{(4)}$, así, de manera sucesiva, se obtienen raíces hasta que la expresión $x^{n-2} + b_1x^{n-3} + \cdots + b_kx^{n-2-k} + \dots b_{n-3}x + b_{n-2}$, tenga un grado menor a 4. En cuyo caso, se debe aplicar el método trivial. Esta aplicación repetida del algoritmo 3, sugiere el siguiente procedimiento para hallar todas las raíces de un polinomio.

Algorithm 4

Require: $P_n(x) : \sum_{i=0}^{n} a_ix^{n-i}\ ,\ a_0 \neq 0$
Ensure: $X = \{x \in \mathbb{C} \mid P_n(x) = 0\}$
1: **procedure** PolynomialRoots
2: $n \leftarrow$ Degree of $P_n(x)$
3: $X \leftarrow \{\}$
4: **if** $n \leq 3$ **then**
5: Solve it using algebraic methods (quadratic formula, Cardano's method).
6: Direct algebraic solution.
7: $X \leftarrow X \cup \textsc{Solve}(P_n(x))$
8: **else**
9: $P_n(x) \leftarrow \sum_{i=0}^{n} \left(\frac{a_i}{a_0}\right) x^{n-i}$
10: **while** $n > 3$ **do**
11: $p,\ q,\ P_n(x) \leftarrow \textsc{Bairstow}(P_n(x),\ \epsilon)$
12: $X \leftarrow X \cup \textsc{Solve}(x^2 + px + q)$
13: $n \leftarrow$ Degree of $P_n(x)$
14: $X \leftarrow X \cup \textsc{Solve}(P_n(x))$
return X

El algoritmo 4, dependiendo del grado del polinomio, resuelve este de manera algebraica o de manera numérica. Si $n \leq 3$, entonces lo hace de la primera forma (aplicando la fórmula cuadrática o la fórmula de Cardano), de lo contrario, normaliza el polinomio (linea 9 del algoritmo 4), de tal forma que $a_0 = 1$. Posteriormente, se aplica iterativamente (reduciendo el grado a $i-2$ por cada iteración) el método de Bairstow (algoritmo 3), para obtener la factorización de cada polinomio. La linea 11 de este algoritmo 4, calcula p, q, y el nuevo valor del polinomio. Con los dos primeros términos (p y q), se hallan dos raíces más del polinomio original (haciendo uso de la formula cuadrática), y finalmente, se agregan al conjunto de solución X. Después del proceso iterativo, por un hecho expuesto más adelante, se cuenta con un polinomio de grado menor a 4, por lo que las raíces de este, se calculan de manera algebraica, a través de la función SOLVE, el cual para polinomios de grado 1 debe despejar x, para polinomios de grado 2 debe calcular X haciendo uso de la fórmula cuadrática, y para polinomios cúbicos, debe usar el método de Cardano. Esta sugerencia, se adopta de la referencia [2].

Definición: Sea $\mathbb{N}_e = \{n \in \mathbb{N}^+ \mid \exists\ k \in \mathbb{N}^+\ ;\ n = 2k\}$, sea $\mathbb{N}_o = \mathbb{N}^+ \setminus \mathbb{N}_e$.

Dado $P_n(x)$, si n es par, $n \in \mathbb{N}_e$. Por el lema de Zorn, se sabe que n está contenido en un conjunto $E = \{2,\ 4,\ \ldots,\ n,\ 2(n+1)\}$. De manera equivalente, si $n \in \mathbb{N}_o$, $\exists\ O \subset \mathbb{N}_o$ tal que $O = \{1,\ 3,\ \ldots,\ n,\ 2n+1\}$. Sea $pred_A$ una función con tipo $pred\ :\ A \setminus \{\min A\} \to A$, que para cada elemento en $A \setminus \{\min A\}$, retorne el predecesor absoluto bajo una ordenación isomorfa a la $\mathbb{N}$. Es decir, $pred_E\ :\ E \setminus \{2\} \to E$, retorna el par anterior a n. Una construcción análoga para los números impares, permite deducir que la aplicación finita y sucesiva de $pred$ a algún n sobre el conjunto al cual pertenezca (E, u O), terminara en 2 o en 3, dependiendo si n es par o impar respectivamente.

La anterior deducción, aunque un tanto formal, es útil para demostrar que al salir del loop de la linea 10 del algoritmo 4, $P_n(x)$, sera tal que $n \leq 3$, la aplicación sucesiva descendiente de $pred$, mas que la condición de parada, sirven para demostrar la terminación del algoritmo en cuestión. Suponiendo que el algoritmo 3, termina, se ha diseñado un algoritmo capaz de hallar con cierto error o tolerancia, las raíces de un polinomio de grado arbitrario.

- **Implementación.** La implementación del algoritmo 4, en una máquina digital por medio de un lenguaje de programación de alto nivel como MATLAB, requiere contemplar el desarrollo de la formula cuadrática y del método de Cardano para la solución algebraica de polinomios de grados que así lo permitan (Galois y Abel). A continuación, se muestran documentados los archivos de MATLAB, que permiten la materialización del algoritmo 4, se muestran, de orden de relevancia, es decir, se muestra de ultimo el algoritmo final, antes de este, se muestran las funciones necesarias para su realización.

Script 2: Obtención de raíces de un caso trivial: un polinomio cuadrático.

```
function x = quadratic(polynomial)
% fetch the roots of a second degree polynomial over the complex field:
% size(polynomial) = 3
% let pol be polynomial in
% pol(1) != 0 /\ pol(1)x^2 + pol(2)x + pol(3) = 0

%   Arguments:
%       polynomial: A complex valued quadratic polynomial [a b c]

%   USAGE:
%   x = quadratic(polynomial)
%   x(1) /\ x(2) are the unique values that satisfies:
%   forall x in [x(1), x(2)] polyval(polynomial, x) = 0
```

```

% Written by: Sebastian Valencia, Universidad de los Andes, 2016

    % Coefficients
    a = polynomial(1); b = polynomial(2); c = polynomial(3);

    % Roots given by the quadratic formula
    x(1) = (-b + sqrt(b^2 - 4 * a * c)) / (2 * a);
    x(2) = (-b - sqrt(b^2 - 4 * a * c)) / (2 * a);

end
```

Script 3: Fórmulas de Cardano para la transformación y posterior solución de ecuaciones de tercer grado.

```
function x = cardano(polynomial)
% fetch the roots of a cubic polynomial over the real field.
% size(polynomial) = 4
% let pol be polynomial in
% pol(1) != 0 /\ pol(1)x^3 + pol(2)x^2 + pol(3)x + pol(4) = 0

%   Arguments:
%       polynomial: A complex valued quadratic polynomial [a b c]

%   USAGE:
%   x = cardano(polynomial)
%   x(1) /\ x(2) /\ x(3) are the unique values that satisfies:
%   forall x in [x(1), x(2), x(3)] polyval(polynomial, x) = 0

% Written by: Nam Sun Wang, University or?f Maryland, College Park, 2006c
% Taken from: https://raw.githubusercontent.com/LCAV/edmbox/master/cubicfcnroots.m
% Formula used are given in Tuma, Engineering Mathematics Handbook

    % Coefficients
    a = polynomial(1); b = polynomial(2);
    c = polynomial(3); d = polynomial(4);

    p  = c/a - b*b/a/a/3. ;
    q  = (2.*b*b*b/a/a/a - 9.*b*c/a/a + 27.*d/a) / 27. ;

    DD = p*p*p/27. + q*q/4. ;

    if (DD < 0.)
        phi = acos(-q/2./sqrt(abs(p*p*p)/27.));
        temp1=2.*sqrt(abs(p)/3.);
```

```
        y1 =  temp1*cos(phi/3.); y2 = -temp1*cos((phi+pi)/3.);
        y3 = -temp1*cos((phi-pi)/3.);
    else
        temp1 = -q/2. + sqrt(DD);
        temp2 = -q/2. - sqrt(DD);
        u = abs(temp1)^(1./3.); v = abs(temp2)^(1./3.);
        if (temp1 < 0.) u = -u; end
        if (temp2 < 0.) v=-v; end
        y1  = u + v; y2r = -(u+v)/2.; y2i =  (u-v)*sqrt(3.)/2.;
    end

    temp1 = b/a/3.;
    y1 = y1-temp1;
    if (DD < 0.) y2 = y2-temp1; y3 = y3-temp1; else y2r=y2r-temp1; end

    if (DD < 0.)
        x(1) = y1; x(2) = y2; x(3) = y3;
    elseif (DD == 0.)
        x(1) =  y1; x(2) = y2r; x(3) = y2r;
    else
        x(1) = y1; x(2) = y2r + y2i*i; x(3) = y2r - y2i*i;
    end;
end
```

Script 4: Obtención de la solución algebraica, dependiendo el orden o grado del polinomio dado como parámetro. Esta función únicamente resuelve el polinomio si $n \leq 3$.

```
function x = basecase(polynomial)
% fetch the roots of a polynomial of degree <= 3, by using auxiliary
% methods (quadratic equation formula, Cardano fomulae)
% size(polynomial) <= 3
% let pol be polynomial in
% pol(1) != 0 /\ polyval(polynomial, x) = 0

%   Arguments:
%       polynomial: A real valued quadratic polynomial [a b c d?]
%       if d is give, it solves a cubic equation, otherwise, it solves a
%       quadratic equation.

%   USAGE:
%   x = basecase(polynomial)
%   polyval(polynomial, x) = 0

% Written by: Sebastian Valencia, Universidad de los Andes, 2016
    n = size(polynomial, 2) - 1; % The degree
```

```
    x = zeros(1, n); % The desired roots
    if(n == 2)
        x = quadratic(polynomial); % Solve it by quadratic if n = 2
    elseif(n == 3)
        x = cardano(polynomial); % Solve it using Cardano method, if n = 3
    end;
end
```

Script 5: Obtención de una versión equivalente a un polinomio, pero con el coeficiente que define el grado, puesto en 1.

```
function polynomial = normalized(pol)
% returns a normalized version of the polynoial, that is, an equivalent
% polynomial with same roots, same degree, such that pol(1) = 1

%   Arguments:
%       pol: A real valued polynomial of any degree

%   USAGE:
%   polynomial2 = normalized(polynomial1)
%   polynomial2(1) = 1 /\ roots(polynomial1) = roots(polynomial2)
%   forall x in R, then polynomial1(x) = polynomial2(x)

% Written by: Sebastian Valencia, Universidad de los Andes, 2016

    n = size(pol, 2);           % The degree
    polynomial = zeros(1, n);   % Allocating space for the new version of
                                % the polynomil
    coef = pol(1);              % The coefficient to perform division with

    for i=1:n polynomial(i) = pol(i) / coef; end;
end
```

Script 6: Algoritmo de Bairstow para calcular la factorización cuadrática de un polinomio de grado mayor a 3. Es una implementación del algoritmo 3. El término cuadrático de la factorización, determina dos de las raíces del polinomio dado por parámetro.

```
function [p, q, b] = bairstow(polynomial, error)
% find the roots of arbitrary degree polynomials using the Bairstow method,
% the desired polynomial to solve, it's related to polynomial, while the
% tolerance, is attached to the error parameter. Given a polynomial, this
% function finds a factorization determined by p, q and a normalized
% polynomial of degree (n - 2), whose coefficients are in b.
% polynomial = (x^2 + px + q)(x^(n-2) + b(1)x^(n - 2 - 1) + ... + b(n - 2))
% So the roots of (x^2 + px + q), are roots of polynomial

%   Arguments:
%       polynomial: A real valued polynomial, whose degree is determined by
%       the size of this matrix.
%       error: The tolerance required for iterating changing the paramters
%       p, q, and b

%   USAGE:
%   [p, q, b] = bairstow(polynomial, error)
%   (x^2 + px + q)(b(x)) = polyomial + (Rx + S + error) = 0

% Written by: Sebastian Valencia, Universidad de los Andes, 2016
    p = 0; q = 0;  % Initial values for quadratic term coefficients
    n = size(polynomial, 2); % Polynomial's degree
    R = Inf; S = Inf; % Terms to reduce, suitable value for each is 0

    % b for remainder polynomial term, c, d required for computation
    b = zeros(1, n - 2); c = zeros(1, n - 2); d = zeros(1, n - 2);

    while(error < abs(R + S))

        % Initial values for each matrix (polynomial)
        b(1) = polynomial(1) - p; b(2) = polynomial(2) - p * b(1) - q;
        c(1) = -1; c(2) = -b(1) + p;
        d(1) = 0; d(2) = -1;

        % Recurrence relation
        for i = 3:(n - 2)
            b(i) = polynomial(i) - p * b(i - 1) - q * b(i - 2);
```

```
            c(i) = -b(i - 1) - p * c(i - 1) - q * c(i - 2);
            d(i) = -b(i - 2) - p * d(i - 1) - q * d(i - 2);
        end;

        % New values for both R and S
        R = polynomial(n - 1) - p * b(n - 2) - q * b(n - 3);
        S = polynomial(n) - q * b(n - 2);

        % Gradient for R and S
        drdp = -b(n - 2) - p * c(n - 2) - q * c(n - 3);
        dsdp = -q * c(n - 2);
        drdq = -b(n - 3) - p*d(n - 2) - q * d(n - 3);
        dsdq = -b(n - 2) - q * d(n - 2);

        % New values for p and q
        J = drdp * dsdq - dsdp * drdq;
        p = p - (1 / J) * (R * dsdq - S * drdq);
        q = q + (1 / J) * (R * dsdp - S * drdp);

    end;
end
```

Script 7: Solución de ecuaciones de grado arbitrario mediante el uso (de ser posible), de manipulación algebraica, en otro caso, de una secuencia de aplicaciones del algoritmo 3 sobre el polinomio y su residuo. Es una implementación de 4.

```
function x = nroots(polynomial, error)
% find numerical approximations for the roots of the given polynomial, with
% certain tolerance defined by the argument error. The degree of the polynomial
% has not restictions at all. It makes uses of quadratic formula and
% Cradano's method for polynomial degrees < 4. Otherwise, it makes use of the Bairstow method as defined in bairstow.m

%   Arguments:
%       polynomial: A real valued polynomial, whose degree is determined by
%       the size of this matrix.
%       error: The tolerance required for iterating changing the paramters
%       p, q, and b

%   USAGE:
%   roots = nroots(polynomial, error)
%   forall r in roots => polyval(polynomial, r) = 0
```

```
% Written by: Sebastian Valencia, Universidad de los Andes, 2016
    sz = size(polynomial, 2);       % degree of polynomial
    x = zeros(1, sz - 1);           % allocating memory for the roots

    % Is the solution trivial?
    % If so, use alegbraic manipulation
    if((sz - 1) == 1)
        x(1) = -polynomial(2)  / polynomial(1);
    elseif((sz - 1) == 2)
        x = quadratic(polynomial);
    elseif((sz - 1) == 3)
        x = cardano(polynomial);
    % Otherwise, use numerical methods (Bairstow's)
    else
        % Setting up the conditions for proper execution of bairstow's
        % method as define here
        polynomial = normalized(polynomial);
        polynomial = polynomial(1, 2:sz);   % The first coefficient is
            not important
        n = size(polynomial, 2); % The size of the new polynomial

        % While the solution is not trivial
        i = 1;
        while(n > 3)
            % Call Bairstow's method
            [p, q, polynomial] = bairstow(polynomial, error);
            % Get the roots of the quadratic term, then, two new roots
                for
            % the original polynomial
            quad = quadratic([1 p q]);
            x(i) = quad(1); x(i + 1) = quad(2);
            % Now b is the new polynomial, get the new size
            n = size(polynomial, 2);
            i = i + 2; % 2 more roots found
        end;

        % When the solution is trivial, get it by algebraic
            manipulation,
        % and add it to the roots of the gicen polynomial.
        naive = basecase([1 polynomial]);
        c = 1;
        for j=i:(sz - 1)
            x(j) = naive(c); c = c + 1;
        end
    end;
end
```

- **Validación.** Como primer medida de validación, se pueden tomar los ejemplos de los textos citados ([2], [1]). A continuación, se muestran dos polinomios con sus raíces respectivas (según los textos), además, se muestra la ejecución del código escrito sobre la representación de los mismos polinomios:

  Ejemplo tomado de la referencia [1]:

$$P_6(x) : 5x^6 - 30x^5 + 56x^4 - 96x^3 + 131x^2 - 18x + 24 = 0$$
$$R = \{5.3091 \times 10^{-18} \pm 0.44721j,\ 4.7003X10^{-17} \pm 1.7321j,\ 4,\ 2\}$$

  La ejecución de estas lineas:

```
pol = [5 -30 56 -96 131 -18 24];
rs = nroots(pol, 0.05e-20);
```

  Da como resultado:

```
rs = [0.0000 + 0.4472i     0.0000 - 0.4472i     -0.0000 + 1.7321i
-0.0000 - 1.7321i     4.0000 + 0.0000i     2.0000 + 0.0000i]
```

  Ejemplo tomado de la referencia [2]:

$$P_7(x) : x^7 - 4\,x^6 + 25\,x^5 + 30\,x^4 - 185\,x^3 + 428\,x^2 - 257\,x - 870 = 0$$
$$R = \{-1,\ -3,\ 1 \pm 2j,\ 2 \pm 5j,\ 2\}$$

  La ejecución de estas lineas:

```
pol = [1 -4 25 30 -185 428 -257 -870];
rs = nroots(pol, 0.05e-20);
```

  Da como resultado:

```
rs = [2.0000 + 0.0000i     -1.0000 + 0.0000i     1.0000 + 2.0000i
1.0000 - 2.0000i     -3.0000 + 0.0000i     2.0000 + 5.0000i     2.0000 -
5.0000i]
```

  Para proceder con la validación del algoritmos, sobre polinomios de grado arbitrario, se ejecuta un script que calcule las raíces haciendo uso del método implementado, y de la función nativa de Matlab, `roots`. A partir de esto, se ordenan los resultados y para cada uno de ellos, se mide que tan lejos esta la evaluación del polinomio de cero, para cada raíz y por cada método. Para cada grado del polinomio, se determina el error total, y este, se toma como medida para evaluar el desempeño del algoritmo. El siguiente script, realiza estos cálculos:

```
% Validate the execution and computationf of the nroots function, whose
% purpose is to find the roots of an arbitrary degree polynomial. This,
% is a comparisson between nroots (bairstow), and native function roots

degrees = [3 5 6 8 13 15 17 20];    % degrees to perfomr evaluation
iter = 8;                           % size of degrees

epsilon = 0.00001;                  % tolerance for Bairtsow's method

matlaberrors = zeros(1, iter);      % errors by root function
bairstowerrors = zeros(1, iter);    % errors by Bairtsow's methos

i = 1;
while i <= iter                     % for each degree, perform the comparisson
    current = degrees(i);           % current degree

    % a random polynomial of current degree
    polynomial = rand(1, current + 1);

    % root by Matlab'w own function and Bairstow's method
    matlabroots = roots(polynomial);
    bairstowroots = nroots(polynomial, epsilon);

    % acum error for each of the roots
    matlaberror = 0; bairstowerror = 0;

    % evaluate each root at the polynomial, and measure how far this
    % value is from zero. Register each for each method
    for j = 1:current
        matlabroot = matlabroots(j);
        bairstowroot = bairstowroots(j);

        matlabval = polyval(polynomial, matlabroot);
        bairstowval = polyval(polynomial, bairstowroot);

        matlaberror = matlaberror + matlabval;
        bairstowerror = bairstowerror + bairstowval;
    end;

    % populate error vectors
    matlaberrors(i) = abs(matlaberror);
    bairstowerrors(i) = abs(bairstowerror);
    i = i + 1;
end;
```

Durante el proceso de validación, se observaron aspectos relevantes que vale la pena resaltar. Para validar el algoritmo con el script inmediatamente anterior, se usaron diferentes tolerancias ϵ, para el calculo de las raíces con la implementación del algoritmo 4. A continuación, se exponen algunos de estos resultados (ver primero figura 3):

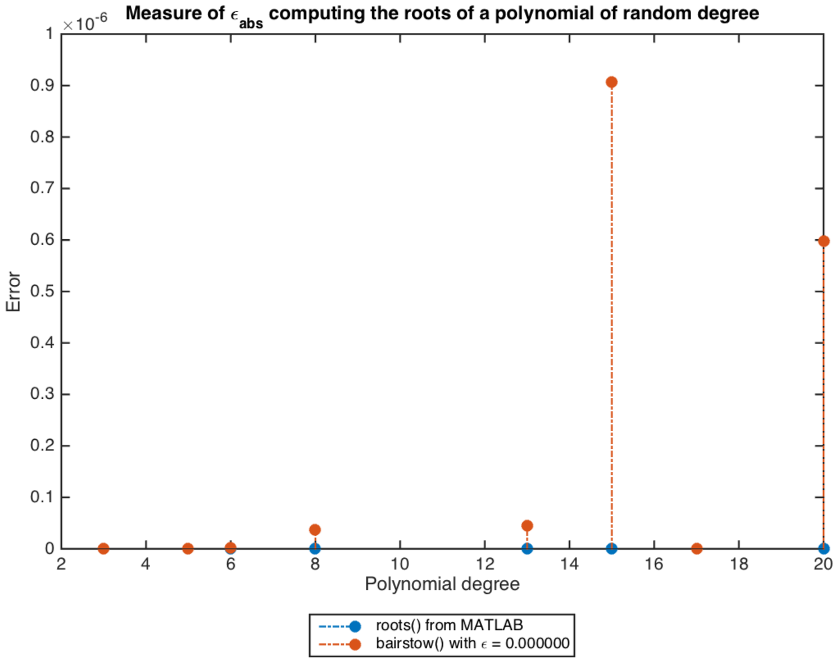

Figura. 2: Cálculo con $\epsilon = 0.00000001$

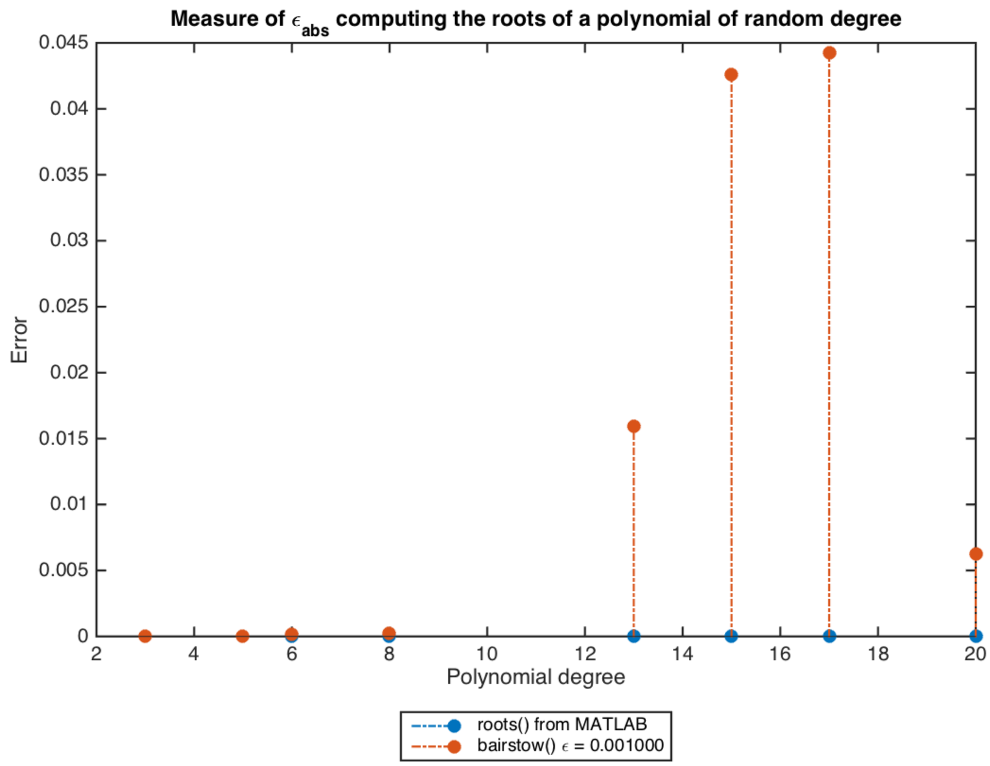

Figura. 3: Medida el error absoluto para el cálculo de las raíces de un polinomio aleatorio dado su grado. Cada medida, corresponde a la suma de las distancias de cada raíz al cero, es decir, el valor real de la evaluación del polinomio en una raíz. Esto, para cada método. Para la ejecución del script `nroots`, se usó una tolerancia de 0.001, este determina completamente el error obtenido, por lo que se afina este con el objetivo de reducir el error absoluto (Ver figura 2).

Como puede verse, las limitaciones de la máquina, del lenguaje y del algoritmo, inducen a la presencia de error en el calculo de las raíces de un polinomio, sin embargo, este error aunque no siempre mejor que el de la función nativa, el error es bajo, y puede considerarse un método estable para valores relativamente pequeños de grados de polinomios .

Bibliografía

[1] Miller, G. *Numerical Analysis for Engineers and Scientists.* 2014. Cambridge University Press. Pags. 175 - 178.

[2] Rosloniec, S. *Fundamental Numerical Methods for Electrical Engineering.* 2008. Lecture Notes in Electrical Engineering - Springer Berlin Heidelberg. Pags. 30 - 35.

[3] Yang, W.Y. and Cao, W. and Chung, T.S. and Morris, J. *Applied Numerical Methods Using MATLAB.* 2005. Wiley. Pags. 111 - 112.

[4] Suli, E. and Mayers, D.F. *An Introduction to Numerical Analysis.* 2003. Cambridge University Press. Pags. 87 - 93.

[5] Cheney, E. and Kincaid, D. *Numerical Mathematics and Computing.* 2007. International student edition - Cengage Learning. Pags. 305 - 306.

[6] Gander, W. and Gander, M.J. and Kwok, F. *Scientific Computing - An Introduction using Maple and MATLAB.* 2014. Texts in Computational Science and Engineering - Springer International Publishing. Pags. 88 - 94.

[7] Watkins, D.S. *Fundamentals of Matrix Computations.* 2004. Pure and Applied Mathematics: A Wiley Series of Texts, Monographs and Tracts - Wiley. Pags. 32 - 48.

[8] Ford, W. *Numerical Linear Algebra with Applications: Using MATLAB.* 2014. Elsevier Science. Pags. 267 - 280.