Práctica Programación II

Ingeniería técnica en informática de gestión

Diseño, análisis teórico y estudio empírico de las distintas funciones recursivas y función iterativa, para el cálculo aproximado del seno de un número.

Alumno: Manuel Rodríguez Sánchez
Centro Asociado: Motril

Contenido

1. ENUNCIADO DE LA PRACTICA

Cálculo Aproximado del Seno de un Número

El seno de un ángulo, x, expresado en radianes y con $-\pi \leq x \leq \pi$ es una función cuyo valor puede aproximarse por un desarrollo en serie de potencias de Taylor. La fórmula general de dicho desarrollo para el cálculo de f(x) en un punto (a) es

$$f(x) = \sum_{i=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n,$$

donde *n!* representa el factorial del número n y $f^{(n)}(a)$ representa la *n*-ésima derivada de f en el punto a. El desarrollo de Taylor para el seno, tomando *a = 0* sería

$$\sin x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!}x^{2n+1}, \qquad \forall x.$$

Dado que se trata de una serie infinita (el polinomio podría evaluarse utilizando sus infinitos términos), tendremos que aproximarla por la suma de un cierto número finito de términos, k, es decir,

$$\sin x \approx_k \sum_{n=0}^{k} \frac{(-1)^n}{(2n+1)!}x^{2n+1}$$

Se desea diseñar una función recursiva completamente verificada que calcule la aproximación de orden *k* al valor del seno en un punto *x* dado mediante la serie de potencias de MacLaurin.

Por ejemplo, la aproximación de orden 3 al seno de $(\frac{\pi}{6})$

$$\sin\left(\frac{\pi}{6}\right) \approx_3 \sum_{n=0}^{3} \left(\frac{(-1)^n}{2\cdot n+1}\left(\frac{\pi}{6}\right)^{2\cdot n+1}\right)$$

o, lo que es lo mismo,

$$\sin\left(\frac{\pi}{6}\right) \approx_3 \left(\frac{(-1)^0}{(2\cdot 0+1)!}\left(\frac{\pi}{6}\right)^{2\cdot 0+1}\right) + \left(\frac{(-1)^1}{(2\cdot 1+1)!}\left(\frac{\pi}{6}\right)^{2\cdot 1+1}\right) + \left(\frac{(-1)^2}{(2\cdot 2+1)!}\left(\frac{\pi}{6}\right)^{2\cdot 2+1}\right) + \left(\frac{(-1)^3}{(2\cdot 3+1)!}\left(\frac{\pi}{6}\right)^{2\cdot 3+1}\right)$$

El diseño de la función que constituye la parte teórica de la práctica debe ser genérico, es decir, debe tratar todos los factores (por ejemplo, el punto en que se calcula el coseno o la bondad de la aproximación, es decir, el número de términos de la serie que lo aproxima, entre otros) como parámetros, sin importar cómo se instancien para resolver el problema (lo que se hará en la implementación de la práctica).

2. DISEÑO Y ANÁLISIS TEÓRICO

2.1 Especificación formal de la función sink

$\{Q \equiv (-\pi \leq x \leq \pi) \wedge (k \geq 0)\}$

fun sink (x : real; k : natural) dev (y : real)

$\{R \equiv y = \sum_{n=0}^{k}[(-1^n) \cdot x^{2n+1}/(2n+1)!]\}$

2.2 Expresión de sink respecto a tmno

$\{Q \equiv (-\pi \leq x \leq \pi) \wedge (k \geq 0)\}$

fun sink (x : real; k : natural) dev (y : real)

$\{R \equiv y = \sum_{n=0}^{k}[pot(-1,n) \cdot tmno(x,2n+1)]\}$

2.3 Análisis por casos de la función sink

Condición del caso trivial	Solución al caso trivial
k=0	*y=x →La solución será la propia x*
Condición del caso no trivial	Solución al caso no trivial
k>0	$y = \sum_{n=0}^{k}[(-1^n) \cdot x^{2n+1}/(2n+1)!]$

2.4 Composición algorítmica para sink

$\{Q \equiv (-\pi \leq x \leq \pi) \wedge (k \geq 0)\}$

fun sink(x: real ;k: natural) dev (y: real)

caso k = 0 → y = x

□ k > 0 → y = pot (-1,k) · tmno (x,2k+1) + sink (x,k-1)

fcaso

ffun

$\{R \equiv y = \sum_{n=0}^{k}[pot(-1,n) \cdot tmno(x,2n+1]\}$

2.5 Verificación formal de la corrección

Completitud de la alternativa.

Se trata de ver que el conjunto de protecciones cubre todos los casos posibles, formalmente:

$\boldsymbol{Q}(\overline{\boldsymbol{x}}) \Rightarrow \boldsymbol{B_t}(\overline{\boldsymbol{x}}) \vee \boldsymbol{B_{nt}}(\overline{\boldsymbol{x}})$
$(-\pi \leq x \leq \pi) \wedge (k \geq 0) \Rightarrow (k = 0) \vee (k > 0)$

Las dos partes han de ser verdaderas, y se cumple siempre que x esté acotada en el rango, k va a ser 0 o mayor que 0.

Satisfacción de la precondición por la llamada recursiva.

Esto quiere decir que la función debe llamarse en estados que cumplan la precondición, expresado formalmente debe demostrarse que: $\boldsymbol{Q}(\overline{\boldsymbol{x}}) \wedge \boldsymbol{B_{nt}}(\overline{\boldsymbol{x}}) \Rightarrow \boldsymbol{Q}(\boldsymbol{s}(\overline{\boldsymbol{x}}))$

$Q(\bar{x}) = (-\pi \leq x \leq \pi) \wedge (k \geq 0)$
$s(\bar{x}) = k - 1$
$Q(s(\bar{x})) = (-\pi \leq x \leq \pi) \wedge (k - 1 \geq 0)$

Vamos a proceder al desarrollo:

$[(-\pi \leq x \leq \pi) \wedge (k \geq 0)] \wedge (k > 0) \Rightarrow (-\pi \leq x \leq \pi) \wedge (k - 1 \geq 0)$
$Q(s(\bar{x})) = (-\pi \leq x \leq \pi) \wedge (k - 1 \geq 0) = (-\pi \leq x \leq \pi) \wedge (k \geq 1) = (-\pi \leq x \leq \pi) \wedge (k > 0)$

Finalizando, nos queda:

$[(-\pi \leq x \leq \pi) \wedge (k \geq 0)] \wedge (k > 0) \Rightarrow (-\pi \leq x \leq \pi) \wedge (k > 0)$

Se verifica pues, la satisfacción de la precondición para la llamada interna de sink.

Base de inducción.

Debemos demostrar que la postcondición se satisface para los casos triviales, en términos formales:
$\boldsymbol{Q}(\overline{\boldsymbol{x}}) \wedge \boldsymbol{B_t}(\overline{\boldsymbol{x}}) \Rightarrow \boldsymbol{R}(\overline{\boldsymbol{x}}, \boldsymbol{triv}(\overline{\boldsymbol{x}}))$

$(-\pi \leq x \leq \pi) \wedge (k \geq 0) \wedge (k = 0) \Rightarrow \sum_{n=0}^{k}[(-1^n) \cdot x^{2n+1}/(2n+1)!]$

$(-\pi \leq x \leq \pi) \wedge (k = 0) \Rightarrow \sum_{n=0}^{k}[(-1^n) \cdot x^{2n+1}/(2n+1)!$ **= x**

La implicación es cierta ya que cuando x está acotada en el intervalo y k=0, el resultado siempre va a ser la propia x.

Paso de inducción.

Hay que probar ahora que la postcondición también se cumple para los casos recursivos.
$\boldsymbol{Q}(\overline{\boldsymbol{x}}) \wedge \boldsymbol{B_{nt}}(\overline{\boldsymbol{x}}) \wedge \boldsymbol{R}(\boldsymbol{s}(\overline{\boldsymbol{x}}), \overline{\boldsymbol{y}}') \Rightarrow \boldsymbol{R}(\overline{\boldsymbol{x}}, \boldsymbol{c}(\overline{\boldsymbol{y}'}, \overline{\boldsymbol{x}}))$

$[(-\pi \leq x \leq \pi) \wedge (k \geq 0)] \wedge [(k > 0)] \wedge [y' = \sum_{n=0}^{k-1}[(-1^n) \cdot x^{2n+1}/(2n+1)!] \Rightarrow$
$y = y' + y = \sum_{n=0}^{k-1}[(-1^n) \cdot x^{2n+1}/(2n+1)!] + (-1^k) \cdot x^{2k+1}/(2k+1)! =$
$\sum_{n=0}^{k}[(-1^n) \cdot x^{2n+1}/(2n+1)!] = y$

Por lo tanto se cumple que $y = y' + (-1^k) \cdot x^{2k+1}/(2k+1)!$

Elección de una estructura de preorden bien fundado.

Hay que encontrar un $\boldsymbol{t}: \boldsymbol{\mathcal{D}_{T1}} \rightarrow \mathbb{Z}$ $\boldsymbol{de\ tal\ forma\ que\ Q}(\bar{\boldsymbol{x}}) \Rightarrow \boldsymbol{t}(\bar{\boldsymbol{x}}) \geq \boldsymbol{0}$

$(-\pi \leq x \leq \pi) \wedge (k \geq 0) \Rightarrow k \geq 0$

Esto es trivialmente cierto con lo cual es suficiente con que $t(\bar{x}) = k$

Demostración del decrecimiento de los datos.

Para el preorden que hemos definido, hay que ver que el tamaño de los datos decrece para cada llamada recursiva. Equivale a demostrar: $\boldsymbol{Q}(\bar{\boldsymbol{x}}) \wedge \boldsymbol{B_{nt}}(\bar{\boldsymbol{x}}) \Rightarrow \boldsymbol{t}(\boldsymbol{s}(\bar{\boldsymbol{x}})) < t(\bar{\boldsymbol{x}})$

$t(\bar{x}) = k$

$t(s(\bar{x}) = k - 1$

$(-\pi \leq x \leq \pi) \wedge (k \geq 0) \wedge (k > 0) \Rightarrow k - 1 < k$ Totalmente cierto, ya que decrecen en el preorden bien fundado que hemos elegido.

2.6 Calculo del coste asintótico temporal en el caso peor de sink

Tenemos que el tamaño del problema o el número de llamadas recursivas viene dado por una variable que llamaremos *n* (k en el caso de nuestra función). Dado que el tamaño del problema va decreciendo por sustracción en cada llamada recursiva (sink (x, n-1)), aplicaremos la fórmula de reducción del problema mediante sustracción:

$$T(n)\begin{cases} cn^k \ si\ 0 \leq n < b \\ aT(n-b) + cn^k \ si\ n \geq b \end{cases}$$

Según el enunciado de la actividad 4.9. *pot (x,2n+1)* y *fact (2n+1)* son dos funciones de coste lineal O(n), de aquí deducimos pues que tmno (x,2n+1), que aunque tiene una operación de división que, es de coste constante O(1), el conjunto pertenecerá a O(n), es decir, coste lineal. En resumen, tenemos lo siguiente:

$$T\big(pot(x, 2n+1)\big) \epsilon\ \mathcal{O}(n)$$

$$T\big(fact(n)\big) \epsilon\ \mathcal{O}(n)$$

$$T\big(tmno(x, 2n+1)\big) \epsilon\ \mathcal{O}(n)$$

Visto esto, veamos los dos casos que se nos presentan:

- El caso trivial, que se dará cuando *n=0*, que es una operación de coste constante O(1)(con k=0)

  T(1) = c (la tomamos como una constante)

- El caso no trivial, que se dará cuando *n>0*, consta de una operación de coste lineal *pot(-1,n)*, una operación producto que es de coste constante(multiplicación por *tmno()*) y una ultima operación de coste lineal, la propia *tmno()*. Lo cual nos queda una operación conjunta de coste lineal O(n). Para la llamada recursiva (sink(x,n-1)), tenemos que se hace una sola llamada recursiva, con lo cual

a=1, y el tamaño del problema (n) va disminuyendo de uno en uno, con lo cual *b=1* y como tenemos un coste lineal para las operaciones no recursivas, entonces *k=1*.

Aclarado todo esto, tenemos que la ecuación de recurrencia es la siguiente:

$$T(n)\begin{cases} c & si\ n = 0 \\ 1 \cdot T(n-1) + c' \cdot n^1 & si\ n > 0 \end{cases}$$

Y como *a=1*, la solución a la recurrencia es:

$$\boldsymbol{T(n) \in\ \theta\left(n^{k+1}\right) = \theta(n^2)}$$

2.7 Optimización del diseño recursivo. Inmersión no final

2.7.1 Nuevo código del sink

$\{Q \equiv (-\pi \leq X \leq \pi) \wedge (k \geq 0)\}$

fun sink(x:real;k:natural) dev(y:real)

caso k=0 → y=x

 k>0 →y=pot(-1,n) · pot(x,2n+1)/fact(2n+1)+sink(x,k-1)

fcaso

ffun

$\{R \equiv y = \sum_{n=0}^{k}[(-1^n) \cdot x^{2n+1}/(2n+1)!]\}$

2.7.2 Inmersión de parámetros. Expresiones para los nuevos parámetros *p* y *f.*

p=pot(x, 2k+1) $\quad p' = p/x^2 \cdot (-1)$

f=fact (2k+1) $\quad f' = f/(2k+1) \cdot (2k)$

Tenemos que la primera llamada a isink llevará los siguientes parámetros:

isink (x, k, pot (x,2k+1)·(1-2(k mod 2), fact(2k+1)), donde:

pot (x,2k+1)·(1-2(k mod 2) → nos va a dar el parámetro p con el signo que ha de llevar, el cual nos lo da la operación (1-2(k mod 2).

fact(2k+1) → Nos va a dar el parámetro f, es decir el factorial de 2k+1.

Sustituimos la llamada recursiva a sink por otra llamada a isink, usando la relación establecida:

Sink(x, k) = isink (x, k, pot(x, 2·k+1) · (1-(2·(k mod2))),fact(2k-1))

Sink(x,k-1)=isink(x ,k-1, p/x² ·(-1), f/(2·k+1)·2·k)

2.7.3 Especificación de *isink:*

$\{Q_1 \equiv (-\pi \leq x \leq \pi) \wedge (k \geq 0) \wedge (p = pot(x, 2k+1) \cdot (1 - 2(k\ mod\ 2))) \wedge (f = fatc(2k+1))\}$

fun isink (x : real; k : natural; p: real; f: natural) dev (y : real)

$\{R_1 \equiv y = \sum_{n=0}^{k}[(-1^n) \cdot x^{n+1}/(2n+1)!]\}$

2.7.4 Código de *isink:*

$\{Q_1 \equiv (-\pi \leq x \leq \pi) \wedge (k \geq 0) \wedge (p = pot(x, 2k+1) \cdot \big(1 - 2(k \bmod 2)\big)) \wedge (f = fact(2k+1))\}$

fun isink (x : real; k : natural; p: real; f: natural) dev (y : real)

caso $k = 0 \rightarrow y = x$

$k > 0 \rightarrow y = {}^{p}/_{f} + isink\ (x, k-1, p/x^2 \cdot (-1), f/\ (2k+1) \cdot (2k))$

fcaso

ffun

$\{R_1 \equiv y = \sum_{n=0}^{k}[(-1^n) \cdot x^{2n+1}/(2n+1)!]\}$

2.7.5 Coste de isink

La ecuación de recurrencia que nos va a dar, va a ser similar a la de sink, con la diferencia que las operaciones no recurrentes que vamos a realizar son de coste constante. Veámoslo por casos:

- Para el caso trivial, *n=0* seguiremos teniendo una operación de coste constante *c*, O(1)(con k=0)
- Para el caso no trivial, tenemos una operación de división *P/F* de coste constante O(1), y en la llamada recursiva, nos encontramos con los parámetros de eficiencia, *P* y *F*, que realizan operaciones aritméticas de coste constante en cada llamada recursiva. Estas son:
  - P→ *P / x · x · (-1)*
  - F→ *F / (2 · n + 1) · (2 · n)*

  Esto da lugar a que *P/F* va a ser de coste constante y las operaciones aritméticas con la llamada recursiva, también lo van a ser, lo que significa que va mejorar la eficiencia del algoritmo. A continuación podemos ver como queda la ecuación de recurrencia:

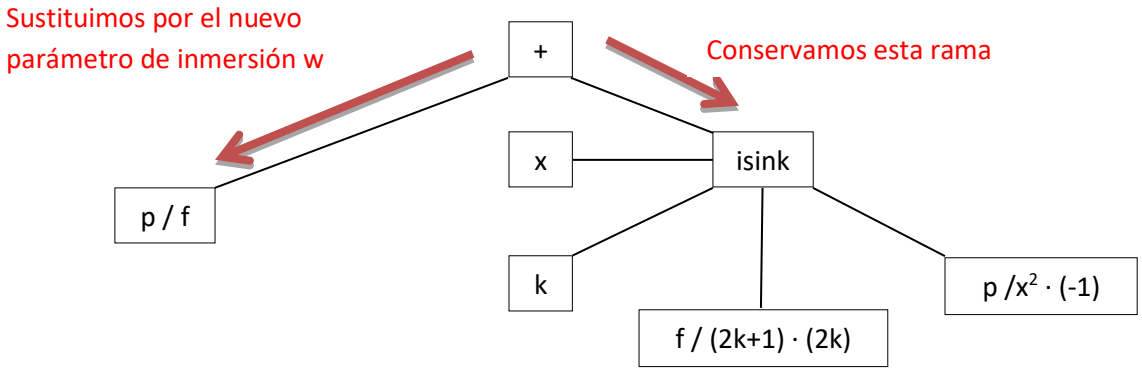
$$T(n)\begin{cases} c & si\ n = 0 \\ 1 \cdot T(n-1) + c' & si\ n > 0 \end{cases}$$

  Visto esto, la solución a la recurrencia con *a=1, b=1 y k=0* es:

$$\boldsymbol{T(n) \in\ \theta(n)}$$

2.8. Transformación a recursivo final: Desplegado – plegado

2.8.1 Árbol sintáctico de la llamada recursiva de isink



El camino hasta la función recursiva, es la rama derecha. La rama izquierda lo sustituimos por el nuevo parámetro de inmersión *w*, de tal forma que ***w = p/f.***

2.8.2 Sustituciones aplicadas

Se ha añadido un parámetro más, llamado *w*. Éste llevará precalculado el resultado en cada llamada recursiva, hasta llegar al caso trivial, donde devolverá el resultado final. La generalización que hacemos para la función es: ***iisink(x, k, p, f, w).***

Tenemos pues que:

$$\boldsymbol{iisink}\ (\boldsymbol{x},\boldsymbol{k},\boldsymbol{p},\boldsymbol{f},\boldsymbol{w}) = \boldsymbol{w} + \boldsymbol{isink}(\boldsymbol{x},\boldsymbol{k},\boldsymbol{p},\boldsymbol{f})$$

2.8.3 Transformación por desplegado y plegado de isink a iisink

Para poder aplicar el desplegado y plegado, la función ha de poseer elemento neutro y ser asociativa

Desplegado

Partiendo de la igualdad iisink (x, k, p, f, **w**) = **isink (x, k, p, f) + w**, sustituimos la función isink para su desarrollo:

$$\boldsymbol{iisink}(\boldsymbol{x},\boldsymbol{k},\boldsymbol{p},\boldsymbol{f},\boldsymbol{w}) = \left\{\begin{matrix}\boldsymbol{caso\ k = 0\ \rightarrow y = x} \\ \boxdot\ \boldsymbol{k > 0 \rightarrow \frac{p}{f} + isink\left(x, k-1, \frac{p}{x^2}\cdot(-1), f/(2k+1)\cdot 2k\right)}\end{matrix}\right\} \ \boldsymbol{+\ w}$$

Trasladamos la operación adicional a cada uno de los casos (desplegado)

$$\boldsymbol{iisink}\ (\boldsymbol{x},\boldsymbol{k},\boldsymbol{p},\boldsymbol{f},\boldsymbol{w}) = \left\{\begin{matrix}\boldsymbol{caso\ k = 0\ \rightarrow y = x + w} \\ \boxdot\ \boldsymbol{k > 0 \rightarrow w + \frac{p}{f} + isink(x, k-1, p/x^2\cdot(-1), f/(2k+1)\cdot 2k)}\end{matrix}\right.$$

Aplicamos las propiedades, elemento neutro al caso trivial y asociativa al no trivial

$$\boldsymbol{iisink}\ (\boldsymbol{x},\boldsymbol{k},\boldsymbol{p},\boldsymbol{f},\boldsymbol{w}) = \left\{\begin{matrix}\boldsymbol{caso\ k = 0\ \rightarrow y = x + w} \\ \boxdot\ \boldsymbol{k > 0 \rightarrow isink(x, k-1, p/x^2\cdot(-1), f/\ (2k+1)\cdot 2k) + (w + \frac{p}{f})}\end{matrix}\right.$$

Plegado

La expresión del caso no trivial, tiene el mismo aspecto que la función isink( )+( ). Por tanto podemos plegar para darnos la expresión:

$$\boldsymbol{iisink}\ (\boldsymbol{x},\boldsymbol{k},\boldsymbol{i},\boldsymbol{p},\boldsymbol{f},\boldsymbol{w}) = \begin{cases} \boldsymbol{caso\ k = 0\ \rightarrow y = w + x} \\ \quad \boxdot\ \boldsymbol{k} > 0 \rightarrow iisink\left(\boldsymbol{x},\boldsymbol{k-1},\boldsymbol{p/x^2}\cdot(\boldsymbol{-1}),\boldsymbol{f/(2k+1)}\cdot\boldsymbol{2k},(\boldsymbol{w}+\frac{\boldsymbol{p}}{\boldsymbol{f}})\right) \end{cases}$$

En resumen, nos queda**:**

$$\boldsymbol{isink}(\boldsymbol{x},\boldsymbol{k-1},\boldsymbol{p/x^2}\cdot(\boldsymbol{-1}),\boldsymbol{f/(2k+1)}\cdot\boldsymbol{2k}) + \left(\boldsymbol{w}+\frac{\boldsymbol{p}}{\boldsymbol{f}}\right) = \boldsymbol{iisink}\left(\boldsymbol{x},\boldsymbol{k-1},\boldsymbol{p/x^2}\cdot(\boldsymbol{-1}),\boldsymbol{f/(2k+1)}\cdot\boldsymbol{2k},(\boldsymbol{w}+\frac{\boldsymbol{p}}{\boldsymbol{f}})\right)$$

Y la llamada inicial para la función iisink sería:

iisink (x, k, pot(x, 2·k+1) · (1-(2·(k mod 2))),fact(2k-1),w=0)

Dónde w =0 desde la primera llamada.

2.8.4 Código de iisink y especificación

$\{Q_2 \equiv (-\pi \leq x \leq \pi) \wedge (k \geq 0) \wedge (p = pot(x, 2k+1) \cdot (1 - 2(k\ mod\ 2))) \wedge (f = fatc(2k+1)) \wedge w = \sum_{n=0}^{k}(-1)^n \cdot x^{2n+1}/(2n+1)!\}$

fun iisink(x: real, k: nat, p: real, f: nat, w: real) dev (y: real)

caso k =0 → y = w + x

◊ k >0 → y = iisink(x, k-1, p / x^2 · (-1), f / (2k+1) · 2k, w + f/p)

fcaso

ffun

$\{R_2 \equiv y = \sum_{n=0}^{k}\left[(-1^n) \cdot \frac{x^{2n+1}}{(2n+1)!}\right] = w\}$

2.9. Diseño iterativo de sinkIt

2.9.1 Invariante y protección del bucle

Partiendo de la postcondición de la función sink:

$\{R \equiv y = \sum_{n=0}^{k}[(-1^n) \cdot x^{2n+1}/(2n+1)!]\}$

Vamos a desarrollar el invariante y la protección para el bucle.

Según el texto base, podemos crear el invariante de dos formas:

- Eliminando una conjunción en la postcondición
- Agregando una variable que debilite la postcondición.

En nuestro caso, lo mejor es la segunda opción, agregar una variable en la postcondición; ésta se va a llamar i.

De esta forma, la postcondición que nos va a quedar para la función sinkIt es la siguiente:

$\{R_{It} \equiv y = \sum_{n=0}^{i}[(-1^n) \cdot x^{2n+1}/(2n+1)!] \wedge (i = k)\}$

Donde:

$\Rightarrow \{RIt_1 \equiv \sum_{n=0}^{k}[(-1^n) \cdot x^{2n+1}/(2n+1)!]\}$

$\Rightarrow \{RIt_2 \equiv i = k\}$

$\Rrightarrow R_{It} \equiv RIt_1 \wedge RIt_2$

A partir de este desarrollo, el invariante va a ser:

$\{P \equiv y = \sum_{n=0}^{i}[(-1^i) \cdot x^{2i+1}/(2i+1)!] \wedge (0 \leq i \leq k)\}$

Entonces la condición para que el bucle itere es que i ≥ 0. Es decir, siendo B= condición, debe cumplirse que:

B Ξ i ≤ k, para que el bucle se ejecute.

En el momento que B sea falso, es decir B Ξ i = k+1 o lo que es lo mismo ¬B, el bucle no hará otra iteración y terminará su ejecución. Es lo que llamamos la protección del bucle, que no es más que la condición de finalización de éste.

Deducimos pues que: $\boldsymbol{P \wedge \neg B \Rightarrow R_{It}}$

2.9.2 Inicialización del bucle

Los valores para la inicialización del bucle serán los siguientes:

y:= 0 irá acumulando el resultado de la operación en cada iteración.

i:= 0 irá contando cada iteración y haciendo que se cumpla el invariante.

2.9.3 Instrucción avanzar y cota

La instrucción i:= i+1, es la que va a hacer avanzar el bucle hacia su terminación, es decir, hará B falso cuando i=k+1.

La cota la podemos establecer como t=k-i, de tal forma que cada vez que i crece, t decrece.

2.9.4 Instrucción para restablecer la invarianza a cada vuelta del bucle

La instrucción avanzar, hará que el invariante calculado, pierda su propiedad de invarianza, entonces será necesario incluir otra instrucción para restablecer esta perdida de propiedades. La instrucción es:

y:= y+ pot(-1,i)·tmno(x,2·i+1)

2.9.5 Código de *sinkIt*

$\{Q_{It} \equiv (-\pi \leq x \leq \pi) \wedge (k \geq 0)\}$

fun sinkIt(x:real;k:nat) dev(y:real)

var i:natural fvar

< y, i > := <0, 0>

$\{\boldsymbol{P} \equiv \boldsymbol{y} = \sum_{\boldsymbol{n=0}}^{\boldsymbol{i}}[(-\mathbf{1}^{\boldsymbol{i}}) \cdot \boldsymbol{x}^{\mathbf{2}\boldsymbol{i}+\mathbf{1}}/(\mathbf{2}\boldsymbol{i}+\mathbf{1})!] \wedge (\mathbf{0} \leq \boldsymbol{i} \leq \boldsymbol{k})\}$

mientras i ≤ k hacer

y:= y+ pot(-1,i)·tmno(x,2·i+1);

i:=i+1;

fmientras

$\{R_{It} \equiv y = \sum_{n=0}^{i}[(-1^{n}) \cdot x^{2n+1}/(2n+1)!] \wedge (i = k)\}$

2.9.6 Calculo del coste para *sinkIt*

Las dos instrucciones de asignación antes del bucle, tendrán un coste constante. La instrucción booleana i<=k tendrá un coste constante O(1). La operación restablecer tiene los siguientes costes :

Pot()=O(n) tiene coste lineal

Tmno()=O(n) también tiene coste lineal

La operación avanzar i:=i+1 será una operación de coste constante O(1).

La operación completa la podemos expresar de la siguiente forma:

$\mathcal{O}(n) \cdot \mathcal{O}(n) = \mathcal{O}(n^2)$ Por las dos operaciones de coste lineal y aplicando la regla de la suma para todas las operaciones del algoritmo iterativo nos queda:

$\max(\mathcal{O}(1) + \mathcal{O}(n^2)) \rightarrow \boldsymbol{T}(\boldsymbol{n}) = \boldsymbol{\theta}(\boldsymbol{n}^{\mathbf{2}})$ **y siempre dependiendo del valor de k.**

El número de iteraciones será $f_{iter}(k-i)$. Dependiendo de k.

3. IMPLEMENTACIÓN Y ANÁLISIS EMPÍRICO

3.1. Implementación en Modula-2

3.1.1. Llamadas a las funciones

Llamada inicial a la función sink:
sink(X,K);

Llamada inicial a la función isink:
*isink(X,K,pot(X,(2*K)+1)*(1.0-(2.0*VAL(REAL,K MOD 2))),fact((2*K)+1));*

Llamada inicial a la función iisink:
*iisink(X,K,pot(X,(2*K)+1)*(1.0-(2.0*VAL(REAL,K MOD 2))),fact((2*K)+1),0.0);*

Llamada inicial a la función sinkIt:
sinkit(X,K);

3.1.2. Código fuente de las funciones

(* Implementacion de la funcion sink *)

PROCEDURE sink(X:REAL;K:NATURAL):REAL;

```
BEGIN
  IF K=0 THEN    (*Caso trivial*)
    RETURN(X);
  ELSE        (*Caso No Trivial*)
    RETURN(pot(-1.0,K)*tmno(X,2*K+1)+sink(X,K-1)); (*Llamada recursiva*)
  END;
END sink;
```

```
(* Implementación de la función isink *)
(* El parámetro F es REAL (y no NATURAL como en el desarrollo teórico) para
  solventar el problema del rango de los enteros en Modula-2 *)
PROCEDURE isink(X:REAL;K:NATURAL;P,F:REAL):REAL;
BEGIN
  IF K=0 THEN    (*Caso trivial*)
    RETURN(X);
  ELSE            (*Caso No trivial*)
    RETURN(P/F+isink(X,K-1,(P/(X*X)*(-1.0)),(F/VAL(REAL,((2.0*VAL(REAL,K))+1.0)*(2.0*VAL(REAL,K))))));
    (*llamada recursiva con inmersión por eficiencia con los parámetros P y F. Recursividad NO-FINAL*)
  END;
END isink;
```

```
(* Implementacion de la funcion iisink *)
(* El parámetro F es REAL (y no NATURAL como en el desarrollo teorico) para
  solventar el problema de la falta de precisión de los enteros en Modula-2 *)
PROCEDURE iisink(X:REAL;K:NATURAL;P,F,W:REAL):REAL;
(*W es la variable que va a almacenar los resultados acumulados en cada llamada*)
BEGIN
  IF K=0 THEN    (*Caso Trivial*)
    RETURN(S+X);
```

```
   ELSE        (*Caso no trivial*)
      RETURN(iisink(X,K-1,(P/(X*X))*(-1.0),F /VAL(REAL,((2.0*VAL(REAL,K))+1.0)*(2.0*VAL(REAL,K))),W+(P/F)));
       (*llamada recursiva con inmersión por eficiencia con los parámetros P y F. Recursividad FINAL*)
   END;
END iisink;
```

```
(* Implementacion de la funcion sinkit *)
(*Declaramos las variables solución e i. *)
(* "solución" irá sumando los resultados de las sucesivas iteraciones hasta*)
(*dar el resultado final. i será el contador de bucle*)
PROCEDURE sinkit(X:REAL;K:NATURAL):REAL;
VAR solucion:REAL;i:NATURAL;
BEGIN
   (*Inicializamos a cero las dos variables locales*)
   i:=0; (*contador de iteraciones*)
   solucion:=0.0; (*Almacenamiento de la solución*)
   WHILE i<=K DO (*Protección del bucle*)
      solucion:=solucion+pot(-1.0,i)*tmno(X,2*i+1); (*Instrucción restablecer*)
      i:=i+1; (*Avanzar*)
   END;
   RETURN solucion; (*Devolvemos solución*)
END sinkit;
```

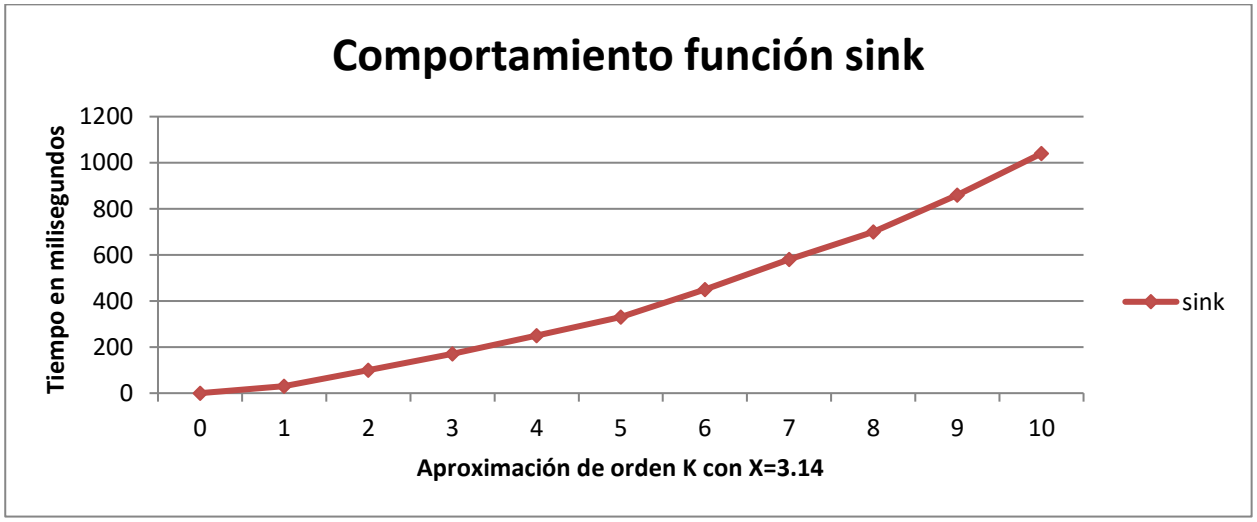
3.2. Análisis empírico

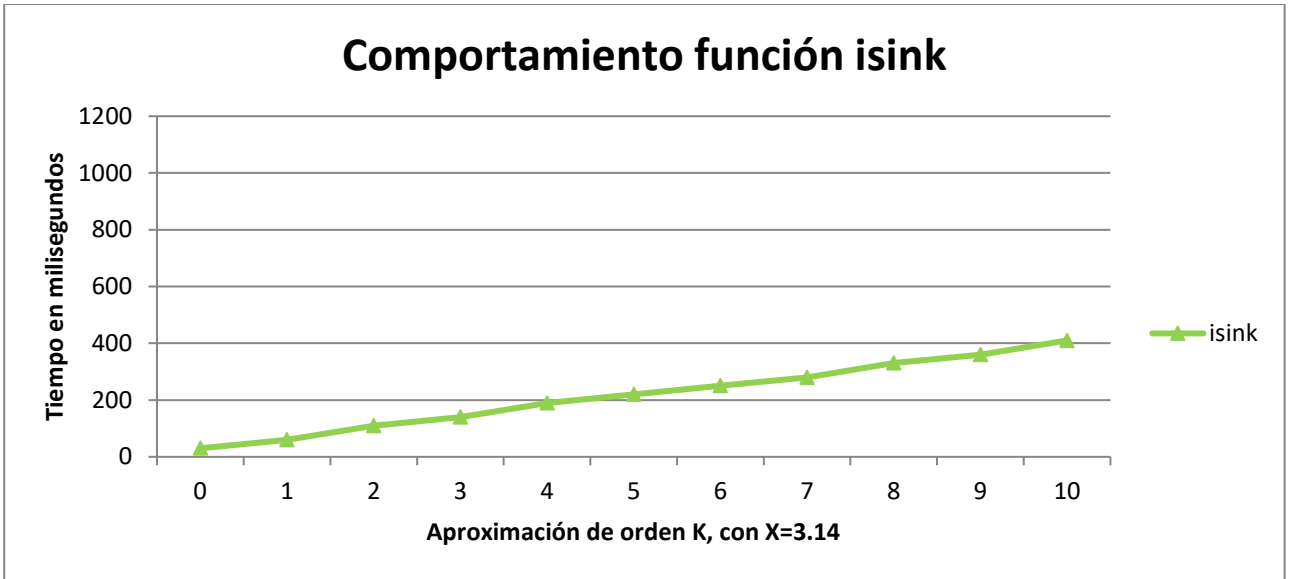
A continuación podemos ver una tabla con valores obtenidos de cada una de las funciones, ejecutando desde K=0 hasta K=10, con X=3.14:

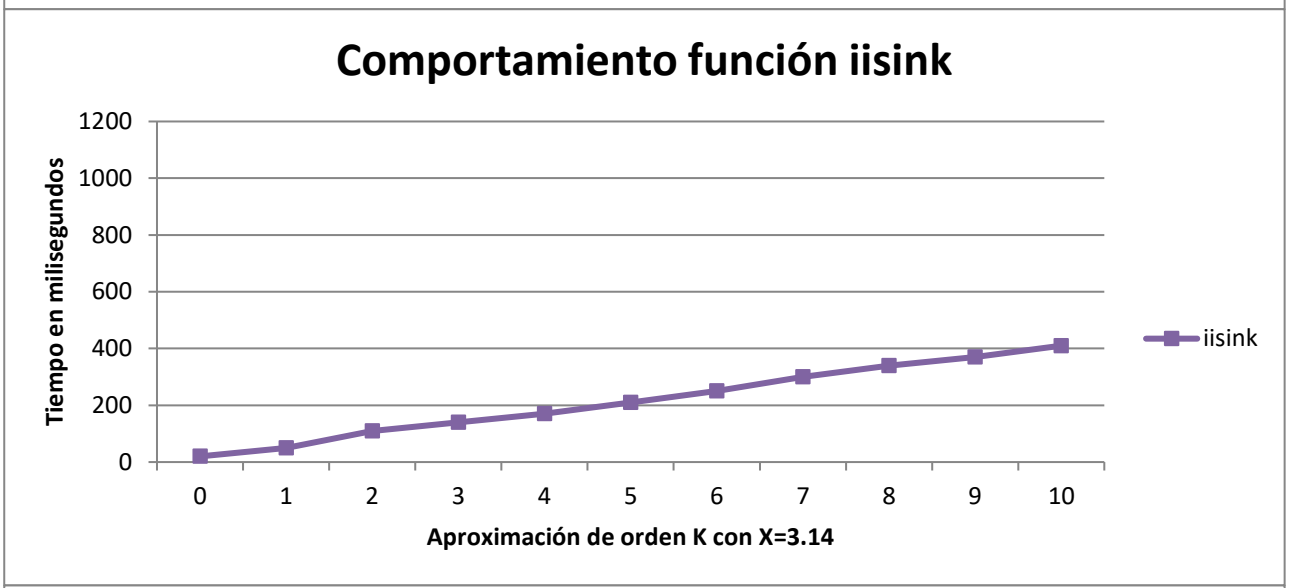
Para x=3.14	tiempos(milisegundos)			
Valor de K	**sink**	**isink**	**iisink**	**sinkIt**
0	0	30	20	30
1	30	60	50	60
2	100	110	110	90
3	170	140	140	160
4	250	190	170	250
5	330	220	210	350
6	450	250	250	450
7	580	280	300	580
8	700	330	340	710
9	860	360	370	850
10	1040	410	410	1010

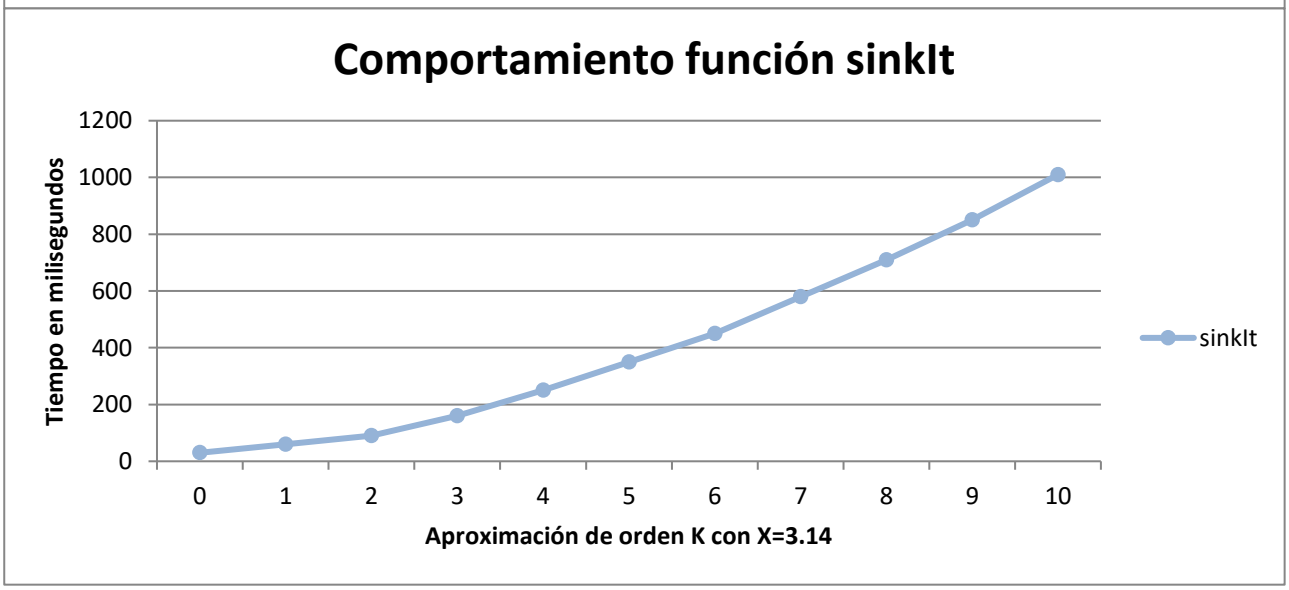
Graficado:

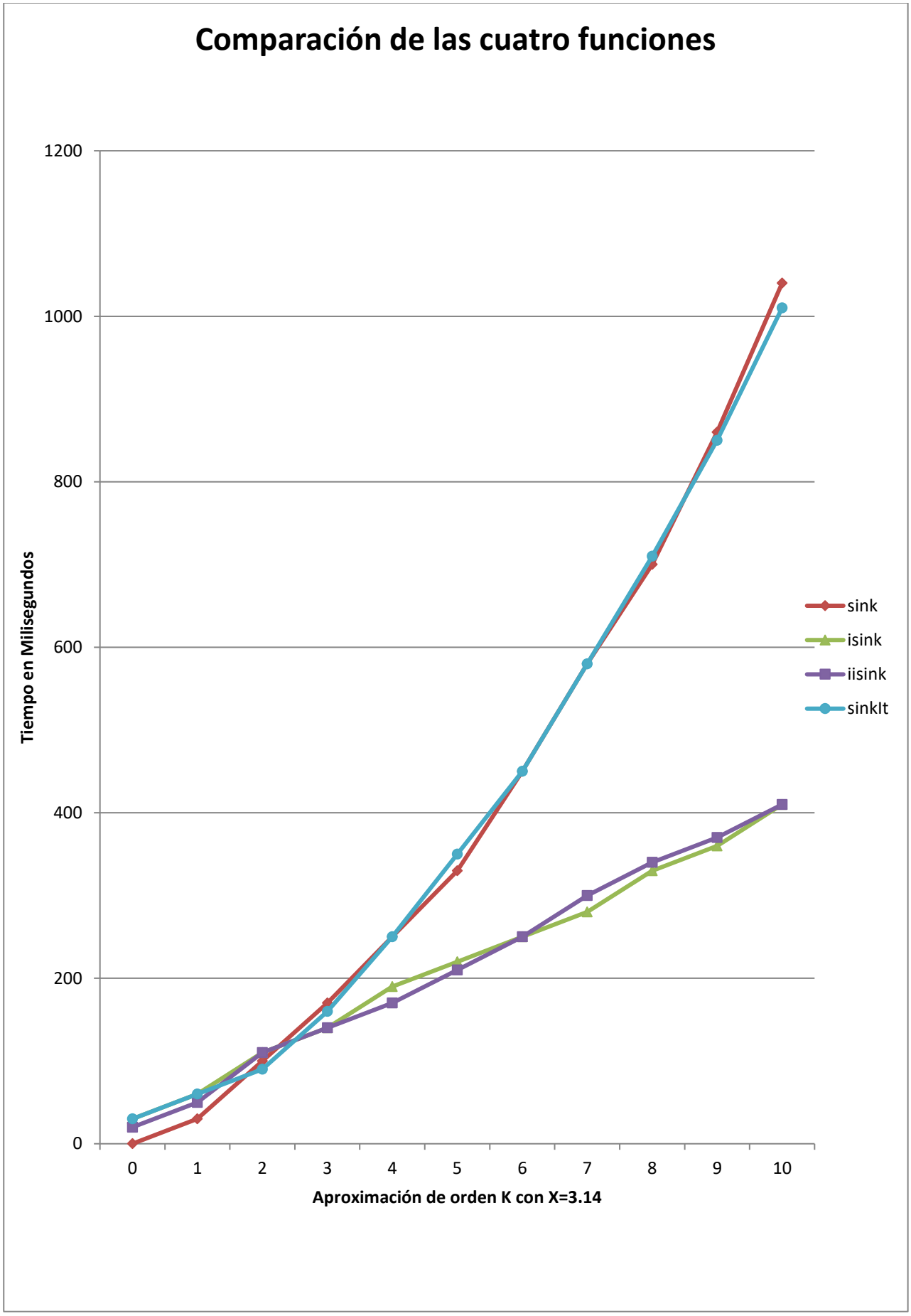
En las siguientes gráficas basadas en la tabla anterior, podemos ver claramente como las funciones isink e iisink son más eficientes que sink y sinkIt:




Comportamiento función isink
Tiempo en milisegundos
1200
1000
800
600
400
200
0
0 1 2 3 4 5 6 7 8 9 10
Aproximación de orden K, con X=3.14
isink


Comportamiento función iisink
Tiempo en milisegundos
1200
1000
800
600
400
200
0
0 1 2 3 4 5 6 7 8 9 10
Aproximación de orden K con X=3.14
iisink

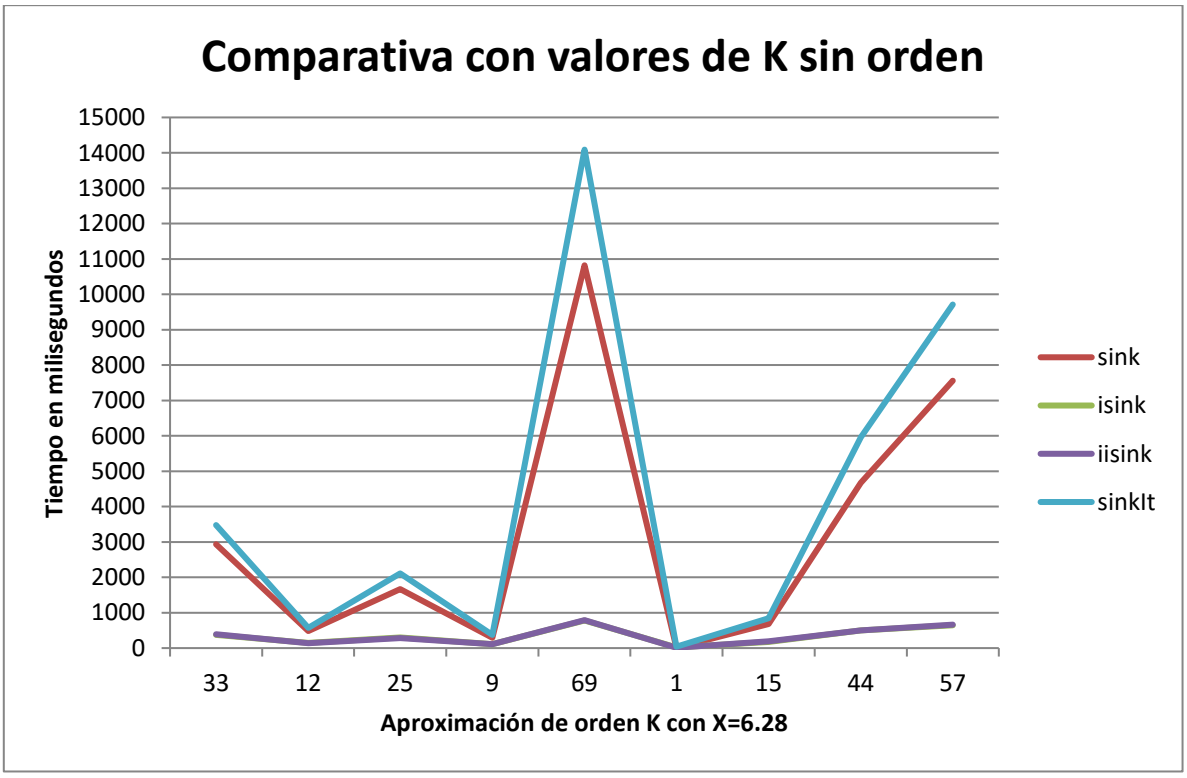

Comportamiento función sinkIt
Tiempo en milisegundos
1200
1000
800
600
400
200
0
0 1 2 3 4 5 6 7 8 9 10
Aproximación de orden K con X=3.14
sinkIt


Comparación de las cuatro funciones
1200
1000
800
600
400
200
0
Tiempo en Milisegundos
0
1
2
3
4
5
6
7
8
9
10
Aproximación de orden K con X=3.14
sink
isink
iisink
sinkIt

En la siguiente tabla, se pueden apreciar los datos devueltos por las funciones con datos de entrada donde no son consecutivos y X=6.28:

Para x=6.28	tiempos(milisegundos)			
Valor de K	**sink**	**isink**	**iisink**	**sinkIt**
33	2930	370	390	3480
12	480	150	140	570
25	1670	300	280	2110
9	290	110	110	370
69	10820	780	790	14090
1	10	20	10	40
15	680	170	190	840
44	4670	500	500	5960
57	7560	640	660	9710

El graficado de estos datos, donde se puede apreciar de nuevo cuales son las funciones más eficientes, dependiendo de los valores:



Se puede apreciar que isink e iisink casi van de la mano en cuestión de eficiencia. Sink y sinkIt se disparan conforme los datos (K en este caso) son mayores.