INSTITUTO POLITÉCNICO NACIONAL
ESCUELA SUPERIOR DE FÍSICA Y MATEMÁTICAS

Servicio Social

Apuntes del tema:

Fórmula de Trench para los determinantes de matrices de Toeplitz generadas por funciones racionales

Jesús Alberto Flores Hinostrosa

Proyecto de investigación: IPN-SIP 20230216

Director del proyecto de investigación:
Egor Maximenko

Ciudad de México
2025

1. Introducción

Denotemos por $\mathbb{T}$ a la circunferencia unitaria en el plano complejo:

$$\mathbb{T} \coloneqq \{z \in \mathbb{C}:\ |z| = 1\}.$$

Supongamos que a es una función racional que no tiene polos en $\mathbb{T}$. Entonces, a se puede descomponer en una serie de Laurent:

$$a(z) = \sum_{k\in\mathbb{Z}} t_k z^k.$$

Esta serie converge en un anillo abierto con centro 0 que contiene a $\mathbb{T}$. Los coeficientes t_k se determinan de manera única. Para cada n en $\mathbb{N}$, asociamos a la función a la *matriz de Toeplitz* $T_n(a)$ mediante la siguiente regla:

$$T_n(a) \coloneqq \left[t_{k-j}\right]_{j,k=1}^{n}. \tag{1}$$

Por ejemplo, si $n = 5$, entonces

$$T_5(a) = \begin{bmatrix} t_0 & t_1 & t_2 & t_3 & t_4 \\ t_{-1} & t_0 & t_1 & t_2 & t_3 \\ t_{-2} & t_{-1} & t_0 & t_1 & t_2 \\ t_{-3} & t_{-2} & t_{-1} & t_0 & t_1 \\ t_{-4} & t_{-3} & t_{-2} & t_{-1} & t_0 \end{bmatrix}.$$

Se dice que a es la *función generadora* o el *símbolo generador* de las matrices $T_n(a)$.

Más general, se puede suponer que a es una función integrable en $\mathbb{T}$, y para cada k en $\mathbb{T}$ definir t_k como el k-ésimo coeficiente de Fourier de a:

$$t_k \coloneqq \frac{1}{2\pi}\int_0^{2\pi} a(e^{ix})e^{-kix}\,dx.$$

Las matrices de Toeplitz se han estudiado desde los principios del signo XX. Los libros [2, 1] contienen la información histórica y explican muchas propiedades de estas matrices.

El objetivo de este trabajo es entender y explicar el resultado de Trench [4] sobre los determinantes y los polinomios característicos de las matrices de Toeplitz $T_n(a)$ generadas por funciones racionales. Se programaron distintas formulas del artículo y comprobaciones numéricas para esto.

2. Notación del artículo

En el desarrollo del artículo de Trench se considera que el simbolo generador de las matrices de Toeplitz se representa como el siguiente cociente:

$$R(z) = \frac{C(z)}{A(z)B(1/z)}, \tag{2}$$

donde A y B son dos polinomios y C es un polinomio de Laurent

$$A(z) = \sum_{\mu=0}^{r} a_\mu z^\mu \quad (r \in \mathbb{N}), \tag{3}$$

$$B(z) = \sum_{\nu=0}^{s} b_\nu z^\nu \quad (s \in \mathbb{N}), \tag{4}$$

$$C(z) = \sum_{j=-q}^{p} c_j z^j \quad (p, q \in \mathbb{Z}). \tag{5}$$

Se está asumiendo que los polinomios $A(z), z^s B(1/z)$ y $z^q C(z)$ no tienen factores en común, además,

$$a_0, a_r, b_0, b_s, c_p, c_{-q} \neq 0 \quad \text{y} \quad p + q > 0,$$

aunque no se está asumiendo que p y q tienen el mismo signo. Para escribir mejor los resultados se define

$$M = \max(p, r), \quad N = \max(q, s), \quad k = M + N.$$

La convención utilizada en el artículo de Trench es que el polinomio característico esta dado por

$$p_n(\lambda) = \det[\lambda I_n - T_n]. \tag{6}$$

De acuerdo a los resultados de Trench, se requiere definir el polinomio

$$P(z;\lambda) = z^N (C(z) - \lambda A(z)B(1/z)), \tag{7}$$

el cual es de grado k en z para todos los valores (salvo uno) de λ. Las raices de este polinomio son cruciales para la formula del polinomio característico. Escribimos

$$A(z)B(1/z) = \sum_{j=-s}^{r} \theta_j z^j, \tag{8}$$

con esto podemos reescribir la ecuación (7) como

$$P(z;\lambda) = \sum_{j=-N}^{M} (c_j - \lambda\theta_j) z^{j+N}. \tag{9}$$

De la última expresión se pueden obtener las siguientes observaciones:

i) $P(z;\lambda)$ es un polinomio en la variable z de grado k, a menos que

$$c_M - \lambda\theta_M = 0, \tag{10}$$

lo que ocurre para un valor de λ si y sólo si $M = r$.

ii) Para todos los valores de λ que no satisfacen (10), $P(z;\lambda)$ tiene k raíces contando multiplicidad y todas son distintas de cero a menos que

$$c_{-N} - \lambda\theta_{-N} = 0, \tag{11}$$

lo que ocurre para un valor de λ sólo sí $N = s$.

Es por estas razones que se asumira que en general para cada λ se tienen k raices del polinomio $P(z;\lambda)$.
En lo siguiente presentamos la construcción de una matriz que tiene un papel importante en la formula de Trench, primero se consideran polinomios arbitrarios.

Matriz Ω

Para λ fijo que no satisface (10), se consideran que $z_1, \ldots, z_L$ son las distintas raíces de $P(z;\lambda)$, con respectivas multiplicidades $m_1, \ldots, m_L$; por lo que

$$L \leq k, \quad m_j \geq 1 \quad (1 \leq j \leq L), \quad m_1 + \cdots + m_L = k.$$

Consideramos una familia de k polinomios $Q_1(z), \ldots, Q_k(z)$ y definimos el vector columna

$$w(z) = [Q_1(z)\, Q_2(z) \cdots Q_k(z)]^t.$$

La matriz $\Omega \in M_k(\mathbb{C})$ se contruye como sigue:

i) En las primeras m_1 columnas participan las derivadas de $w(z)$ evaluadas en z_1. Para $l \in [0, m_1 - 1]$, la l-ésima columna de Ω es $w^{(l)}(z_1)$.

ii) Las siguientes m_2 columnas forman una matriz donde la l-ésima columna es $w^{(l)}(z_2)$ con $l \in [0, m_2 - 1]$.

iii) En las siguientes columnas se hace algo similar separando en bloques de tamaño de las multiplicidades.

Para el caso en el que se tienen k raíces distintas y los polinomios

$$Q_j(z) = z^{j-1}, \quad 1 \leq j \leq k,$$

con esta construcción resulta la matriz de Vandermonde:

$$V = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ z_1 & z_2 & \cdots & z_k \\ z_1^2 & z_2^2 & \cdots & z_k^2 \\ \vdots & \vdots & & \vdots \\ z_1^{k-1} & z_2^{k-1} & \cdots & z_k^{k-1} \end{bmatrix}. \tag{12}$$

El caso especial que nos interesa es para los polinomios

$$Q_j(z) = \begin{cases} z^{j-1}A(z), & 1 \le j \le N, \\ z^{n+j-1}B(1/z) & N+1 \le j \le k, \end{cases}$$

denotaremos la matriz Ω por Ω_n. En particular, para el caso de raíces distintas a pares resulta

$$\Omega_n = \begin{bmatrix} A(z_1) & A(z_2) & \cdots & A(z_k) \\ \vdots & \vdots & & \vdots \\ z_1^{N-1}A(z_1) & z_2^{N-1}A(z_2) & \cdots & z_k^{N-1}A(z_k) \\ z_1^{n+N}B(1/z_1) & z_2^{n+N}B(1/z_2) & \cdots & z_k^{n+N}B(1/z_k) \\ \vdots & \vdots & & \vdots \\ z_1^{n+k-1}B(1/z_1) & z_2^{n+k-1}B(1/z_2) & \cdots & z_k^{n+k-1}B(1/z_k) \end{bmatrix}. \tag{13}$$

Se define

$$\Delta_n(\lambda) = \frac{\det \Omega_n}{\det V}. \tag{14}$$

Hay que resaltar que $\Delta_n(\lambda)$ no es una función de λ explícitamente pero si depende indirectamente ya que involucra las raíces de $P(z;\lambda)$. Además, hay que recordar que n es el tamaño de la matriz de Toeplitz que estamos considerando.

Matriz R

Denotaremos por R a la matriz de tamaño $k \times k$ con filas $j = 1, \cdots, k$ construidas de la siguiente manera:

a) Para $1 \le j \le N$, las primeras $j-1$ entradas son ceros, después se agregan los coeficientes $a_0, \cdots, a_M$ y el las últimas $N-j$ entradas hay ceros.

b) Para $N+1 \le j \le k$, las primeras $j-N-1$ entradas son ceros, después se agregan los coeficientes $b_N, \cdots, b_0$ y el las últimas $k-j$ entradas hay ceros.

Por ejemplo, tomando $M = r = 3$ y $N = s = 2$ la matriz R es

$$R = \begin{bmatrix} a_0 & a_1 & a_2 & a_3 & 0 \\ 0 & a_0 & a_1 & a_2 & a_3 \\ b_2 & b_1 & b_0 & 0 & 0 \\ 0 & b_2 & b_1 & b_0 & 0 \\ 0 & 0 & b_2 & b_1 & b_0 \end{bmatrix}.$$

3. Teoremas utilizados

En esta sección se enlistan las expresiones en las que se basa el programa para realizar los calculos. Evidentemente una de las expresiones que participa es la formula de Trench, considerando las definiciones del teorema anterior se tiene el siguiente teorema:

3.1 Teorema (**Formula de Trench para el polinomio característico de matrices de Toeplitz generadas por simbolos racionales**). *Si*

$$n > \max(r + s - k, 0),$$

el polinomio característico (6) *está dado por*

$$p_n(\lambda) = (-1)^{(M-1)n} \det(R)^{-1}(a_0 b_0)^{-n}(c_M - \lambda\theta_M)^n \Delta_n(\lambda).$$

Podemos ver que la formula de Trench da una expresión explícita de la forma que tiene el polinomio característico con λ como variable. Para realizar los cálculos se consideró que λ es un parámetro que se conoce.

Para verificar la formula se consideran dos situaciones en las que se pueden realizar calculos y después se comparan los cálculos, que deben de dar el mismo resultado. Una de esas formas consiste en utilizar directamente la formula, mientras que la otra consiste en utilizar expresiones que permiten encontrar la matriz de Toeplitz asociada a los datos iniciales y calcular directamente el valor del polinomio característico (6).

De acuerdo al artículo de Trench, se puede conocer los coeficientes de la matriz de Toeplitz $\{t_j\}_{j\in\mathbb{Z}}$ por medio de la siguiente expresión

$$t_j = \sum_{l=-q}^{p} c_l \phi_{j-l}, \tag{15}$$

donde

$$A(z)B(1/z)\left(\sum_{j=-\infty}^{\infty} \phi_j z^j\right) = 1 \quad \text{para ciertos valores de } z.$$

Estos coeficientes $\{\phi_j\}_{j\in\mathbb{Z}}$ pueden calcularse con técnicas que Greville y Trench desarrollaron en un artículo anterior [3].

3.2 Teorema (Greville-Trench). *Sean* $A(z)$ *y* $B(z)$ *como en* (3) *y* (4), *con* $a_0 b_0 \neq 0$ *y* $A(z)$ *primo relativo con* $z^s B(1/z)$. *Entonces existe una única sucesión* $\{\phi_j\}_{j\in\mathbb{Z}}$ *tal que*

$$\sum_{\nu=0}^{r} a_\nu \phi_{j-\nu} = b_0^{-1}\delta_{j,0}, \quad j \geq 0,$$

y

$$\sum_{\mu=0}^{s} b_\mu \phi_{-j+\mu} = a_0^{-1}\delta_{j,0}.$$

Más aún, si $m > r + s$*, entonces la matriz de Toeplitz*

$$\Phi_m = [\phi_{j-l}]_{l,j=0}^{m-1} \tag{16}$$

es invertible, con inversa

$$\Phi_m^{-1} = [h_{l,j}]_{l,j=0}^{m-1}, \tag{17}$$

donde

$$h_{l,j} = \theta_{j-l} - \sum_{\nu=l+1}^{s} a_{j-l+\nu} b_\nu - \sum_{\mu=m-l}^{r} b_{l-j+\mu} a_\mu, \quad 0 \le l, j \le m-1. \tag{18}$$

4. Implementación del programa

Vamos a escribir un programa en Sagemath. Las componentes de vectores y matrices serán numeros racionales (del tipo QQ), esto permite evitar errores numéricos relacionados con rendondeos. Para simplificar los calculos se está asumiendo que $M = r$, $N = s$ y que el polinomio $P(z;\lambda)$ tiene k raices simples $z_1, \ldots, z_k$. Un polinomio

$$P(z) = p_0 + p_1 z + \cdots + p_d z^d,$$

lo guardamos en el programa como un vector

$$\left(p_0,\, p_1,\, \cdots,\, p_d\right).$$

Además de polinomios usuales, el artículo de Trench considera polinomios de Laurent, que tienen la forma

$$Q(z) = \sum_{j=-s}^{r} q_j z^j = z^{-s}(q_{-s} + q_{-s+1} z + \cdots + q_r z^{r+s}), \quad (r, s \in \mathbb{N}). \tag{19}$$

Pensado de esta forma podemos considerar en polinomios de Laurent como vectores siempre que se trabaje adecuadamente de los coeficientes, de modo que la información guardada de la ecuacion (19) sería

$$\left[s,\quad \left(q_{-s},\, q_{-s+1},\, \cdots,\, q_r\right)\right]. \tag{20}$$

Debido a que estamos ocupando vectores para guardar la información de polinomios, es necesario programar operación relacionadas con polinomios. Las siguientes son algunas de las funciones sencillas que se programaron

- **derivatePoly(a)** regresa el arreglo que corresponda a la derivada el polinomio,
- **prodMonAndPoly(a,b)** multiplica el monomio $(x + a)$ por el polinomio b,
- **polyFromRoots(raices)** construye un polinomio a partir de sus raices,
- **prodPoly(a,b)** multiplica el polinomio a con el polinomio b
- **evaluatePoly(c,p)** dado un polinomio p y un valor c, regresa el valor p(c).
- **polyCompUnoEnZ(B,j)** determina el polinomio que resultada de multiplicar z^j por $B(1/z)$. Para asegurar que el resultado es un polinomio se está asumiendo que $j > -grad(B(z))$.

Anexamos los codigos de las funciones mencionadas anteriormente.

```
def derivatePoly(a):
    n = len(a) - 1
    b = vector(QQ, n)

    for j in range(n):
        b[j] = (j+1) * a[j + 1]
    return b

def prodMonAndPoly(a, b):
    n = len(b)
    c = vector(QQ, n + 1)

    c[0] = a * b[0]
    c[n] = b[n - 1]
    for j in range(1, n):
        c[j] = b[j - 1] + a * b[j]
    return c

def polyFromRoots(raices):
    n = len(raices)
    c = vector(QQ, [1] )

    for i in range(n):
        c = prodMonAndPoly(-raices[i], c)

    return c

def prodPoly(a, b):
    n = len(a) - 1
    m = len(b) - 1

    p = vector(QQ, n + m + 1)
    q = vector(QQ, n+ m + 1)
    c = vector(QQ, n + m + 1)

    p[0 : n + 1] = a
    q[0 : m + 1] = b

    for k in range(m + n + 1):
        for j in range(k + 1):
            c[k] += p[j] * q[k - j]
    return c
```

```
def evaluatePoly(c, p):
    n = len(p)
    r = 0
    for j in range(n):
        r = p[j] * (c ** j) + r
    return r

def polyCompUnoEnZ(B, j):
    s = len(B) - 1
    C = vector(QQ, j + 1)

    for k in range(s + 1):
        C[j - s + k] = B[s - k]
    return C
```

Para comprobar la formula, el programa genera aletoriamente dos polinomios A y B, se considera un valor para λ, un vector cuyos elementos son las raices de $P(z;\lambda)$ y se elige el tamano de la matriz de Toeplitz n.

4.1 Algoritmo (Calculo del polinomio de Laurent $C(z)$)**.** Conociendo A, B, λ y las raíces del polinomio $P(z;\lambda)$ es posible encontrar $C(z)$ despejandolo de la expresión (9). Para encargarse de eso se utiliza la siguiente función:

```
    def calPolyC(A, B, lamb, raices):
        r = len(A) - 1
        s = len(B) - 1
        k = len(raices)
        if(k != (r + s)):
            print("k distinto de r + s")
            return 0

        p = polyFromRoots(raices)
        auxtheta = calcTheta(A, B)
        theta0 = auxtheta[1]

        #Calculamos los coef de c
        c = vector(QQ, k + 1)
        theta = vector(QQ, k + 1)
        theta[0 : len(theta0)] = theta0

        for j in range(k + 1):
            c[j] = p[j] + (lamb * (theta[j]))
```

```
        return (s, c)
```

De acuerdo con el convenio mencionado anteriormente, la función devuelve la tupla $(\mathfrak{s}, \mathfrak{c})$.

Determinante de la matriz de Toeplitz asociada.

4.2 Algoritmo (Calculo de los coeficientes ϕ_j)**.** Este algoritmo sigue al teorema 3.2. Se calcula la matriz $\Phi_{\mathfrak{m}}^{-1}$ y su inversa $\Phi_{\mathfrak{m}}$ por medio de los coeficientes (18). Los datos iniciales son $\mathfrak{m}$ (el tamaño de la matriz que queremos construir) y los polinomios A y B dados como lista de coeficientes.

```
def calc_coef_phi(m, A, B):
    #Se necesita que A(z) y z^s B(1/z) sean primos relativos

    r = len(A) - 1
    s = len(B) - 1

    # De acuerdo al teorema, necesitamos que n > r + s
    if m < r + s:
        print("se␣debe␣cumplir␣n␣>␣r␣+␣s")
        return 0

    #Recordar que theta_j = 0 si j < -s o j > r
    #auxtheta[0] = theta_{-s}
    auxtheta = prodPoly(A, polyCompUnoEnZ(B, s))

    H = matrix(QQ, n, n)
    for i in range(n):
        for j in range(n):
            c=0
            v=0

            if i + 1 > s:
                c=0
            else:
                for k in range(i + 1, s + 1):
                    if j - i + k > r or  j - i + k < 0:
                        c += 0
                    else:
                        c += A[j - i + k] * B[k]
```

```
            if( n-i > r):
                v=0
            else:
                for m in range(n - i, r + 1):
                    if( i - j + m > s or i - j + m < 0 or m < 0 or m > r ):
                        v += 0
                    else:
                        v += B[i - j + m] * A[m]

            if( j - i + s > r + s or j - i + s < 0):
                H[i,j] = - c - v
            else:
                H[i,j] = auxtheta[j - i + s] - c - v

    Q = H.inverse()
    phip = vector(QQ, n)
    phin = vector(QQ, n)

    for i in range(n):
        phin[i] = Q[i, 0]
    for i in range(n):
        phip[i] = Q[0, i]

    return (phip, phin)
```

Notamos que el resultado de este programa son los coeficientes $(\phi_j)_{j\in\mathbb{Z}}$ separados en dos arreglos, uno con los términos de indices mayores o iguales que cero y otro con términos de indices negativos.

Estos calculos se realizaron pensando en ocupar la formula (15), que corresponden a la matriz de Toeplitz (1).

4.3 Algoritmo. Se ocupan los coeficientes de $C(z)$ y $(\phi_j)_{j\in\mathbb{Z}}$, que se calcularon anteriormente, y el tamaño de la matriz $T_n(a)$.

```
def calCoefsT(C, s, phip, phin, n):
    if(len(phip) != len(phin)):
        return 0

    r = len(C) - (s + 1)
    tpos = vector(QQ, n)
    tneg = vector(QQ, n)
```

```
    #c_l = C[l+s] con -s <= l
    #j math = -j en el programa
    for j in range(n):
        for l in range(-s, r + 1):
            m = - j - l
            phi_coef = phip[m] if m >= 0 else phin[-m]
            tneg[j] += C[l + s] * phi_coef

    for j in range(n):
        for l in range(-s, r + 1):
            m = j - l
            phi_coef = phip[m] if m >= 0 else phin[-m]
            tpos[j] += C[l+s] * phi_coef

    return (tpos, tneg)
```

El resultado son los coeficientes separados en dos arreglos, uno con los términos de indices mayores o iguales que cero y otro con términos de indices negativos.

Las funciones anteriores ayudan a calcular los coeficientes que necesitamos para construir la matriz (1), por lo que se necesitan funciones que construyan la matriz y calculen su polinomio característico.

4.4 Algoritmo (Construcción de matrices de Toepliz a partir de dos arreglos)**.** Dados dos arreglos, siendo A el que tienen los coeficientes con indices mayores o iguales que cero y B el que tiene los coeficientes con indices negativos, se contruye una matriz de Toeplitz.

```
def toeplitzMatrizFromArrays(A, B):
    n = len(A)
    m = len(B)
    if n != m:
        return 0

    M = matrix(QQ, n, n)
    for i in range(n):
        for j in range(n):
            if j > i:
                M[i, j]=A[j - i]
            else:
                M[i, j]=B[i - j]

    return M
```

La siguiente función junta las funciones que calculan los valores necesarios para construir la matriz de Toeplitz $T_n(a)$.

4.5 Algoritmo (Construcción de la matriz $T_n(a)$)**.** Se utiliza el tamaño de la matriz $T_n(a)$, los polinomios A, B, las raices del polinomio $P(z;\lambda)$ y el valor λ.

```
def toeplitzFromData(n, A, B, raices, lamb):
    polLaurentC = calPolyC(A, B, lamb, raices)
    C = polLaurentC[1]
    s = polLaurentC[0]
    r = len(A)
    k = r + s

    phip, phin = calc_coef_phi(n + k, A, B)
    tp, tn = calCoefsT(C, s, phip, phin, n)
    M = toeplitzMatrizFromArrays(tp, tn)

    return M
```

Por último, se calcula su polinomio característico evaluado en λ. Que es un de los datos que compararemos al final.

4.6 Algoritmo (Calculo del polinomio característico evaluado en λ)**.** Para esta función se utiliza el tamaño de la matriz $T_n(a)$, los polinomios A, B, las raices del polinomio $P(z;\lambda)$ y el valor λ.

```
def caractPolFromData(n, A, B, raices,lamb):
    T = toeplitzFromData(n, A, B, raices, lamb)
    T1 = lamb * identity_matrix(n) - T

    return T1.det()
```

Formula de Trench

Para ocupar la formula de Trench (3.1) se programaron funciones que calculan por separado los terminos involucrados.

4.7 Algoritmo (Calculo del determinante de la matriz R)**.** Se utilzan los polinomios A, B y los valores M, N para calcular el derminante de la matriz R.

```
def calcR(A, B, M, N):
```

```
    if( M < len(A)-1):
        print("El grado de A es menor que M")
        return 0

    if( N < len(B)-1):
        print("El grado de B es menor que N")
        return 0

    k = M + N
    p = vector(QQ, k) #A extendido
    q = vector(QQ, k) #B extendido

    #Extendemos los polinomios hasta polinomios de grado k-1

    p[0 : len(A)] = A
    q[0 : len(B)] = B

    mat = matrix(QQ, k, k)

    for i in range(N):
        for j in range(M + 1):
            mat[i, j + i] = p[j]

    for i in range(N, k):
        for j in range(N + 1):
            mat[i, j + i - N] = q[len(B) - 1 - j]

    return det(mat)
```

Además de la matriz $\mathtt{R}$, es necesario construir la matriz Ω_n, definida anteriormente (13), y calcular el cociente $\Delta_n(\lambda)$.

4.8 Algoritmo (Construcción de la matriz Ω_n)**.** Esta función utiliza el tamaño de la matriz $\mathtt{T}_n(\mathfrak{a})$, los polinomios A y B, el valor $\mathtt{N}$ y las raices de $\mathtt{P}(z;\lambda)$.

```
def constOmega(n, A, B, N, raices):
    #Suponemos que las raices son simples
    k = len(raices)
    if(N > k):
        print("No se puede construir")
        return 0
```

```
    if(n + N < len(B) + 1):
        print("No se puede construir")
        return 0

    mat = matrix(QQ, k, k)

    for j in range(k):
        mono = [1]
        for i in range(N):
            mat[i, j] = evaluatePoly(raices[j], prodPoly(mono, A))
            mono = prodMonAndPoly(0, mono)

    for j in range(k):
        for i in range(N, k):
            mat[i, j] = evaluatePoly(raices[j], polyCompUnoEnZ(B, n + i))

    return mat
```

Aunque podría ocuparse la función anterior para construir la matriz de Vandermonde, requiere menos memoria programar una función que lo haga de manera individual.

4.9 Algoritmo (Construcción de la matriz de Vandermonde)**.** Esta función construye una matriz de vandermonde de tamaño `k` a partir de un arreglo con `k` elementos.

```
def vandermonde(raices):
k = len(raices)
mat = matrix(QQ, k, k)

#las raices deben ser simples por lo que hay k raices distintas
#"raices" es el arreglo que tiene los valores de las raices

#mat = [[0 for i in range(k)]] * k
# Al crear la matriz salia mal
poly = [1]
#Asignamos los polinomios evaluados en las filas
for j in range(k):
    asignRow(mat, j, raices, poly)
    poly = prodPolyAndMon(0, poly)

return mat
```

Con las matrices Ω_n y V se calcula de manera muy fácil el cociente 14.

4.10 Algoritmo (Calculo del coeficiente Δ_n)**.** Como datos iniciales se toma el tamaño de la matriz $T_n(a)$, los polinomios A y B, el valor N y las raíces de $P(z;\lambda)$.

```
def deltaene(n, A, B, N, raices):
    return det(constOmega(n, A, B, N, raices)) / det(vandermonde(raices))
```

4.11 Algoritmo (Calculo de la formula de Trench para un valor λ)**.** Recordamos que estamos considerando como datos el tamaño de la matriz $T_n(a)$, los polinomios A y B, las raíces de $P(z;\lambda)$ y el valor λ. Esto permite calcular C, la matriz R, el cociente Δ_n, que se ocupan para calcular la formula de Trench (3.1).

```
def trenchPoly(n, A, B, raices, lamb):
    #A,B son arreglos
    #C es de la forma [j , [c0,c1,..,cn] ] = z^{-j} (c0 + c1 z+...+ cn z^n )
    r = len(A) - 1
    s = len(B) - 1
    # p= grado maximo de C
    # q= grado minimo de C

    M = r
    N = s
    k = M + N

    auxC = calPolyC(A, B, lamb, raices)
    C = auxC[1]

    auxtheta = prodPoly(A, polyCompUnoEnZ(B, s))

    factor0 = (-1) ** ((M - 1) * n)
    factor1 = 1 / calcR(A, B, M, N)
    factor2 = 1 / (A[0] * B[0]) ** n
    factor3 = deltaene(n, A, B, N, raices)
    factor4 = C[r + s] - lamb * auxtheta[r + s]

    return factor0 * factor1 * factor2 * factor3 * factor4
```

Esta formula genera el segundo valor que tuvimos como objetivo calcular. Ambos valores se compararon utilizando funciones que generan valores aleatorios.

4.12 Algoritmo (Pruebas con valores aleatorios)**.** Para comparar los calculos se generó una función que recibe p, q y n como parametros, a partir de eso genera los datos requeridos para comparar los dos distintos calculos, devuelve los dos resultados y un operador que indica si ambos valores son iguales. La función es:

```
def randomTestTrench(p,q,n):
    A=random_vector(QQ, p+1, 32)
    B=random_vector(QQ, q+1, 32)
    lamb=QQ.random_element(32)

    raices=random_vector(QQ, p+q, 32)
    #Asegurar que los elementos son diferentes entre si
    #usar list(set(a))

    result1 = trenchPoly(n,A,B,raices,lamb)
    result0 = caractPolFromData(n,A,B,raices,lamb)

    comp = result1 == result0

    return (result1,result0, comp)
```

En todos los casos se obtiene el mismo resultado para los dos distintos calculos. Realizando una medición del tiempo con lo siguiente:

```
a = time.time()
randomTestTrench(4,5,13)
b = time.time()
c = b - a
print('tiempo␣=␣'  + str(c))
```

La función devolvio como resultado:

```
tiempo = 0.05089926719665527.
```

Referencias

[1] Böttcher, A.; Grudsky, S. (2005): Spectral Properties of Banded Toeplitz matrices. SIAM,
doi:10.1137/1.9780898717853.

[2] Böttcher, A.; Silbermann, B. (1999): Introduction to Large Truncated Toeplitz Matrices. Springer,
doi:10.1007/978-1-4612-1426-7.

[3] Trench, W.F., Greville, T.N.E. (1979): Band matrices with Toeplitz inverses. Lin. Alg. Appl, 27: 199–209.
doi:10.1016/0024-3795(79)90042-9

[4] Trench, W.F. (1985): On the eigenvalue problem for Toeplitz matrices generated by rational functions. Linear Multilinear Algebra, 17:3–4, 337–353,
doi:10.1080/03081088508817665.