1. Interpolación Polinómica

1.1. Método de Lagrange

En este apartado se pide implementar la función `lagrange_eval`, que evalúa el polinomio interpolador de Lagrange de grado **n** en un punto **z**, usando los nodos `x[0..n]` y los valores `y[0..n]`.

Para implementarla, sigo directamente la fórmula del polinomio interpolador:

Π_n(z) = Σ (y_i * L_i(z)), i = 0..n

L_i(z) = Π_{j=0..n, j ≠ i} (z - x_j) / (x_i - x_j)

Para esto, debo seguir la fórmula como indica el enunciado,primero defino el bucle que va de i=0…n como indica la primera fórmula. En cada iteración reinicio el li(z) para calcular bien el valor de cada iteración.
Ahora en este punto me di cuenta que para calcular Lagrange necesito hacer un segundo bucle que calcule el Li(z) para cada iteración.

Creó el bucle j = 0…n y pongo la condición que si coincide i y j salte la iteración porque sino daría 0 el cálculo del denominador y sería un error crítico.
calculo por separado y guardo el valor del denominador y numerador del calculo para poder aplicar la condición de si no cumple la tolerancia, cálculo el resultado de li(z) y una vez terminado ya puedo calcular el resultado de lagrange con la iteración del bucle inicial siguiendo la fórmula Π_n(z) += y[i] * L_i(z).

Y finalmente guardar el resultado obtenido por la fórmula después de todas las iteración en el puntero pz con *pz debido porque es puntero y el contenido que refiera sea el esperado y no de conflicto poniendo simplemente pz=lagrange; porque eso solo cambia la referencia de dónde apunta el puntero pero no su contenido, para finalmente devolver 0 de que el cálculo ha sido exitoso.

Ahora para generar la segunda parte de los nodos y cálculos para generar el archivo para comparar resultados:
Empezamos de forma clásica main() y declara las variables que usare para leer los datos necesarios.
Solicitamos y rellenamos las variables, reservamos memoria que usare para el polinomio y valores de vectores, es decir x e y.
Declaro lo que podría ser el primer error, si por alguna razón falla el reserva de memoria de x o y el sistema imprimirá el error declarando específicamente qué es lo que pasó y devolverá el valor -1.

Ahora declaramos los nodos ya sea modo equidistantes o chebyshev con las fórmulas dadas

llado main_lagrange.c. Esta función tiene que leer el
os extremos del intervalo $[a, b]$, permita de elegir entre no

$$x_j = a + j\frac{b-a}{n}$$

$$x_j = \frac{a+b}{2} + \frac{b-a}{2}\cos\left(\frac{2j+1}{2(n+1)}\pi\right)$$

que evalúe el polinomio interpolador en 1000 puntos [a,b].

Para poder implementar las fórmulas se necesita incluir en el proyecto la librería math.h par poder tener las funciones de cos() y el valor de PI declarada en M_PI.

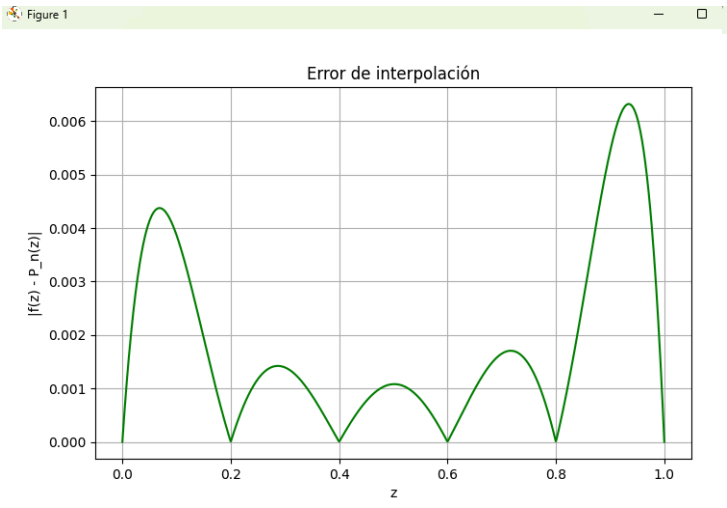
Una vez tenido los nodos calculados, tenemos que conseguir las valores, guardados en el vector y, para ello, un bucle que recorra el vector x que es introducido en la función fun(). Para el método fun() solo declara la cabecera arriba del main(), para que el programa sepia existe la función. El fun() será utilizado dependiendo que .c de función compile con el archivo del main_lagrange.c .

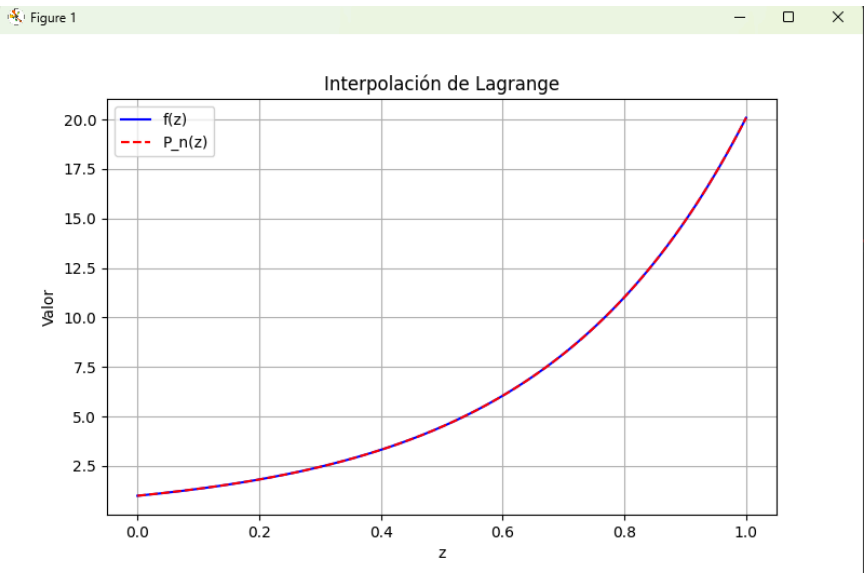
Una vez hecho eso, podemos empezar a evaluar en 1000 puntos, Declaramos el bucle que irá de 0 a 1000 declaramos como el punto donde queremos evaluar y donde guardaremos los resultados en pz, he seguido los mismo nombres que en lagrange_eval para facilidad de entendimiento.
Generamos el archivo donde se guardará e imprimimos los resultados en él. Y finalmente liberamos la memoria.

Es hora de comprobar los resultados obtenidos:

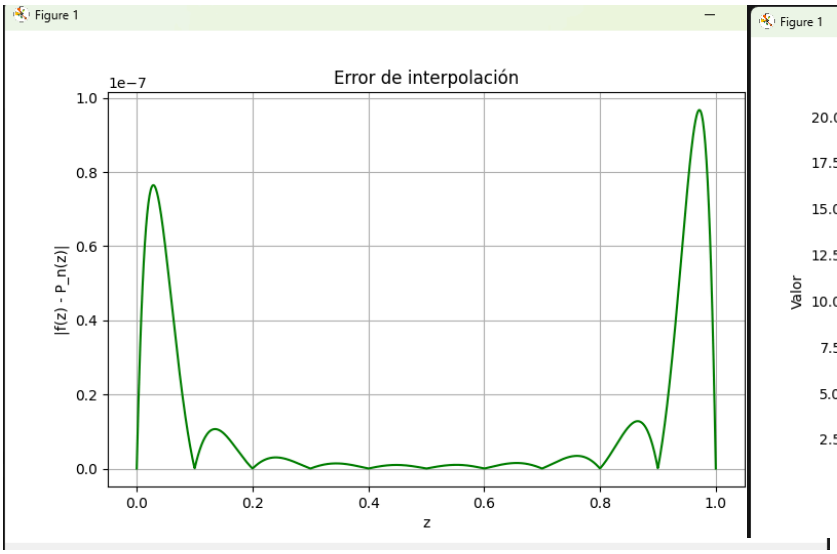
FUNCION EXP(3X)

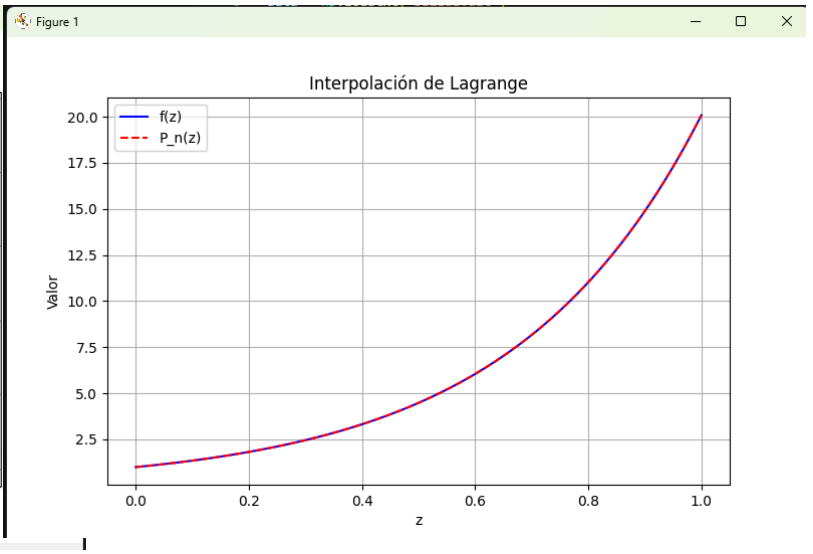
para n=5



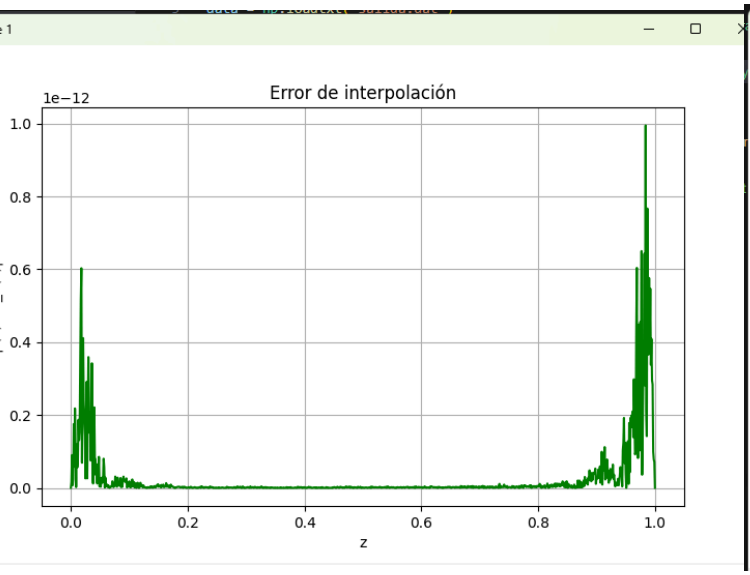


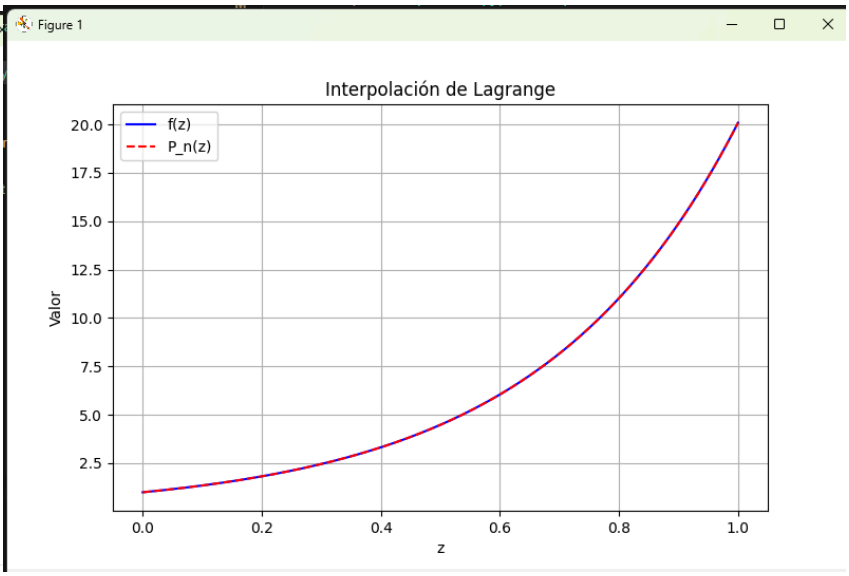
para n=10:





para n=15:

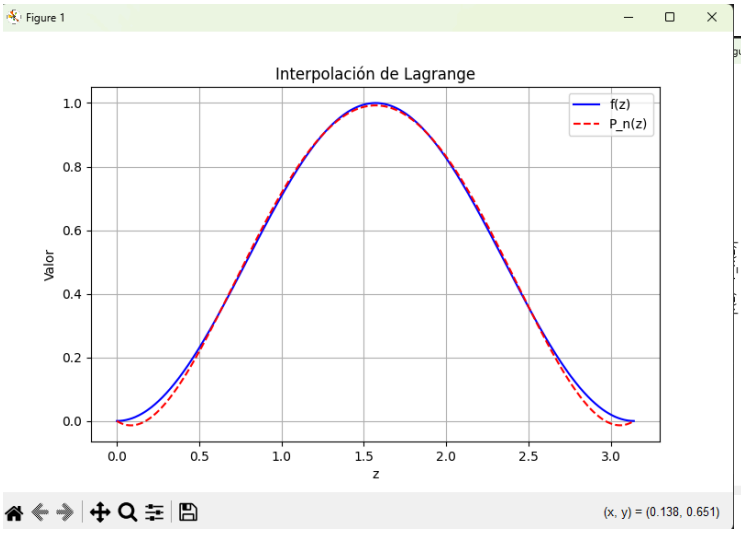


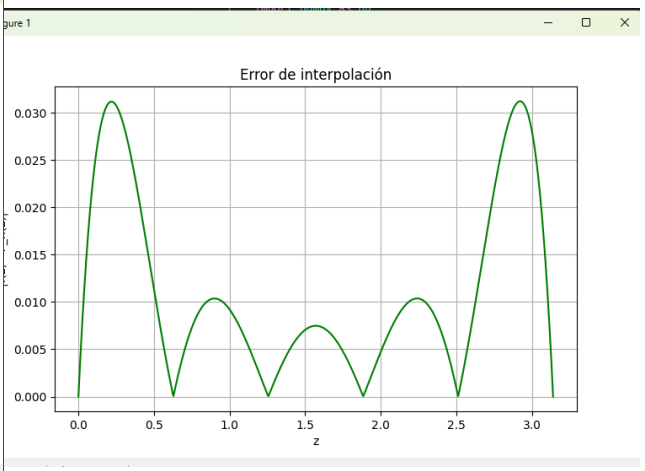


Comentarios de gráficas:
Mientras aumentando el n de interpolación va creciendo la gráfica de interpolación ha simple vista parece ser iguales y es una grafica bastante bien ver que el método funciona dentro del rango. Pero la cosa cambia viendo el resultado de las gráficas de Error vemos que la gráfica si varía notoriamente y con mayor sea la “n” el error coje magnitud menor empezando de 1e-3 para n=5 hasta 1e-12 en n=15. Para las gráficas he utilizado los nodos del tipò equidistantes de la primera fórmula debido que ambos métodos me daban resultados bastante similares y no vi necesario la cimentación de Chebyshev
Conclusión: para funciones suaves y bien comportadas, Lagrange funciona correctamente, y el aumento del grado mejora la aproximación de forma predecible.

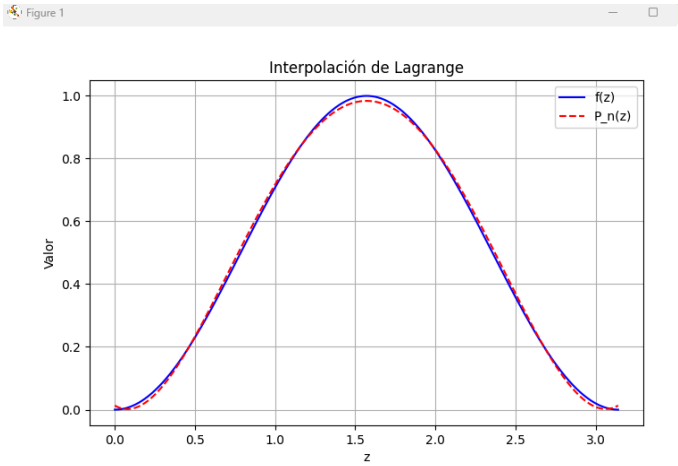
FUN:SIN:

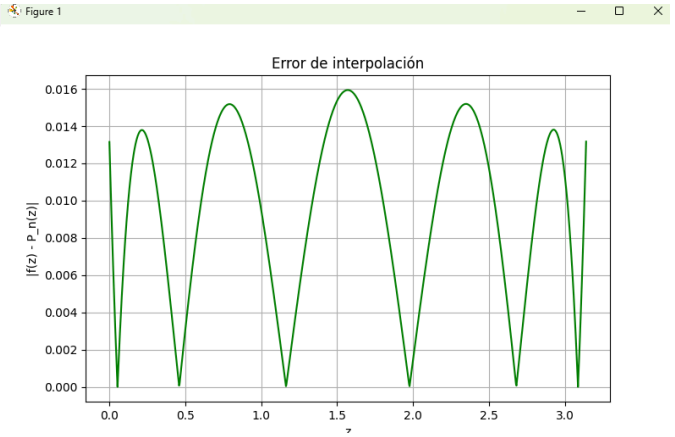
n=5
1 primera formula



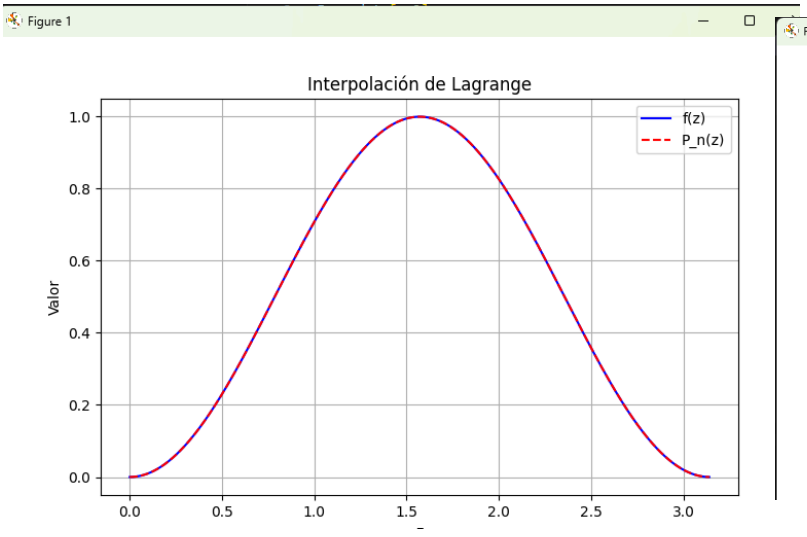


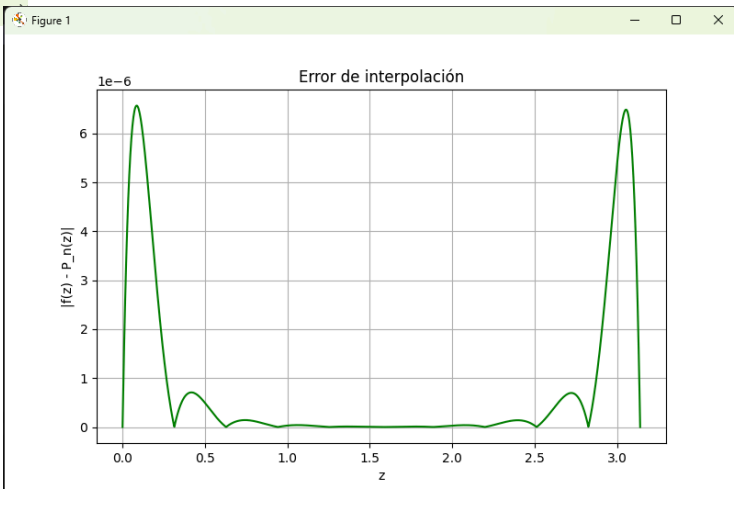
2.Chebyshev

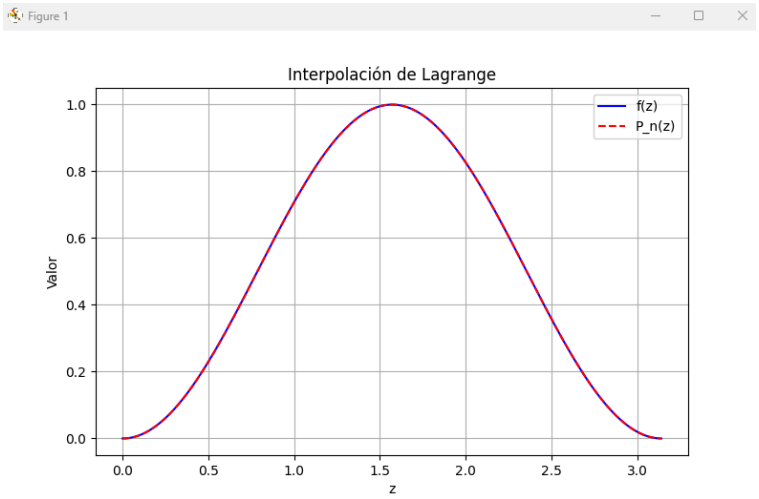




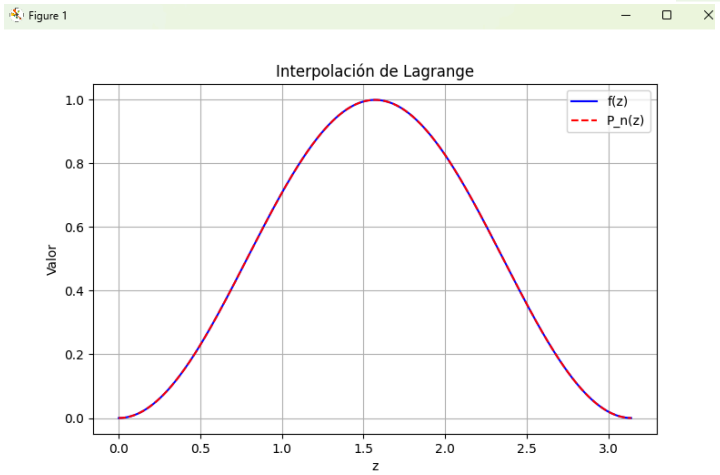
Para n=10

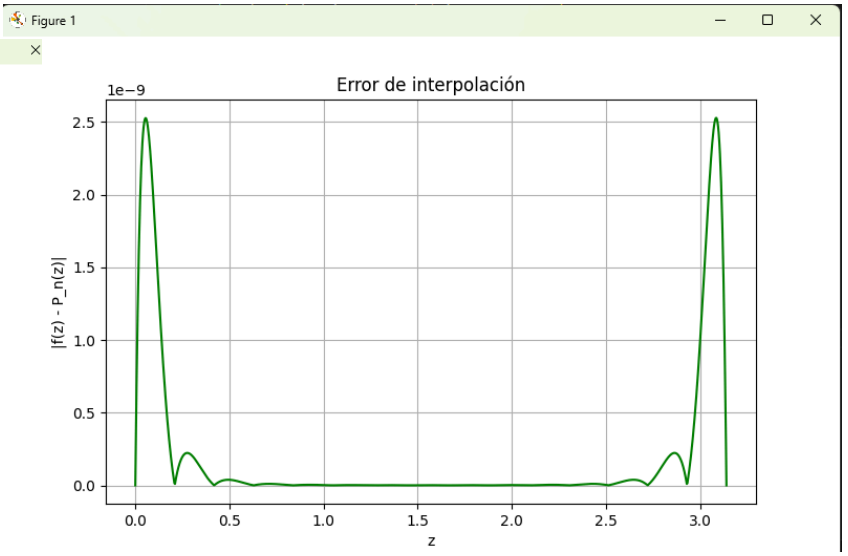


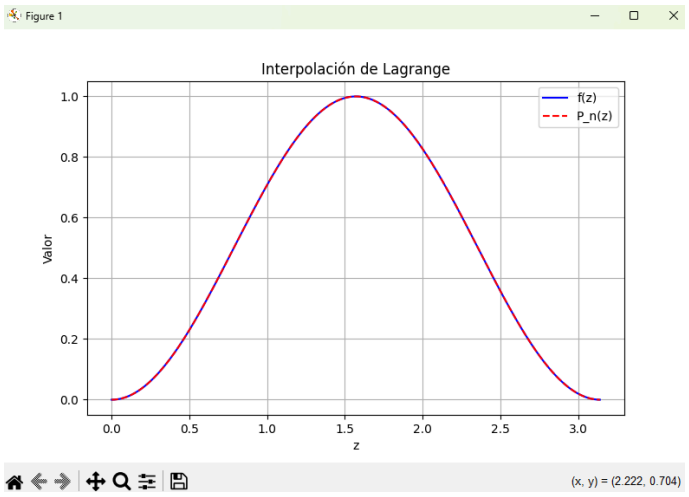


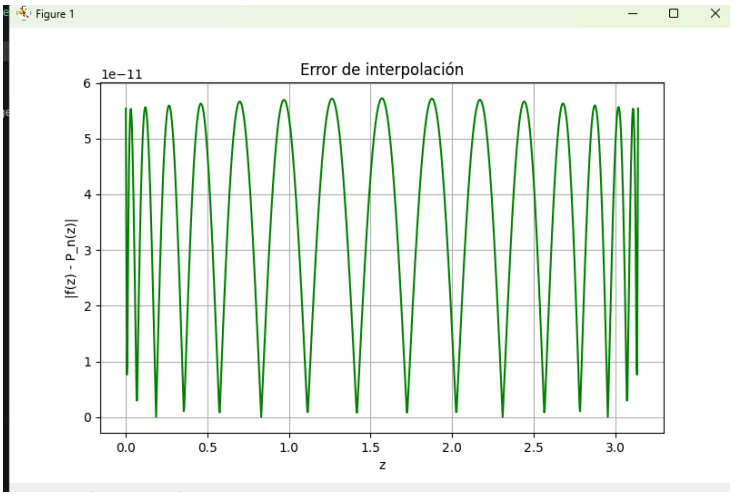


PARA N=15:









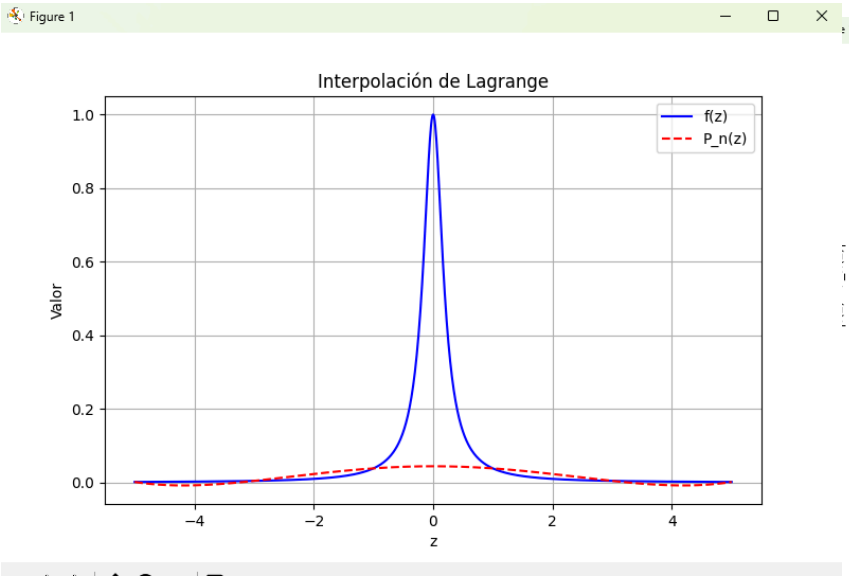
Para esta función de sin() se puede ver en el interpolación que como una podría esperar con los puntos equidistantes a los extremos falla bastante mas en comparación a los

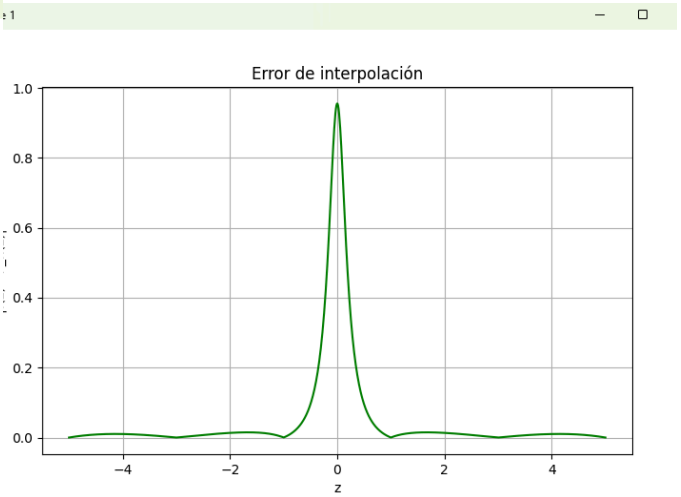
extremos de la función y mediante vamos subiendo la n es más notoria la falla en los extremos de la función y la zona céntrica va siendo menor el error y cada 5 en n el error disminuye en escala 1e-3.
Mientras con chebyshev el Error es más repartido es semblante en extremos como en medios pero de una escala menor con cada 5 en n la escala disminuye en 1e-4.

Conclusión: Para crear un polinomio para puntos cerca del punto que queremos analizar los puntos equidistantes es una buena opción pero con más lejos nos vamos del punto z es más inestable la función creada. Para eso está chebyshev que a principio tiene un error en la zona media mayor a los puntos pero es más estable y regular a lo largo de la función

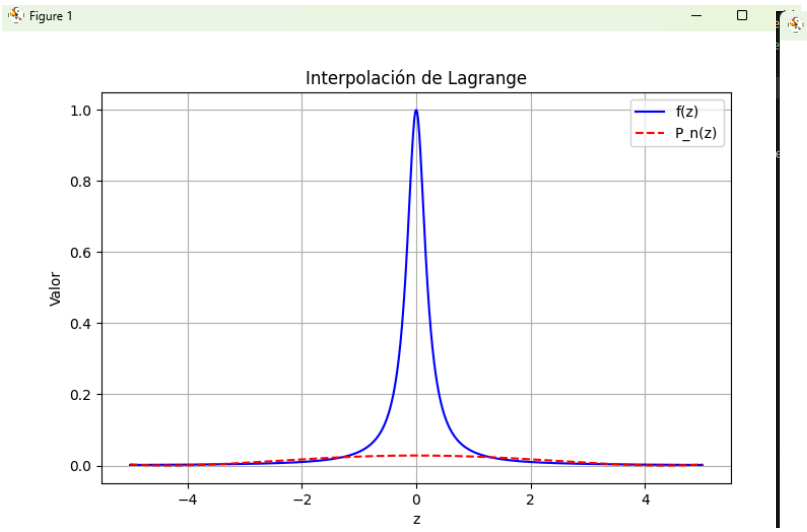
FUN() runge:

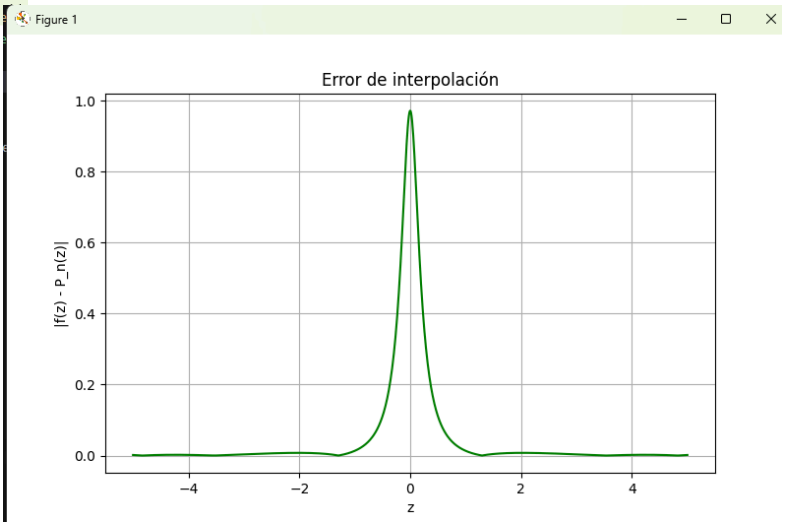
Para n=5
equidistants



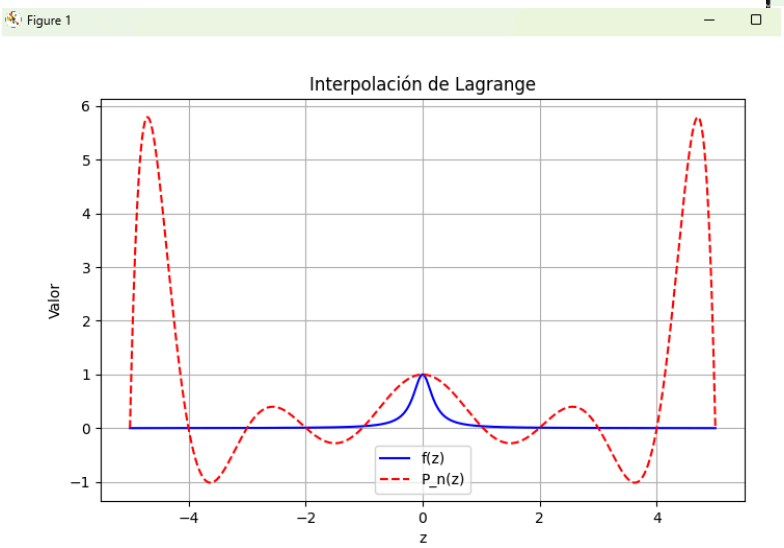


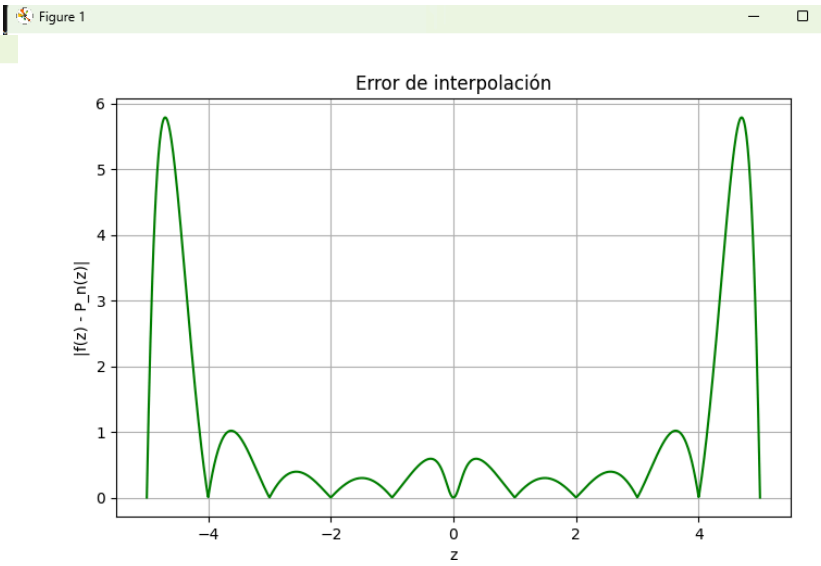
chebyshev:

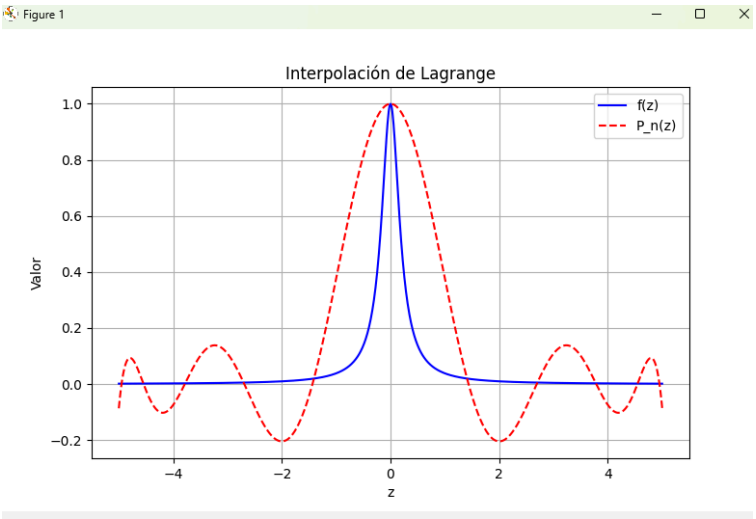


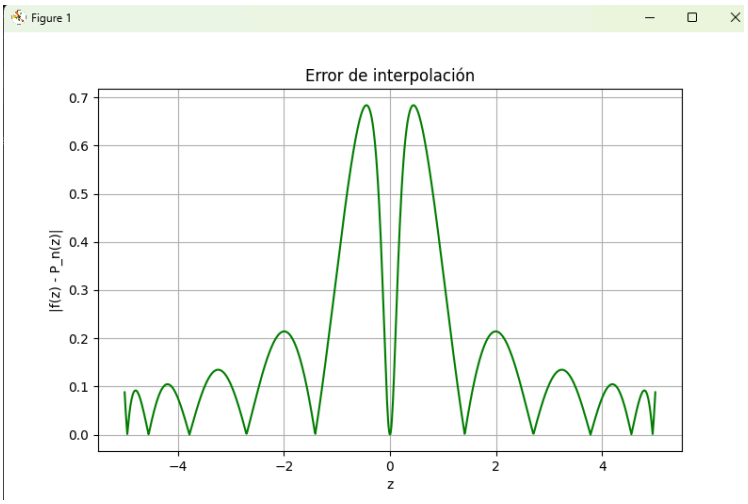


PARA N=10

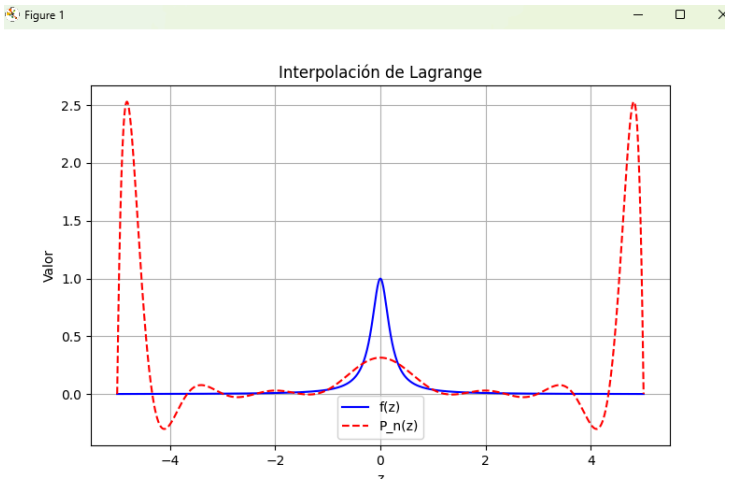


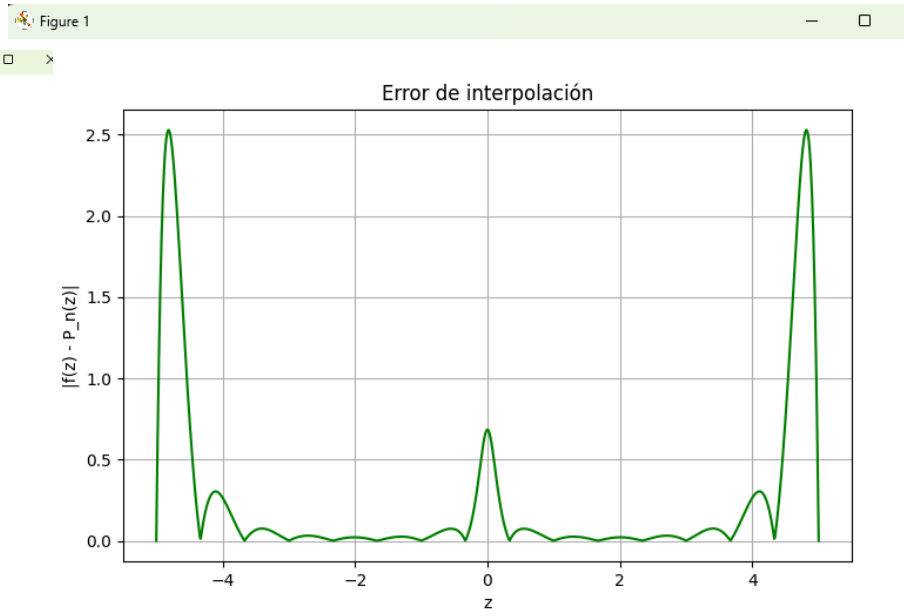


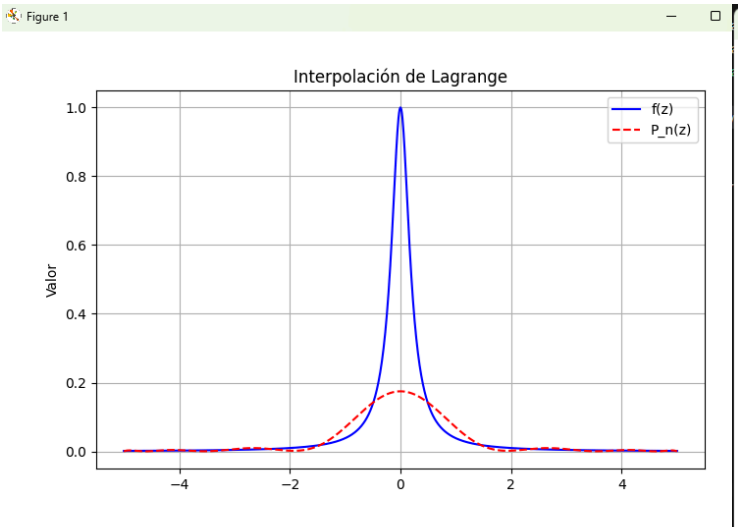


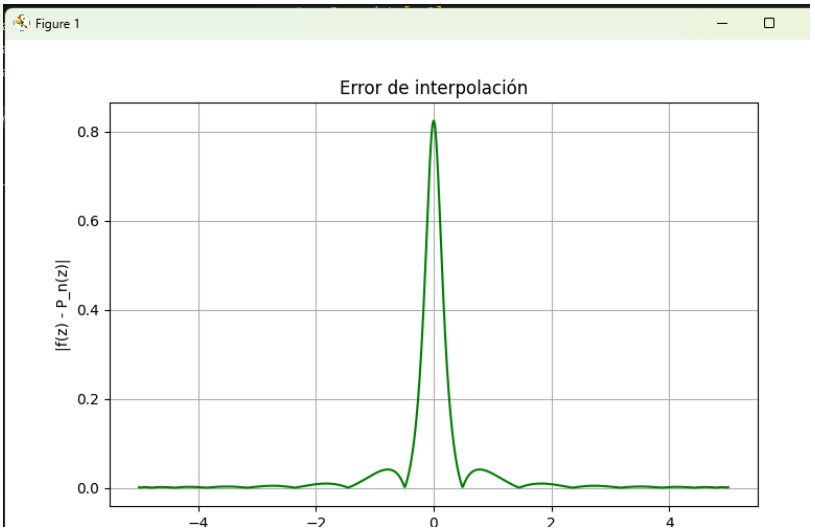


PARA N=15:









comentarios: Aquí podemos analizar que los resultados son mucho más dispares, el error es claramente más notable y que ninguno de los dos puntos puede generar una función que sea exactamente igual o casi igual a la original. Pero notamos que claramente chebyshev sigue siendo más efectivo en general porque el error de los equidistantes en los extremos y en el centro es muy notable

Conclusión: Para esta función que es más inestable debido a que es de la familia 1/x los dos métodos no son del todo efectivo para representar una función similar pero en los extremos chebyshev es muy efectivo pero con mas al centro se va notando el error con más n mayor

Conclusión final: Para analizar funciones aproximadas en puntos céntricos a z los puntos equidistantes es una buena forma pero si queremos crear una función parecida a l original chebyshev es la mejor opción porque mantiene un error más estable para funciones “suaves” como más “complejas”

1.2. Interpolación Polinómica: Método de Newton

Primero se nos da el objetivo que es hacer el método de Newton para ello me pide implementar horner para calcular el polinomio en un punto z, para ello, en el documento de Introducción a C buap horner que está implementado como auxiliar de un main con otras variables, copie horner y tuve que adaptarlo a la forma que se me pide de las variables, cambiando solo la parte de cálculo con las variables que necesito, aplicando la fórmula dada por `resultado * (z - x[i]) + c[i];` saco el valor y lo devuelvo.

Para difdiv busque la fórmula por internet y encontré:

$$f[x_i, \ldots, x_{i-j}] = \frac{f[x_i, \ldots, x_{i-j+1}] - f[x_{i-1}, \ldots, x_{i-j}]}{x_i - x_{i-j}}$$

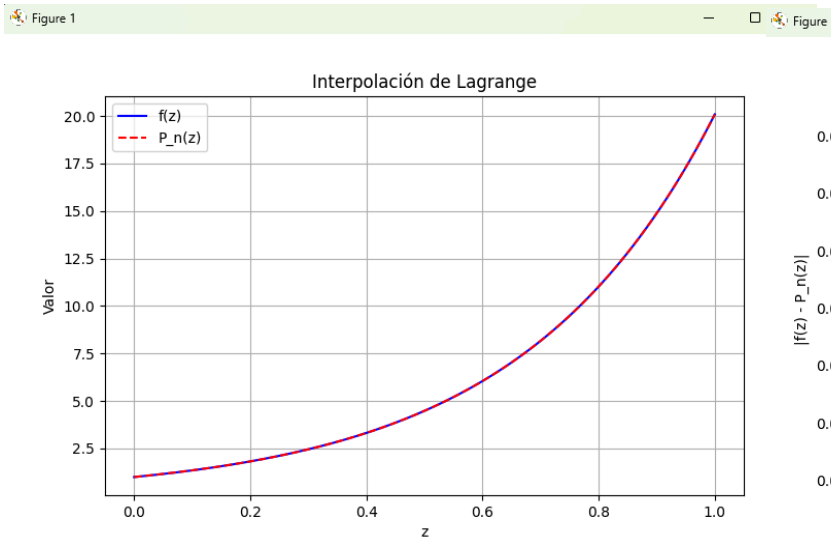
y para codigo tuve que usar un método “in place” para evitar el uso de listas extras que ocupen memoria extra, hice que primero empecemos calculando y[i] y vayamos calculando hacia atrás es decir y[i]-y[i-1] para la diferencia de valores y vaya creando y modificando la lista y para no evitar la lista como dije. Pero primero necesitaba calcular el denominador para poder comprobar que cumpla la tolerancia especificada. Una vez obtenido mediante el paso x[i]-x[i-j]] donde básicamente j es el orden de magnitud a la inversa para poder hacer la diferencia con el faltante del caso que estemos haciendo, ejemplo si tengo f[x1,x2]] con j usaremos x0 para poder completar la x.

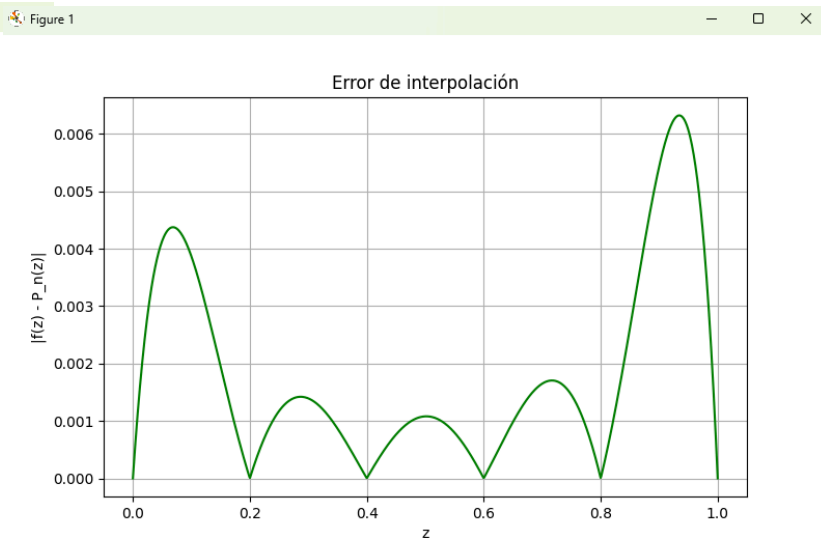
Una vez hecho las funciones era implementar el main que no tiene misterio ni complicación, copie el contenido usado en main_lagrange.c y lo pegue en el nuevo modificando la parte final que es el cálculo requerido porque lo demas anterior era igual, lectura, memoria y comprobaciones.
Primero vemos si el difdiv da un error sino continuamos. Código de archivo lo mantengo igual a como lo tenía, en el bucle de los 1000 puntos hacemos lo mismo solo que en lugar de llamar a lagrange llamamos a horner y guardamos resultados en el archivo.

ACLARACIÓN: ME OLVIDE CAMBIAR EL TÍTULO DE LAS GRÁFICAS Y POR ESO SALE LAGRANGE Y NO NEWTON, PERO ES NEWTON EL MÉTODO USADO, ME DI CUENTA TARDE Y NO TENÍA TIEMPO PARA CORREGIRLO

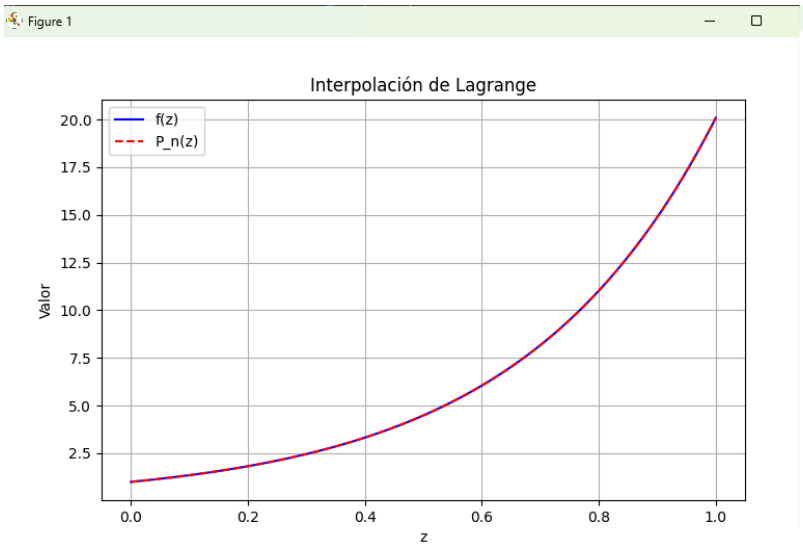
Resultados exp():

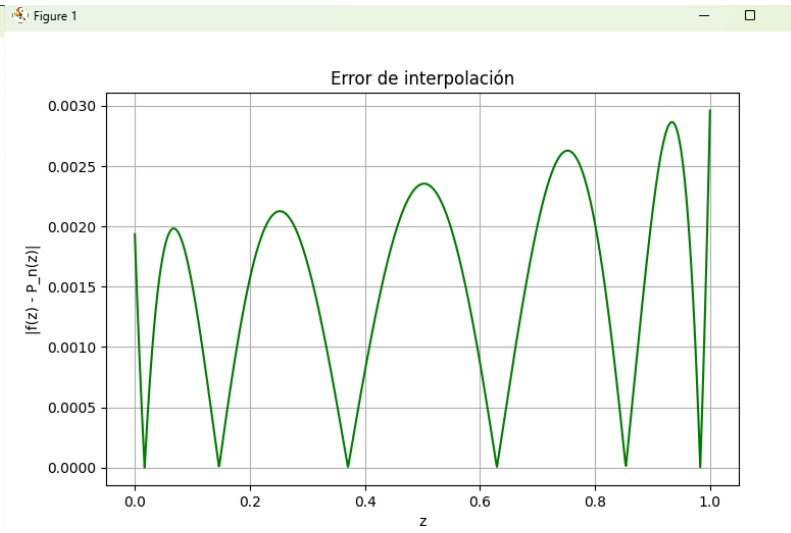
PARA N=5
Equidistantes:





Chebyshev:

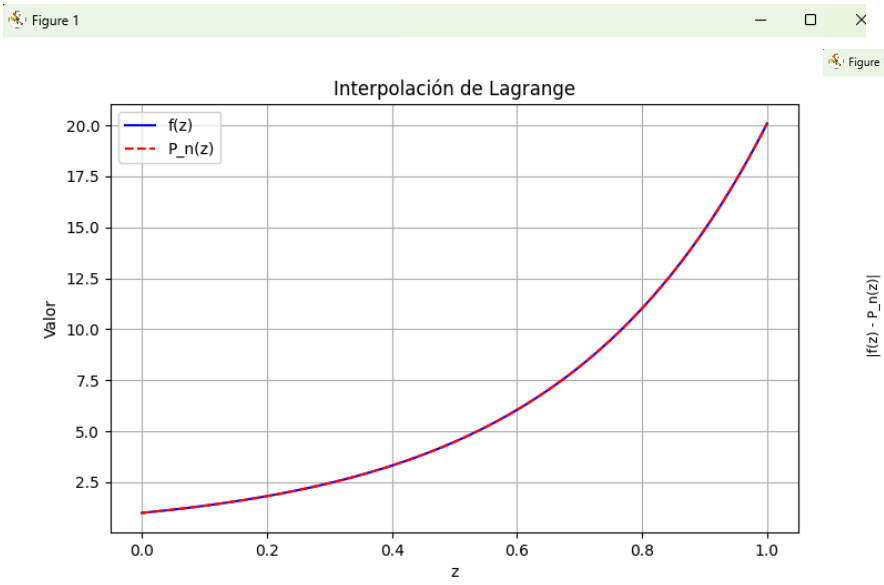


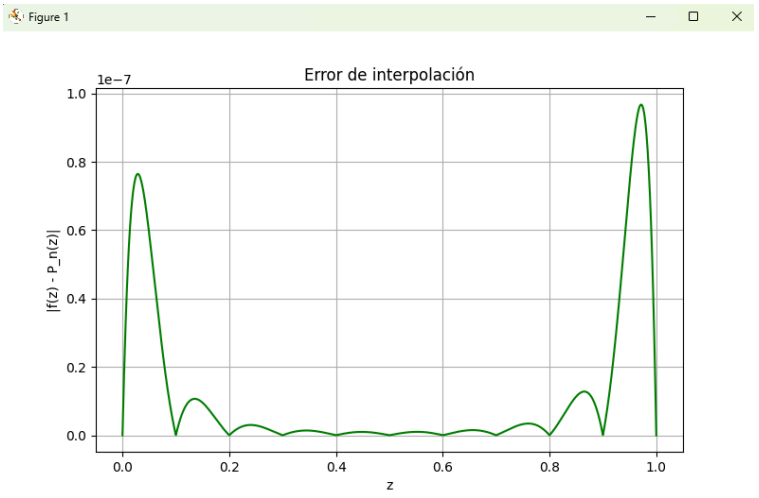


Observaciones: Como pasaba con el primer interpolador, vemos que Chebyshev es una opción más buena para aproximación de función a lo largo de la función ya que el pico de error producido al final es la mitad que si usamos puntos equidistantes de 3e-3 y 6e-3 respectivamente y como ya sabíamos puntos equidistantes es mayor precision en el centro de la ecuación.

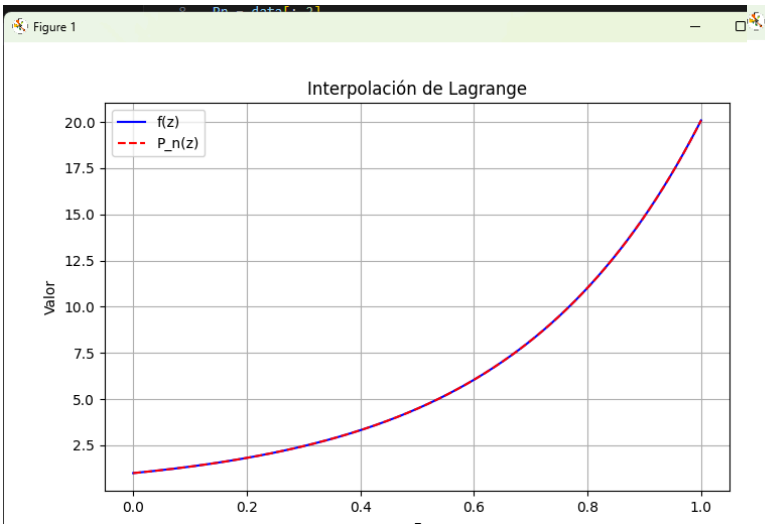
PARA N=10:

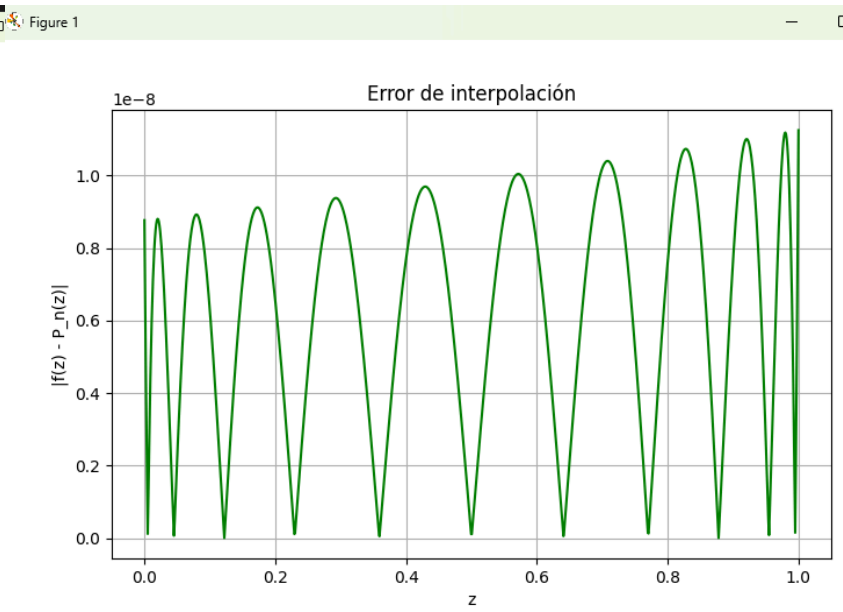
equidistantes:





Chebyshev:

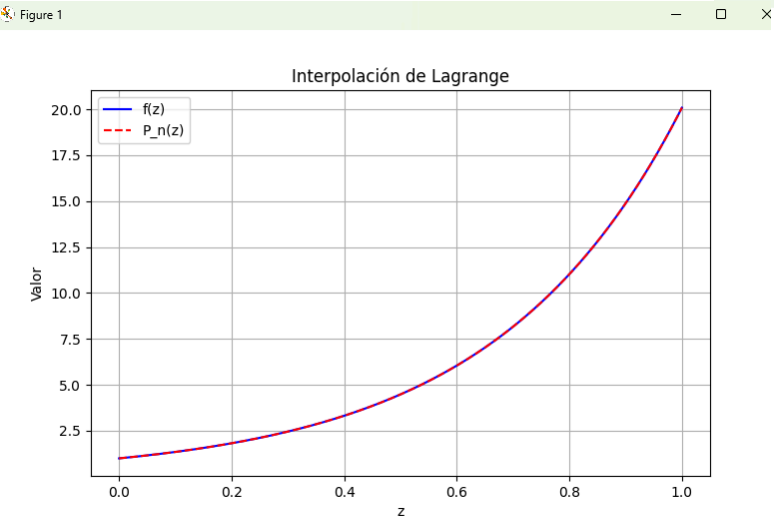




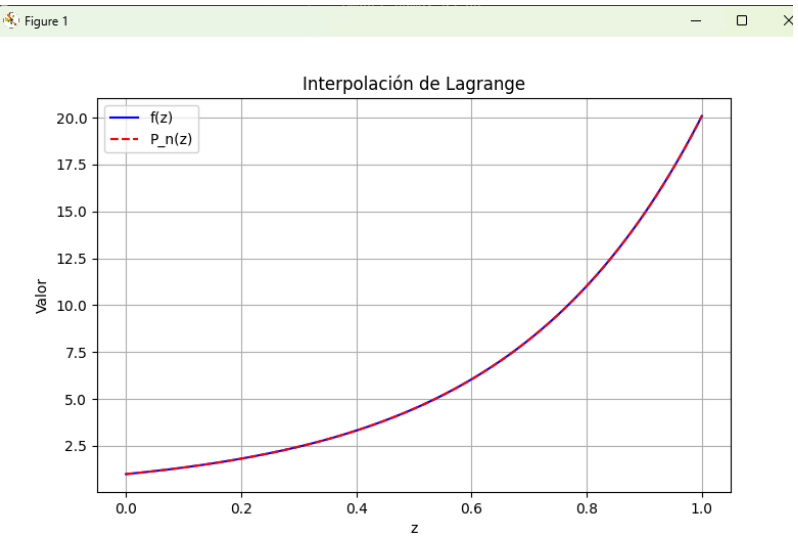
Observación: Misma regla, chebyshev es mejor para la función en general teniendo un 1e-1 masque con puntos equidistantes

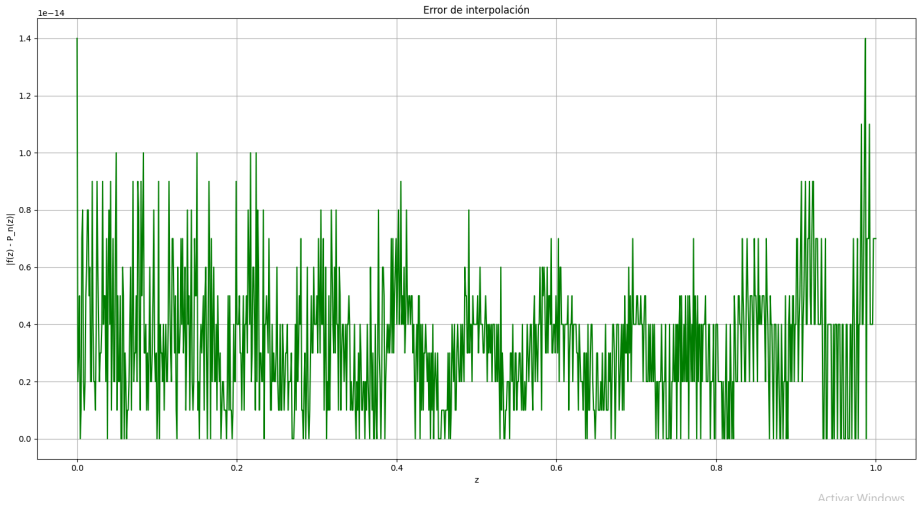
PARA N=15:

equidistantes:



Chebyshev:



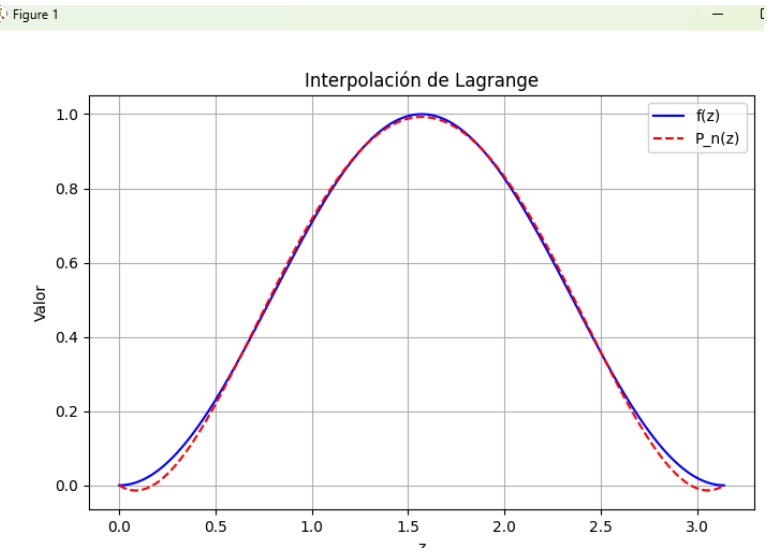


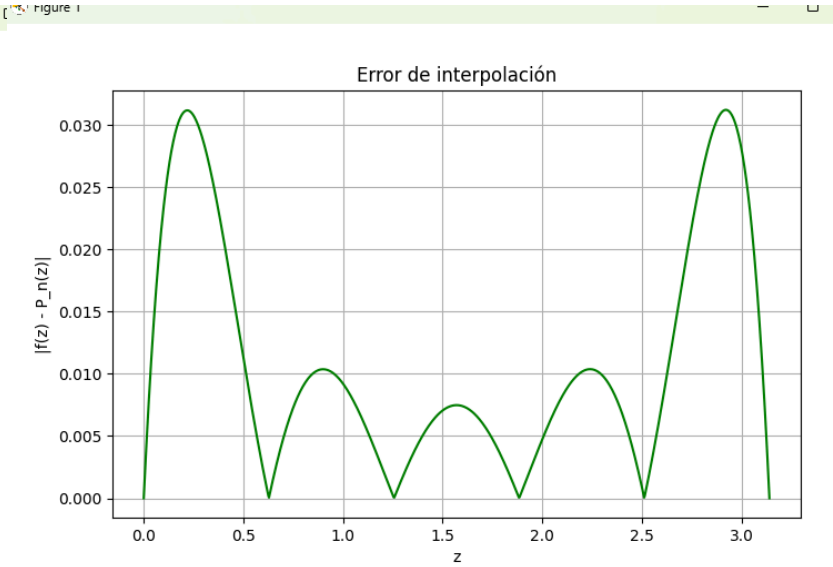
Observación: Aqui vemos que la gráfica de error de Chebyshev ha pasado de ser mas de ondas similares o lo que parecería una frecuencia con interferencias.Sigue siendo un error menor al de puntos equidistantes pero me parece un comportamiento curioso

RESULTADOS SIN^2():

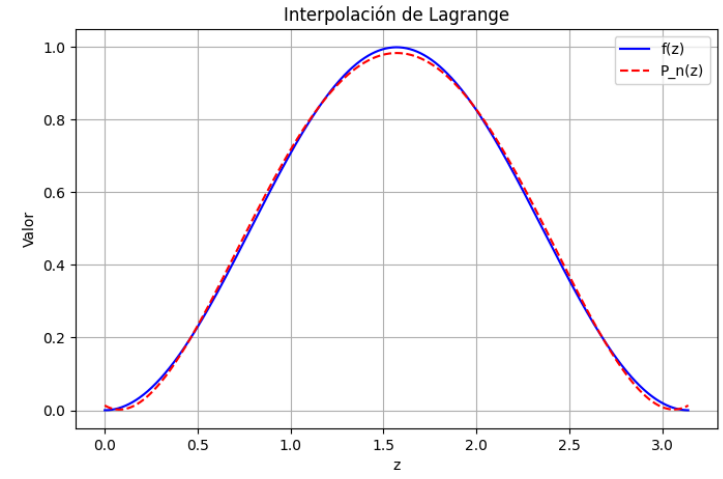
para n=5:

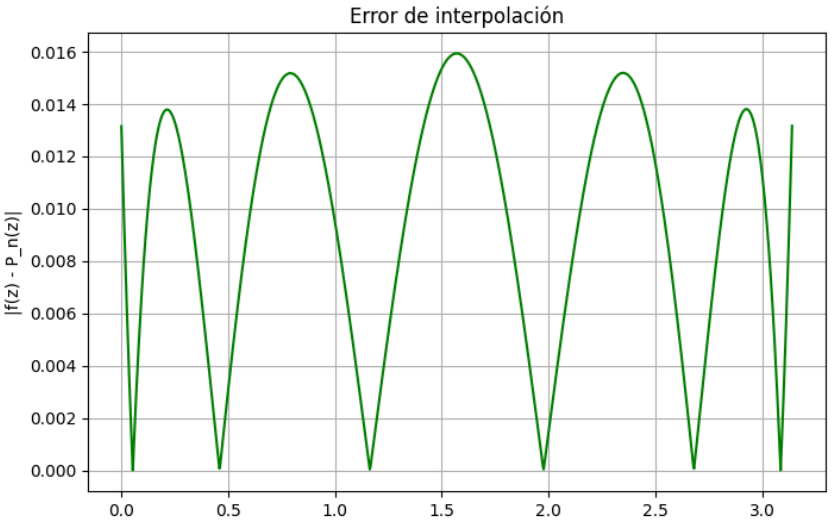
equidistantes:





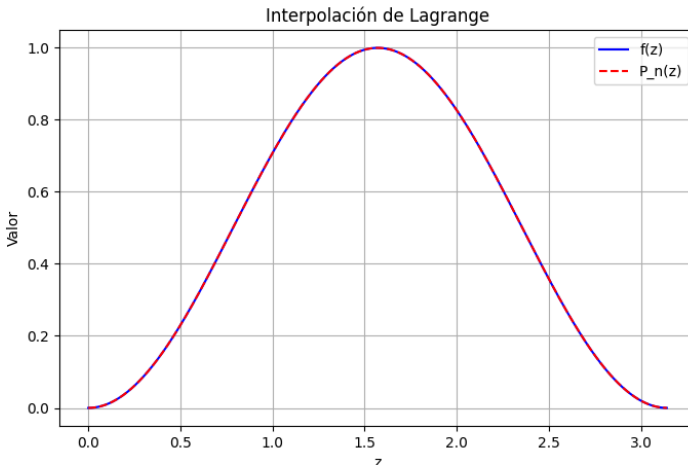
Chebyshev:

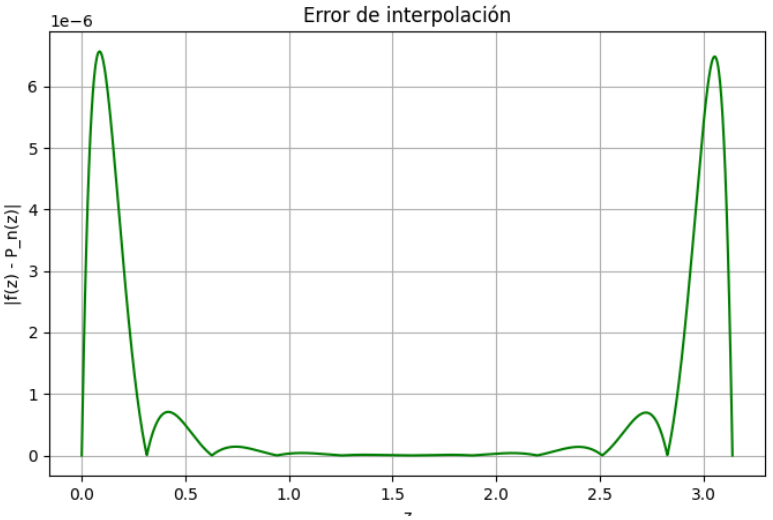




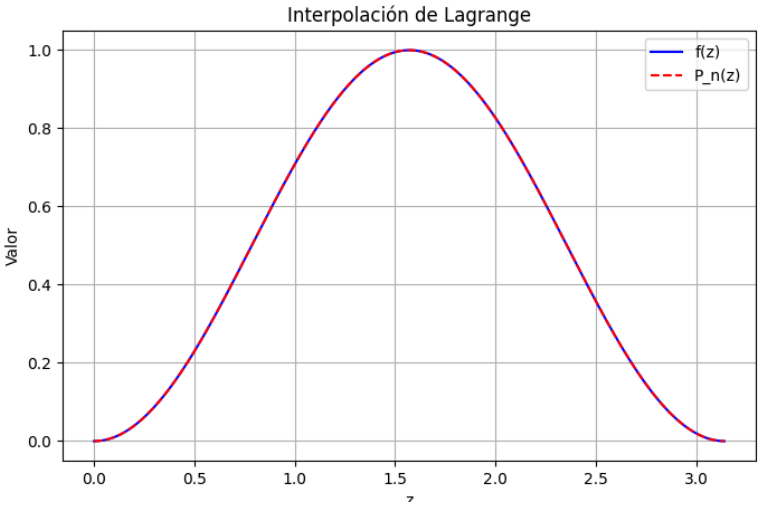
para n=10:

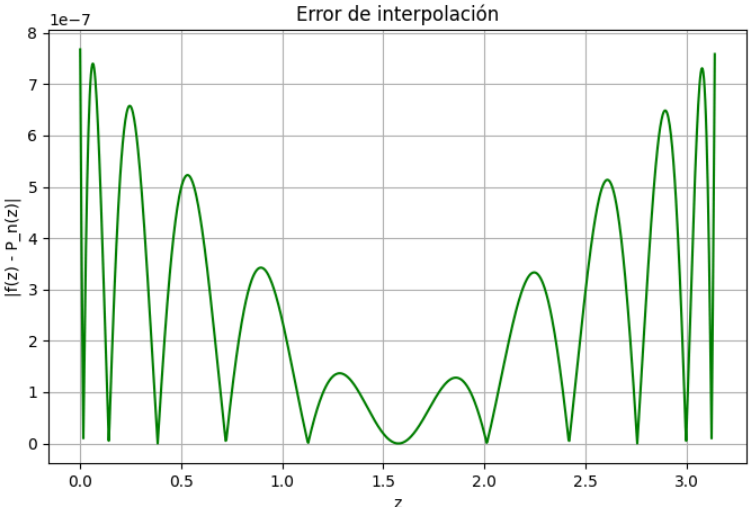
equidistantes:





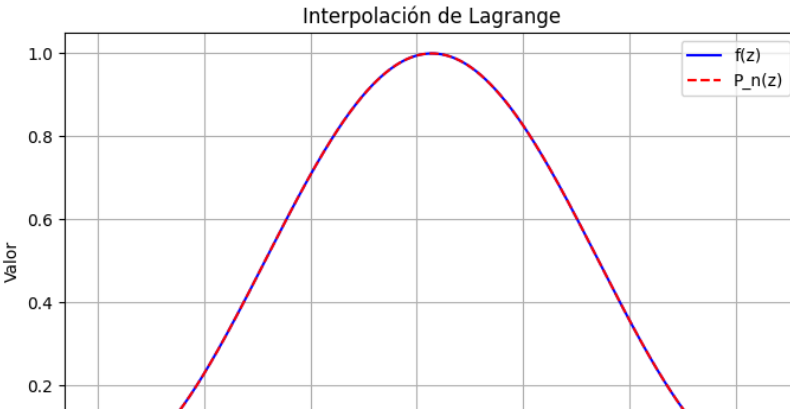
Chebyshev:

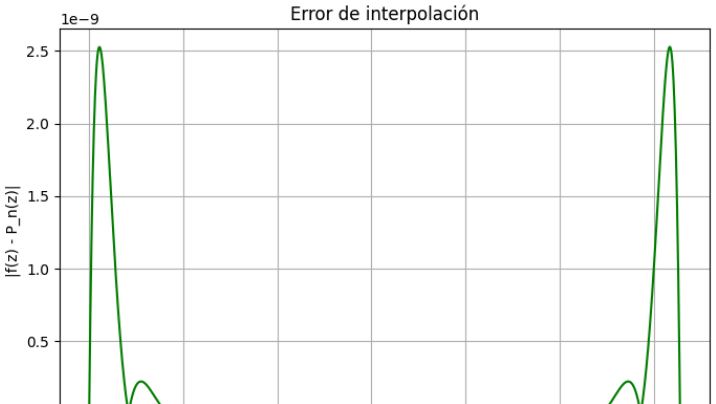




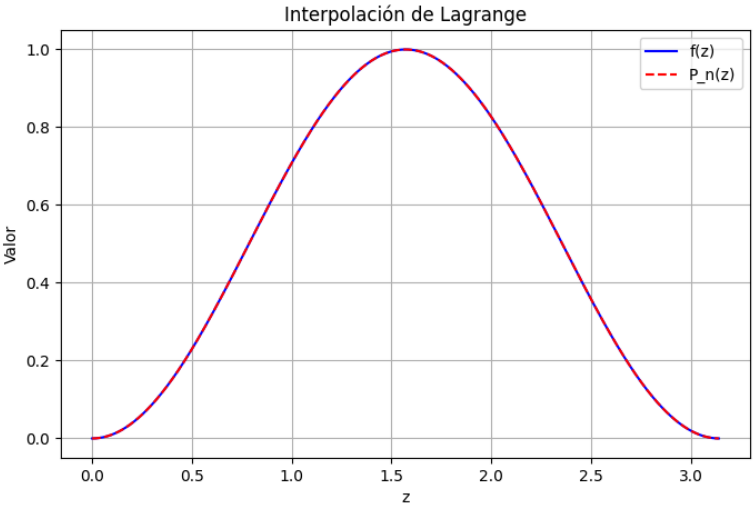
para n=15:

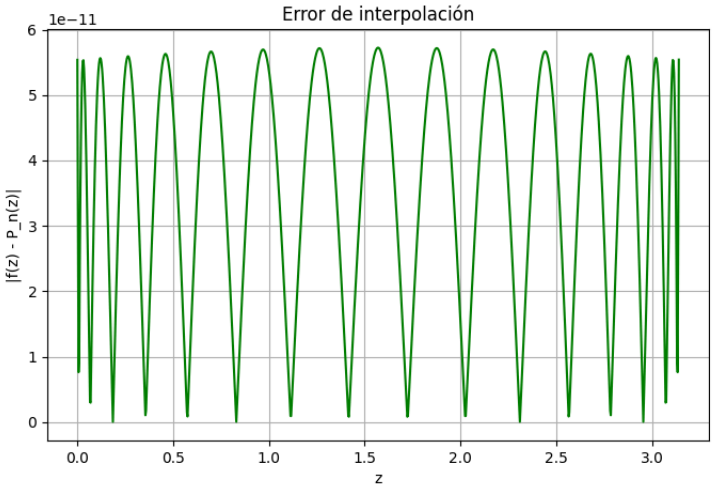
equidistantes:





Chebyshev:



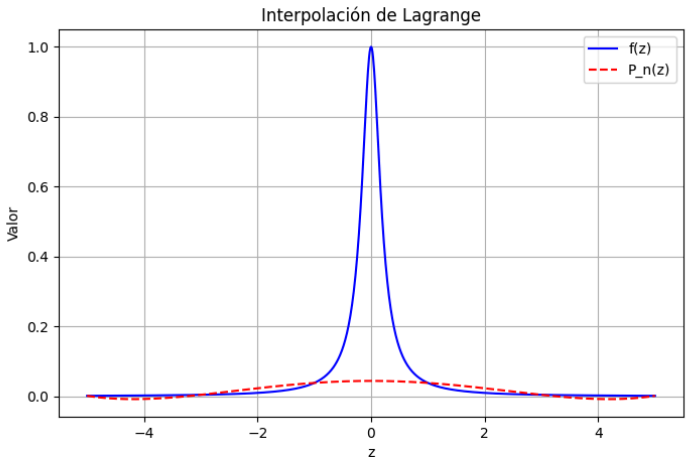


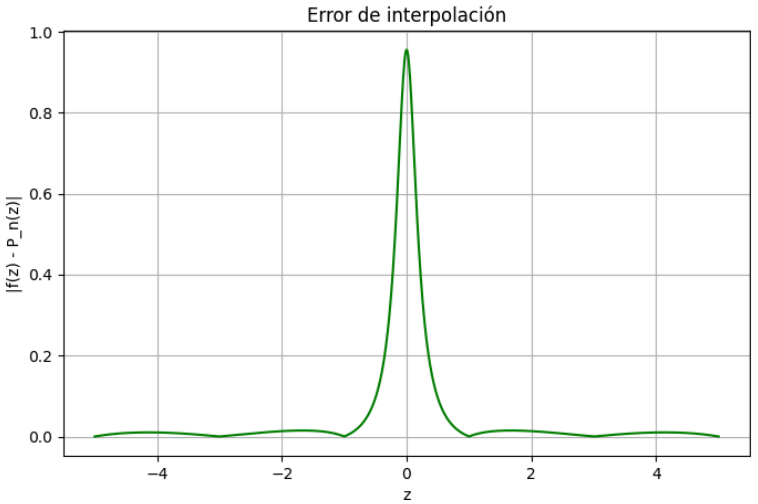
Conclusión: Tanto para interpolador como con Newton los puntos mediante Chebyshev es la mejor manera porque cuan mayor es el nivel de n la magnitud de error es menor y más estable a lo largo de la función

FUNCION runge():

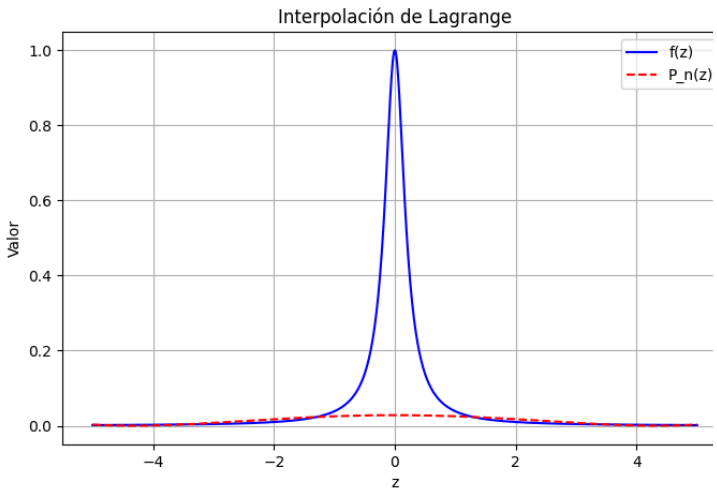
para n=5:

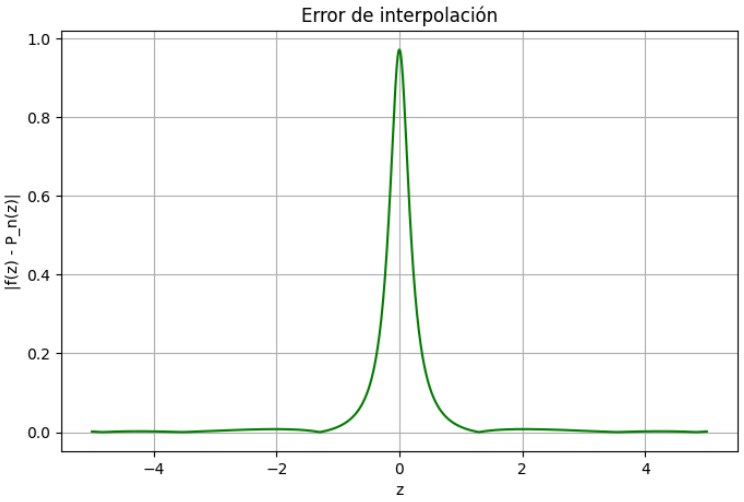
equidistantes:





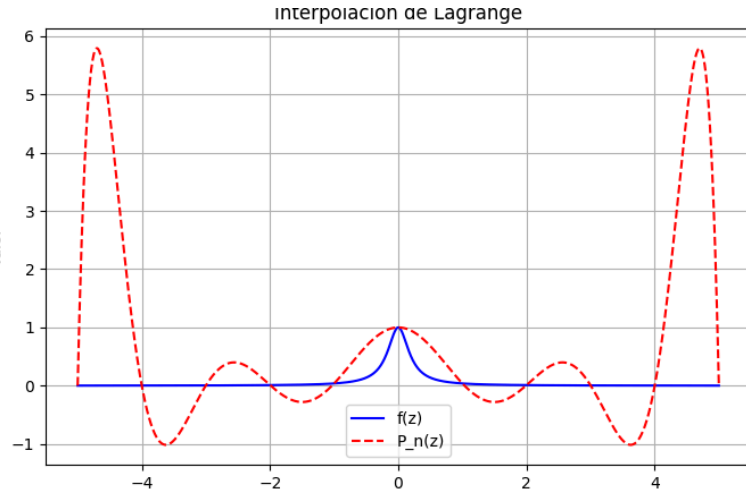
Chebyshev:

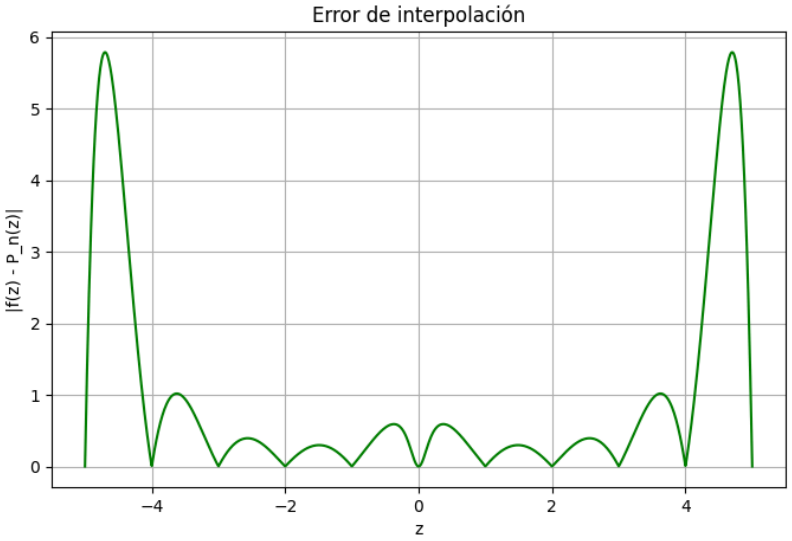




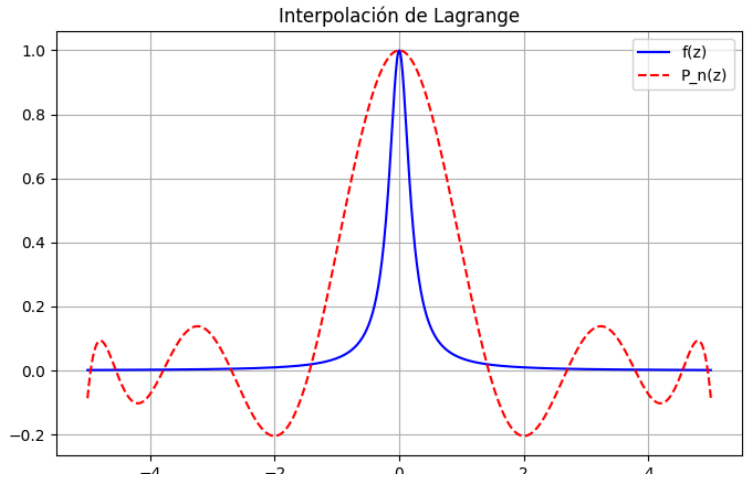
para n=10:

equidistantes:



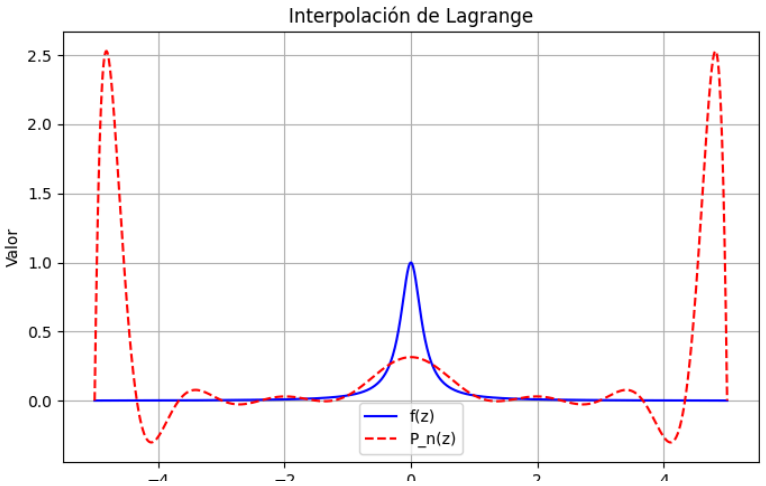


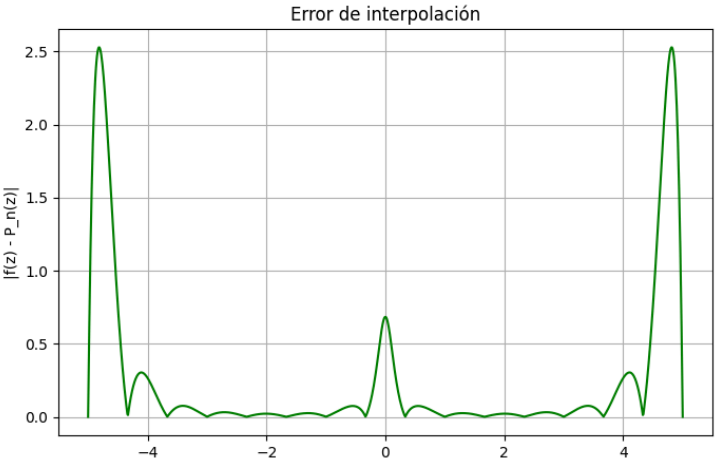
Chebyshev:



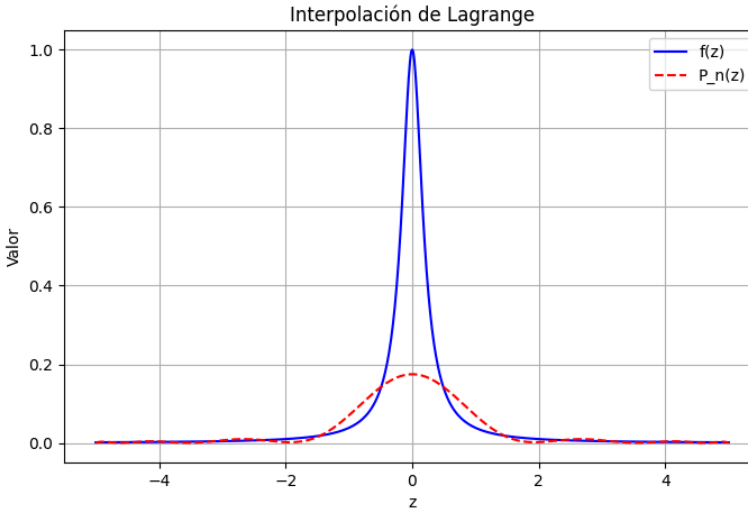
para n=15:

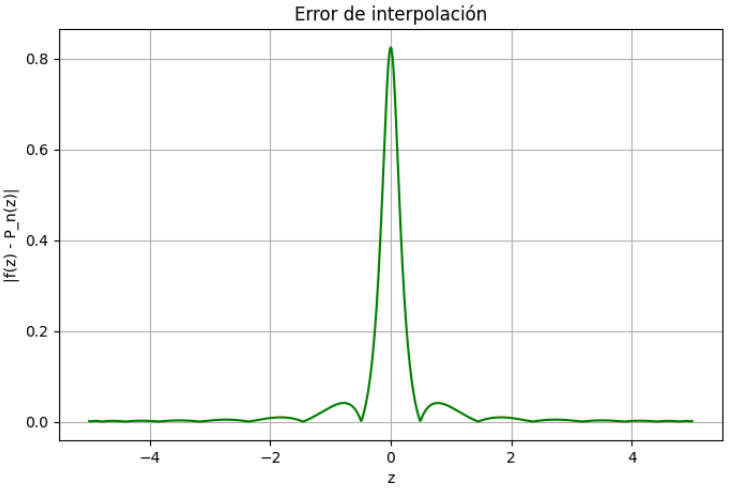
equidistantes:





Chebyshev:





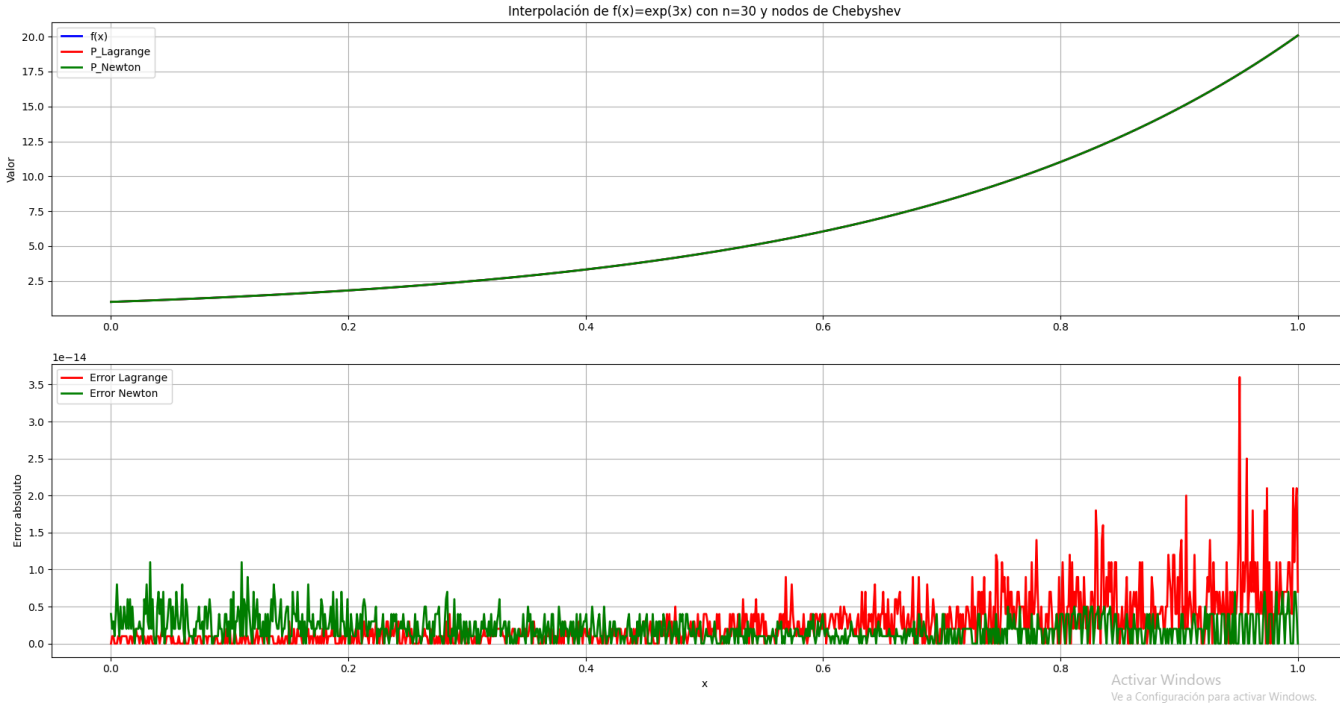
Conclusión: Igual que pasaba con el polinomio interpolador inicial los pu tos hechos cupón Chebyshev son mejores para hacer la función más estable a lo largo de los puntos próximos a z pero ambos puntos son bastantes inestables en el contorno de 0 en funciones de familia 1/x por el peligro que pasa cuando x es = q tiende a crecer

1.3. Comparación entre los dos métodos

Para código fue tan fácil como copiar el comienzo que ambos archivos comparten como lectura memoria y eso de datos.
Pegamos la parte de cálculos de cada uno y los juntamos en un mismo bucle para calcularlos a la vez en la iteración y no hacer 2 bucles de 1000. Añadimos el comparador de error para buscar y al final quedarnos con el error máximo de las funciones para poder comparar resultados.

RESULTADOS:

Max error Lagrange = 0.000000000000036
Max error Newton = 0.000000000000011



Observaciones: Podemos ver las funciones a priori parecían ser iguales o no perceptibles en el gráfico de arriba que compara las funciones aproximadas con la original.
Pero lo importante está en el gráfico que nos indica el error, se aprecia que con los puntos que van desde 0 a x_maxima/2 el método de lagrange es más efectivo que Newton pero de x_maxima/2 hasta x_maxima es claro y muy visible que Newton es mucho mejor que Lagrange porque como se ve cuando Newton era peor sacó un error de 11e-15 máximo mientras que Lagrange de 36e-15 el triple de valor

¿Por qué el método de Newton proporciona una aproximación mejor?

Porqué la forma de calcular cada uno de los métodos influye, debido que para lagrange dentro del bucle de 1000 utiliza la fórmula para generar el polinomio en z, pero esta fórmula depende de divisiones, entonces con mayor el número de iteración vamos arrastrando el error de cada división en el sumatorio porque se usan sumatorios de multiplicaciones que vienen de divisiones por cada i.
Mientras Newton solo hace el cálculo que requiere división una vez entonces no sufre ese arrastrar de error de decimales.

Conclusión: Newton es mejor debido que solo hace una llamada a un cálculo que requiere dividir y no arrastra error a lo largo de las iteraciones(puntos) y Lagrange arrastra errores.

2. Integración numérica

El objetivo de este apartado es aproximar

$$I = \int_0^1 e^x dx = e - 1.$$

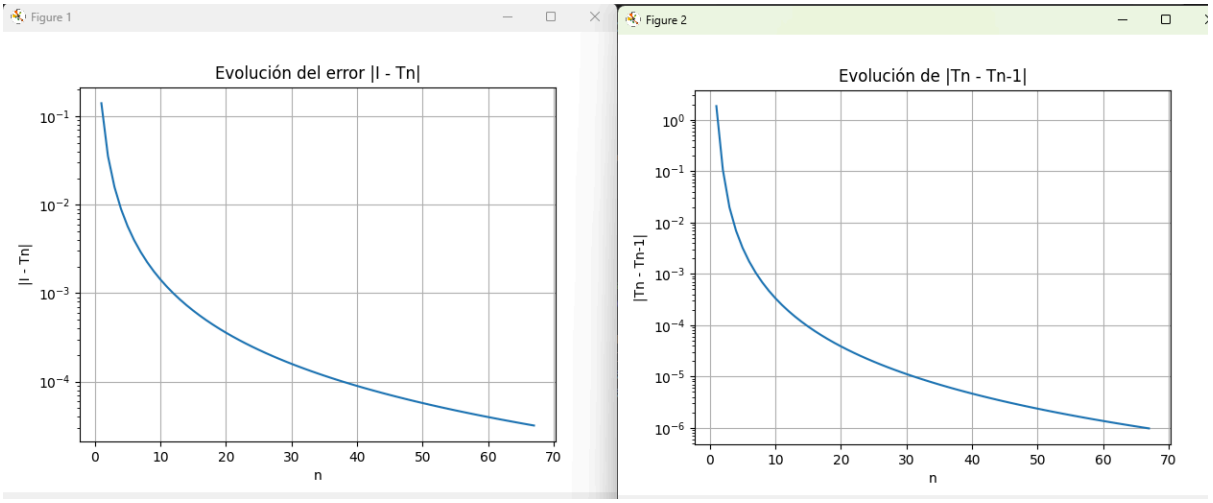
2.1. Regla compuesta trapezoidal “naive”

Para implementar trap_full; hice:

```c
double trap_full(int n, double a, double b){
    double h = (b - a) / n;
    double integral = 0.5 * (fun(a) + fun(b));
    for (int j = 1; j < n; j++) {
        double x_i = a + j * h;
        integral += fun(x_i);
    }
    integral *= h;
    return integral;
}
```

Primero calculo la h que usare para la fórmula, calculo aparte extrayendo factor comuun de ½ por la fun(a)+fun(b), con un bucle que va de 0 a n-1 que calcule la Xj y su sumatorio con f(Xj) para finalmente multiplicar la h para obtener la aproximacion.

Resultados:



Explicado en la memoria por qué la condición |Tn − Tn−1| < 10−6 no constituye un buen criterio de parada para el algoritmo.
La razón es por la misma explicación de que representa cada cosa.

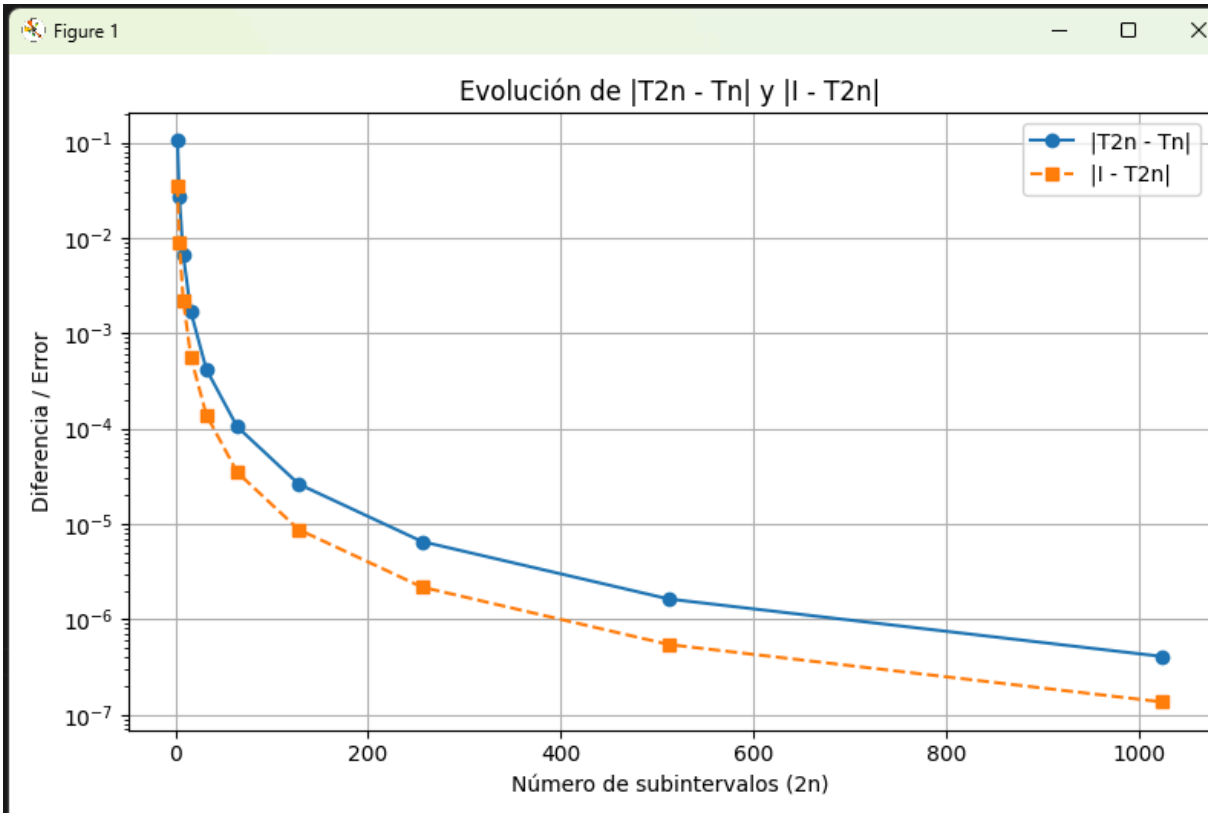
```
|Tn - Tn-1|
```

Representa la diferencia entre la iteración anterior y actual mientras que |I -Tn| Es la diferencia entre el valor REAL y la que aproximamos nosotros.
La lógica que seguimos en nuestro algoritmo es para la iteración cuando la diferencia entre nuestra aproximación anterior y actual es menor a 1e-6. Comparamos qué diferente es nuestra aproximación respecto a la misma, cuando la diferencia sea menor a la dada damos por hecho que es parecido al valor exacto pero eso no implica que sea cierto, porque vemos que la diferencia de la función real con nuestros resultados está en 1e-4, bastante considerado respecto a lo que queríamos.

2.2. Regla compuesta trapezoidal

Para la implementación de trap refined(); he copiado el contenido de trap full y cambiando donde influye n y j lo multiplico por 2 para poder obtener el T2n.
Y cuando llamamos a la función en el main doblar la n*2 para poder tener el 2n y seguir la regla de los subintervalos y el resto es prácticamente lo mismo que el otro main.

Explica en la memoria por qué main_trapez.c es mucho mejor que main_trapez_naive.c. y por qué en este caso |T2n − Tn| < 10−6 es un buen criterio de parada para el algoritmo.



Como podemos apreciar en el gráfico gracias a la refinación de no recalcular los puntos ya hechos nos da un margen mayor, me refiero, en el método anterior apenas llegaba a 1e-6 se detenía y nos quedaba una diferencia bastante significante mediante a la real, pero con este nuevo método vemos que como solo añade los números nuevos nos queda un margen inferior a la condición mayor, es decir, menor diferencia entre los intervalos de la iteración entre T2n a Tn y hace que T2n es más próxima que la integral real, vemos que se aproxima el error real a 1e-7 es magnitud de 1e-3 menor respecto a lo que teníamos con el otro método.

Es un buen criterio de parada en este método porque como comparamos T2n respecto Tn el número de subintervalos es mayor que si lo hacemos desde Tn-Tn-1. Con lo cual con la mayor acumulaacion de subintervalos si la diferencia es pequeña, el error respecto al valor real será también más pequeña porque estaremos mucho más cerca a un valor aproximado del real.