eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Grado en Ingeniería Informática

Computación

Trabajo de Fin de Grado

Mejoras en la implementación de TaylorSeries para resolver EDO utilizando el método de Taylor en el entorno Julia

Autor/a

Iñigo Olaso Rodrigo

2020

eman ta zabal zazu

Universidad del País Vasco
Euskal Herriko Unibertsitatea

INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Grado en Ingeniería Informática

Computación

Trabajo de Fin de Grado

Mejoras en la implementación de TaylorSeries para resolver EDO utilizando el método de Taylor en el entorno Julia

---

Autor/a

Iñigo Olaso Rodrigo

Directore/a(s)

Joseba Makazaga

Resumen

En este trabajo se ha estudiado la posibilidad de lograr una mejora de los tiempos de ejecución en las operaciones básicas entre series de Taylor. En JULIA existe el paquete *taylorSeries* que permite operar con las series de Taylor pero creemos que los tiempos de ejecución y la gestión de memoria que realiza el paquete son mejorables. Para lograr el objetivo, se han estudiado distintas estructuras que permiten realizar las mismas operaciones. Para ello, se han codificado las funciones necesarias para las distintas estructuras de datos y se han comparado los tiempos de ejecución del nuevo código con los tiempos que obtiene el paquete estándar de JULIA.

Índice general

Resumen I

Índice general III

1. Introducción 1

2. Objetivos del proyecto 3

3. Desarrollo del proyecto 5

3.1. Series de Taylor 5

3.2. Julia 7

3.2.1. Operadores aritméticos 8

3.2.2. Valores y tipos 9

3.2.3. Variables, expresiones y sentencias 10

3.2.4. Expresiones y sentencias 10

3.2.5. Variables globales y locales 11

3.2.6. Arrays 13

3.2.7. Objetos y Valores 14

3.2.8. Estructuras y objetos 17

3.3. TaylorSeries.jl 18

3.3.1. Igualdad 21

3.3.2. Suma y Resta 22
3.3.3. Multiplicación 29

4. Soluciones investigadas **33**
4.1. Problema propuesto 33
4.2. Soluciones propuestas 35
4.3. Series de Taylor para la resolución de ecuaciones diferenciales 57

5. Conclusiones **67**
5.1. Trabajo futuro 68

Bibliografía **69**

1. CAPÍTULO

Introducción

Este trabajo de fin de grado pretende mejorar los tiempos en las operaciones básicas entre series de Taylor del paquete taylorseries de JULIA. Se trata de un objetivo ambicioso, ya que pretendemos mejorar un paquete bien establecido y que se usa extensamente.

En JULIA existe un paquete accesible a todo el mundo (*taylorseries.jl*) en el que encontramos el código que realiza todo tipo de operaciones entre series (suma, resta, multiplicación,...). Sin embargo, creemos que este código es mejorable ya que realiza un gran número de reservas de memoria que ralentizan las operaciones entre las series.

Por esa razón, lo primero de todo es aprender el lenguaje JULIA: conocer su funcionamiento, el uso que hace de las estructuras y de la memoria. Además, hay que estudiar en profundidad cómo funcionan las series de Taylor y sobre todo, de qué manera se utilizan en JULIA. Para ello, hay que analizar el paquete *taylorseries.jl* en el que están programadas todas las funciones que realizan las operaciones entre las series. Lo más importante será entender bien el código que JULIA utiliza ya que es algo totalmente nuevo.

En la memoria hay una pequeña introducción de las series de Taylor, después da comienzo el capítulo que explica el desarrollo del proyecto, en él hay una sección donde se presenta el lenguaje JULIA y en la que se habla de tipos de datos, estructuras, operaciones entre variables, cómo se definen las funciones, constructores de tipos de datos, etcétera. A continuación, hay un análisis del paquete *taylorSeries.jl* ya que el objetivo es tratar de

mejorar los tiempos de ejecución de este paquete.

El siguiente capítulo plantea el problema en base al código del paquete y muestra un conjunto de posibles soluciones que se han investigado, mostrando los pros y los contras de cada solución. En la última sección de este capitulo se prueba la mejor solución encontrada con un problema real de uso de las series de Taylor, la resolución de ecuaciones diferenciales ordinarias mediante el método de Taylor.

Por último, se dan las conclusiones y las lineas futuras en las que se podría continuar trabajando.

2. CAPÍTULO

Objetivos del proyecto

Este trabajo tiene por un lado unos objetivos generales y por otro lado, unos específicos. Como objetivos generales mencionaría el aprendizaje del lenguaje JULIA (sintaxis, ventajas, desventajas, uso de la memoria,...) y la comprensión de las series de Taylor (para qué se utilizan y cómo se programan en JULIA). Una vez estudiados y conseguidos estos objetivos, abordaríamos los objetivos específicos en los que planteamos una solución eficiente para nuestro problema.

Concretamente, el objetivo de la solución es mejorar los tiempos de ejecución en las operaciones entre estas series. ¿Por qué? Julia realiza un gran número de copias de variables y esto hace que se ocupe demasiado espacio en memoria provocando que las operaciones entre series sean lentas.

Como ya he mencionado previamente, era importante conocer las estructuras que JULIA empleaba para almacenar las series y sobre todo, saber cuanto le costaba acceder a esta estructura ya que ahí podía encontrar la solución a mi problema.

El tema del coste a primera vista parece irrelevante, pero ocupa una posición más que importante en este trabajo. La razón es obvia, dominar el tema de la memoria; es decir, saber que estructura ocupa menos espacio, a cual es más fácil acceder, nos llevará a proponer soluciones que probablemente consigan una mejora con respecto a la original.

3. CAPÍTULO

Desarrollo del proyecto

3.1. Series de Taylor

Es evidente que las protagonistas de este trabajo son las series de Taylor.

Una serie de Taylor es una serie funcional y surge de una ecuación en la que se puede encontrar una solución aproximada para cualquier función. Esta solución se devuelve en forma de polinomio, lo que resulta muy interesante ya que los polinomios son fáciles de manipular y operar. Esta serie sirve para aproximar el valor de una función en un punto concreto en base al valor de la función y sus derivadas en otro punto.

Para hacer esta aproximación, no se pueden utilizar todas las expresiones del polinomio, sino únicamente aquellas que aproximen mejor el valor que buscamos. Es por eso que aparece lo que conocemos como error residual; es decir, el resto de términos que no empleamos para aproximarla.

Definición:

$$f(x) = f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 + ... \quad (3.1)$$

Aunque... de forma más compacta, podemos expresarla de la siguiente forma:

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

Aquí, n! es el factorial n y $f^n(a)$ hace referencia a la n-ésima derivada de f en el punto a. La derivada 0 de f se define como la propia f y por último, tanto $(x-a)^0$ como el factorial de 0 (0!) son iguales a 1.

Si esta serie converge para todo x perteneciente al intervalo (a-r, a+r) y la suma es igual a f(x), entonces la función f(x) se dice que es analítica. Para comprobar si la serie converge a f(x), se suele utilizar una estimación del resto del teorema de Taylor. Una función es analítica si y solo si se puede representar con una serie de potencias; los coeficientes de esa serie son necesariamente los determinados en la fórmula de la serie de Taylor. Si a = 0, a la serie se le llama serie de Maclaurin. Esta representación tiene 3 principales ventajas:

1. La derivación e integración de una de estas series se puede realizar término a término ya que resultan en operaciones triviales.

2. Se utilizan para calcular valores aproximados de la función.

3. Es posible demostrar que, si es viable la transformación de una función a una serie de Taylor, es la mejor aproximación posible.

3.2. Julia

Julia [Lauwens and Downey, 2019] [Jeff Bezanson, 2017] es un lenguaje muy matemático pero a la vez muy intuitivo. Sus características principales son las siguientes:

Julia es un lenguaje de programación multiplataforma de tipo dinámico, alto nivel y desempeño para la computación genérica, técnica y científica, con una sintaxis que es familiar para los usuarios de otros entornos de computación técnica y científica, ya que es similar a otro tipo de lenguajes. Dispone de Jit (un compilador muy avanzado).

La librería estándar, escrita casi completamente en Julia, también integra las mejores librerías de C y Fortran para el álgebra lineal, generación de números aleatorios y procesamiento de señales y cadenas. Además, la comunidad de desarrolladores de Julia contribuye un número de paquetes externos a través del gestor de paquetes integrado de Julia. IJulia, una colaboración entre las comunidades de IPython y Julia, provee de una poderosa interfaz gráfica basada en el navegador para Julia.

En Julia los programas están organizados en torno al despacho múltiple; definiendo funciones y sobrecargándolas para diferentes combinaciones de tipos de argumentos, los cuales también pueden ser definidos por el usuario.

En resumidas cuentas, agrupando y resumiendo lo hablado anteriormente, sus características principales son estas:

1. **Despacho múltiple**: permite definir el comportamiento de las funciones a través de múltiples combinaciones de tipos de argumentos (métodos).

2. **Sistema de tipo dinámico**: tipos para la documentación, la optimización y el despacho.

3. **Buen desempeño**, comparado al de lenguajes estáticamente compilados como C.

4. Gestor de paquetes integrado.

5. Macros tipo Lisp y otras comodidades para la meta programación.

6. Permite la llamada a funciones de otros lenguajes, añadiendo paquetes como: Python (PyCall), C, Fortran, Java (JavaCall) y Matlab (MATLAB).

7. Licencia MIT: libre y de código abierto.

3.2.1. Operadores aritméticos

Los operadores +, -, y * realizan sumas, restas y multiplicaciones respectivamente, al igual que en los sencillos ejemplos que pongo a continuación:

```
julia> 20 + 2
22
julia> 56 - 1
55
julia> 5 * 9
45
```

El operador / realiza la división. Sin embargo, en JULIA, hay que tener en cuenta que los resultados de todas las divisiones son números de tipo coma flotante. Veamos un ejemplo:

```
julia> 24/2
12.0
julia > typeof(ans)
Float64
```

Ahora nos preguntamos cual es la razón por la qué el resultado es un número en coma flotante en vez de un entero (ya que es una operación exacta). Esto será explicado más adelante.

Por último, el operador realiza potencias; es decir, eleva un número a una potencia dada:

```
julia> 6^2 + 6
42
```

3.2.2. Valores y tipos

Un valor es uno de los elementos básicos con los que trabaja un programa, así como una letra o un número. Algunos de los valores vistos hasta el momento son 12.0, 42. Pese a no mencionarlos, las cadenas de caracteres también constituyen valores: "Hola!".

Estos valores pertenecen a diferentes tipos: 42 es un entero (integer en inglés), 12.0 es un número en coma flotante (floating-point number), y ”Hola, Mundo!” es una cadena (string). Además, tal y como he hecho en uno de los comandos utilizados arriba, en caso de tener dudas del tipo de valor con el que estamos trabajando, basta con utilizar la instrucción:

```
julia> typeof("valor")
julia> typeof(42)
Int64
julia> typeof(12.0)
Float64
julia> typeof("Hola!")
String
```

El resultado de esta instrucción te dice cual es el tipo del valor que le pasas como parámetro. JULIA dispone de enteros, floats, cadenas,...etc. Sin embargo, también da la opción de crear estructuras o tipos propios.

En este caso, los resultados devueltos son Int64 y Float64. Ese 64 indica el número de bits que puede utilizar nuestra computadora y en consecuencia, el rango de valores que admite el programa.

Por otro lado, hay que tener cuidado cuando los valores van entrecomillados. No se deben confundir los enteros o números en coma flotante con los String. Todo valor que vaya entre comillas es de tipo String aunque su contenido sea un entero: ”12.0” o ”64” son de tipo String. También es cierto que existen funciones que permiten convertirlas a su tipo original.

Por último y no menos importante, cabe la posibilidad de usar las comas para separar números grandes. Por poner un ejemplo:

```
julia> 5,500,000
(5, 500, 0)

julia> typeof(ans)
Tuple{Int64,Int64,Int64}
```

Por esta razón, nos tenemos que olvidar del uso de comas o puntos para definir números de grandes dimensiones ya que el resultado puede que no sea el esperado.

3.2.3. Variables, expresiones y sentencias

Todo lenguaje de programación ofrece una gran característica: la capacidad de poder manipular variables. Como ya se sabe, una variable es un nombre que hace referencia a un valor. Por ejemplo, la asignación es la operación que realiza este proceso. Cuando se ejecutan instrucciones tipo:

```
julia> msg = "Vamos Juan, vístete, que nos tenemos que ir"

julia> s=5
5
```

Estos dos sencillos ejemplos hacen asignaciones. En el primer caso, se le asigna a la cadena de caracteres a la variable msg mientras que en el segundo ejemplo, se le asigna a la variable s el valor 5 (que en este caso es un entero).

Puede que utilizando Diagramas de Estado, la comprensión sea mas sencilla porque se expresa de una forma más visual. A continuación, se muestra un diagrama de estado con estos dos ejemplos.

```
msg ------> "Vamos Juan, vístete, que nos tenemos que ir"

s------> 5
```

3.2.4. Expresiones y sentencias

Una expresión es una combinación de valores, variables y operadores. Un valor o una variable por sí solos se consideran una expresión, por lo que las siguientes expresiones

son válidas:

```
julia> 24
24
julia> s
5
julia> s + 5
10
```

Cuando se escribe una expresión en la consola, esta se evalúa, encontrando así el valor de la expresión. En este ejemplo, la variable s tiene el valor 5 que le habíamos asignado previamente. Después, se escribe la sentencia n+5 que retorna el valor 10. Una sentencia es una unidad de código que tiene un resultado, entre ellos: crear una variable o mostrar un valor.

```
julia> s = 10
10

julia> println(s)
10
```

Este ejemplo muestra dos sencillos comandos. El primero, realiza una sentencia de asignación en la cual a la variable s se le asigna el valor 10. La segunda es una sentencia de impresión que hace uso de la variable creada previamente y muestra su valor por pantalla. Cuando escribe una sentencia, la consola la ejecuta. En otras palabras, hace lo que dice la sentencia.

3.2.5. Variables globales y locales

Es importante conocer bien la diferencia entre una variable global y una local. Las variables locales son aquellas que se crean dentro de un programa y que se destruyen una vez finaliza su ejecución. Se va a poner como ejemplo una función que suma dos variables que le pasamos como parámetros y cuyo resultado se almacena en una variable local.

```
julia > function sumarDos(x::Int,y::Int)
    z=x+y
end

julia > sumarDos(3,4)
7

julia> z
ERROR: UndefVarError: z not defined



```

La operación guarda en z (local) el resultado de sumar x e y. Sin embargo, tras haber hecho la llamada, al pedirle a JULIA el contenido de la variable z, responde con un mensaje de error que dice que es una variable que no está definida. Como se ha dicho antes, las variables locales se destruyen según finaliza la ejecución del programa y por esa razón nos salta ese mensaje de error.

Se podría solucionar este problema utilizando variables globales. Estas son las que se crean fuera de la función dando así la opción de usarlas con gran libertad en todas las funciones.

Usando la misma simple función que antes, habría que hacerlo así:

```
julia > function sumarDos(x::Int,y::Int)
    global z=x+y
end

julia > sumarDos(3,4)
7

julia> z
7
```

La diferencia es evidente. La variable z se declara como global dentro de la función. Después, se realiza la llamada que almacenará en z el resultado de sumar 3 y 4. A partir de ahora también podremos utilizar esta variable en otros programas. Esta es la única manera de que la variable z también cambie de forma global. Es importante saberlo ya que es fácil caer en el error de hacer lo siguiente.

```
julia > global z=1

julia > function sumarDos(x::Int,y::Int)
    z=x+y
end

julia > sumarDos(3,3)
6

julia > z
1
```

Aunque z (cuyo valor es 1) se declara de forma global, la variable solo cambia de forma interna ya que, cuando finaliza el programa y se pregunta por su contenido, el valor devuelto es el inicial (previo a la ejecución del programa).

3.2.6. Arrays

Al igual que las cadenas de caracteres, los arreglos son secuencias de valores que pueden ser de cualquier tipo. En JULIA, hay varias formas de crear arreglos:

```
julia > a= ["Hola", "Adios", "Hasta pronto"]
3-element Array{String,1}:
 "Hola"
 "Adios"
 "Hasta pronto"

julia > b= [1, 2, 3]
 3-element Array{Int64,1}:
 1
 2
 3

julia > c=[]
 0-element Array{Any,1}

julia> typeof(a)
Array{String,1}

julia> typeof(b)
Array{Int64,1}

julia> typeof(c)
Array{Any,1}
```

Tras crear los arreglos: el primero de cadenas de caracteres, el segundo de enteros y el último vacío, la instrucción *typeof()* índica que tipo de valores contiene nuestro arreglo. A diferencia de a y b, el array c podrá contener cualquier tipo de valor en su interior. Es importante saber que el número que acompaña al contenido que va entre llaves hace referencia a las dimensiones del arreglo.

Al igual que en otros lenguajes, para acceder a alguno de los elementos del arreglo se utiliza el operador corchete y como siempre, el número introducido entre dichos corchetes especifica el índice al que accedemos. Por ejemplo, si se quiere acceder al primer elemento de a...:

julia > a[1]

"Hola"

Una de las características más importante de los arreglos es que son mutables; es decir, tras crearlos, es posible acceder a cualquiera de sus elementos y cambiar su valor. Hay que tener cuidado con los arrays, ya que el comportamiento de las variables de tipo array es como en lenguaje C: una variable de tipo array hay que entenderlo como un puntero a la memoria donde se guardan los valores del array, y pueden haber diferentes referencias o aliases del mismo array (vease el operador (===) en Objetos y Valores).

3.2.7. Objetos y Valores

Vamos a ejecutar las siguientes sentencias:

```
julia> a="Hola"
"Hola"

julia> b="Hola"
"Hola"

julia> a===b
true
```

Estas sentencias de asignación plantean una gran duda. ¿Cada una apunta a un objeto distinto o apuntan ambas al mismo objeto? Para resolverla, el operador (===) aclara si ambas variables se refieren o no al mismo objeto.

El resultado del comando es *true* y por lo tanto, tanto a como b apuntan al mismo objeto. Es importante entenderlo ya que no siempre ocurre de esta manera:

```
julia> a=[1, 3, 5]
3-element Array{Int64,1}:
 1
 3
 5

julia> b=[1, 3, 5]
3-element Array{Int64,1}:
 1
 3
 5

julia> a===b
false
```

Como en el ejemplo anterior, se ejecutan dos sentencias de asignación, de arreglos en este caso; es decir, los objetos a los que a y b apuntan son de tipo *Array*. Sin embargo, estos no son idénticos. En otras palabras, a y b no apuntan al mismo objeto (la tercera sentencia nos da *false*).

Los términos que se utilizan en estos casos son los siguientes: equivalentes e idénticos. Dos variables son equivalentes cuando tienen el mismo o los mismos elementos y son idénticos cuando son el mismo objeto. En resumidas cuentas, si dos arrays son idénticos, también serán equivalentes mientras que si son equivalentes no tienen por qué ser idénticos.

Si dos variables no son idénticas podrán serlo utilizando el operador asignación (**b=a**). De esta forma, se está creando una nueva referencia a dicho objeto. Lo que hay que saber es que si el objeto con alias es mutable, los cambios hechos a un alias afectan al otro y si no es mutable, por mucho que apunten a lo mismo, no sufrirán cambios. Por esta razón, es recomendable que los objetos con más de una referencia sean inmutabes. Pongamos un ejemplo con una variable de tipo entero.

```
julia> x=5
5

julia> y=x
5

julia> x===y
true

julia> y=6
6

julia> x
5

julia> x===y
false
```

Cuando se pasa un objeto de tipo *Array* como parámetro en una función, se crea un alias, una nueva referencia al objeto en cuestión. Por esa razón, al realizar cambios en el alias, también afectan a la variable original (global). El siguiente ejemplo ayuda a entenderlo:

```
julia> a=[1, 3, 5]
3-element Array{Int64,1}:
 1
 3
 5

julia> function apilar(arr::Array)
           push!(arr,5)
       end
apilar (generic function with 1 method)

julia> apilar(a)
4-element Array{Int64,1}:
 1
 3
 5
 5

julia> a
4-element Array{Int64,1}:
 1
 3
 5
 5
```

En este ejemplo se inicializa un array a de 3 elementos. Después se crea una función

a la que se le pasa el la variable a como argumento; lo que hace esta función es apilar un nuevo elemento en el array. Claro está que en la función, la variable *arr* actúa como variable alias. Se realizar la llamada a la función y al preguntar por el contenido de la variable global vemos que esta también ha cambiado.

3.2.8. Estructuras y objetos

Aparte de los tipos que ofrece JULIA, también existe la opción de crear tipos propios; es decir, estructuras creadas por el usuario cuyos atributos define a su manera. Por ejemplo, si se quiere definir una coordenada:

```
1 julia > struct Coordenada
2             x::Float64
3             y::Float4
4         end
```

Es evidente que hay muchas maneras para definir una coordenada; se podrían haber utilizado simplemente 2 variables o incluso una tupla. Sin embargo, crear nuevos tipos tiene algunas ventajas con respecto a las otras opciones.

La estructura *Coordenada* tiene dos campos. Una vez definida, el siguiente paso será crear un punto. Para hacerlo, se trata la estructura como si fuera una llamada a una función.

```
1 julia > coord = Coordenada(1.0,2.0)
2 Coordenada(1.0,2.0)
3
4 julia > coord.x
5 1.0
6
7 julia > coord.y
8 2.0
```

Lo que JULIA devuelve al llamar a la función constructora es una referencia al objeto Coordenada; es decir, un puntero a la estructura. Para obtener los valores de los campos de la estructura, basta con utilizar el operador punto.

Estas estructuras son inmutables. Sin embargo, JULIA ofrece la posibilidad de crear estructuras mutables para poder acceder a sus campos y cambiarlos en caso de que sea necesario.

```
julia > mutable struct Coordenada1
            x::Float64
            y::Float4
        end

julia > coord= Coordenada1(0.0,0.0)
Coordenada1(0.0,0.0)

julia > coord.x = 1.0
1.0

julia > coord.y = 2.0
2.0
```

3.3. TaylorSeries.jl

Una vez explicadas las características principales de JULIA, se explicará detenidamente el código que hubo que manipular para obtener los resultados esperados. Este código es abierto, cualquiera puede estudiarlo y proponer mejoras. El paquete *taylorseries.jl* [Benet, 2014] contiene muchos ficheros pero los más importantes y los que hubo que entender bien eran aquellos que realizaban las operaciones y los que definían las estructuras que utilizaba el programa para tratar las series.

Por eso, lo primero fue estudiar y entender como gestionaba JULIA las series. Lo hace mediante estructuras. Estas estructuras tienen dos componentes principales:

1. La tabla de coeficientes
2. El orden de la serie

Los coeficientes van ligados al orden de la serie ya que es el propio orden el que fija el número de coeficientes del arreglo. Pongamos un ejemplo utilizado en Julia (líneas de comando):

```
1:  julia > x= Taylor1(4)
```

En esta línea de código se crea una serie de Taylor de orden 4 (tendrá 5 coeficientes) que almacenará la variable ”x”. Una vez creada la variable, se utiliza el operador punto para acceder a cada uno de sus campos. Se hace de la siguiente manera, utilizando estos comandos:

```
julia > x.coeffs

0.0   % Primer coeficiente (índice 0: x.coeffs[0])

1.0   % Segundo coeficiente (índice 1: x.coeffs[1])

0.0  % Tercer coeficiente (índice 2: x.coeffs[2])

0.0  % Cuarto coeficiente (índice 3: x.coeffs[3])

0.0  % Quinto coeficiente (índice 4: x.coeffs[4])

julia > x.order

4
```

El campo *coeffs* de la variable x es un arreglo que contiene los coeficientes de la serie. Si la serie es de orden 4, el arreglo será de longitud 5. Tomemos la serie ya creada:

$$x = 0{,}0 + 1{,}0(t) + 0{,}0 + 0{,}0 + 0{,}0 + O(t^5) \quad (3.2)$$

Al crear una serie, la consola muestra sus coeficientes y un último término que indica el error. Es importante saber que JULIA, cuando crea la serie, inicializa a 1 el segundo coeficiente de la misma; es decir, el que acompaña a la variable t.

Por otro lado, el campo *order* devuelve el orden de la serie, en este caso, 4. Estas dos propiedades o campos se encuentran dentro de una estructura tipo Taylor1. Las series de tipo *Taylor1* son estructuras inmutables en su orden; es decir, una vez las creas, su longitud es fija. Únicamente se puede acceder y cambiar el contenido de sus coeficientes.

Es importante saber como utilizar esta estructura *Taylor1* ya que se trabajará con ella

constantemente. Por esa razón, interesa saber que nos ofrece dos posibilidades a la hora de acceder a los coeficientes de las series.

```
julia> x=Taylor1(3)
 1.0 t + [U+FFFD](⁴t)

julia> x.coeffs[0]
ERROR: BoundsError: attempt to access 4-element Array{Float64,1} at index [0]
Stacktrace:
 [1] getindex(::Array{Float64,1}, ::Int64) at .\array.jl:788
 [2] top-level scope at REPL[9]:1

julia> x.coeffs[1]
0.0

julia> x[0]
0.0
```

Este es un claro ejemplo de las 2 opciones que hay: tras crear la serie, se intenta acceder al índice 0 del campo coeffs. Sin embargo, la consola muestra un error ya que el primer coeficiente de este campo se encuentra en el índice 1. La segunda manera (x[0]) es haciendo un uso directo de la estructura pero en este caso si que permite acceder a través del índice 0. Pese a que no parece importante, es un dato a tener en cuenta para las futuras operaciones vayamos a realizar.

La carpeta src contiene todos los ficheros que manipulan las series, sin embargo nosotros solo veremos aquellos que nos convienen para nuestro objetivo. En primer lugar, el fichero “constructors.jl” muestra la estructura que se utiliza para almacenar las series.

```
struct Taylor1{T<:Number} <: AbstractSeries{T}
    coeffs :: Array{T,1}
    order :: Int
end
```

La definición que se muestra encima es la que se utiliza para las series que hacen uso de una única variable independiente. Tal y como se ha dicho antes, esta estructura contiene 2 campos: coeffs y order; el primero muestra la expansión de los coeficientes (cada uno de ellos almacenados en un array) y el segundo el orden de la serie.

En mi caso, se ha abordado únicamente el estudio de las series de Taylor de una única variable, pero el paquete taylorseries.jl también trabaja con series de más variables. La estructura que utiliza es igual a la que emplea con las de una variable; es decir, dos componentes: coeficientes y orden.

Una vez aprendida la estructura de una serie de Taylor, pasamos al fichero “arithmetic.jl”, que contiene todas las operaciones que hacen uso de las series, de las más sencillas a las más complicadas. Una de las particularidades de JULIA es que te permite repetir el nombre de las funciones y a la hora de ejecutarlas, accede a aquella que coincida con las entradas introducidas por el usuario. Pongamos un ejemplo:

function crear(a::Taylor1, b::Int)

function crear(c::Int)

Si se hace la llamada con un único parámetro, accederá a la segunda. Por otro lado, si se le pasa una serie y un entero, accederá a la primera.

3.3.1. Igualdad

La igualdad está definida por las siguientes líneas de código.

```
1
2  ## Equality ##
3  for T in (:Taylor1, :TaylorN)
4
5      @eval begin
6          ==(a::$T{T}, b::$T{S}) where {T<:Number, S<:Number} = ==(promote(a,b)...)
7
8          function ==(a::$T{T}, b::$T{T}) where {T<:Number}
9              if a.order != b.order
10                 a, b = fixorder(a, b)
11             end
12             return a.coeffs == b.coeffs
13         end
14     end
15 end
```

El primer bucle: for T in (:Taylor1, :TaylorN) indica cuales son los parámetros de entrada

con los que se va a ejecutar la función. En este caso acepta series de cualquier número de variables (Taylor1, TaylorN).

La función es simple, sin embargo, hay una llamada que se utiliza mucho en este paquete y que me gustaría explicar. fixorder(a,b) es una función a la que el programa llama siempre que el orden de las series que se le pasan como parámetro es distinto.

Si vamos a operar con una serie de orden 3 y otra de orden 4, "fixorder()" hace una especie de encaje en el que las dos series pasan a tener el mismo orden para poder operar entre ellas. Tras realizar la llamada (en caso de que los órdenes sean distintos) devolverá true si los coeficientes son los mismos y false en caso contrario. Es evidente que el encaje es imprescindible ya que si los órdenes fueran distintos, los coeficientes nunca podrían ser iguales.

Pongamos un ejemplo sencillo. Supongamos que creamos dos series: x que va a ser de orden 3 e y será una de orden 4. Al ser de distintos órdenes, en este caso, la variable y tendrá un coeficiente más por lo que la instrucción: x.coeffs==y.coeffs devolverá false. Sin embargo, cuando hacemos que y sea de orden 3 y repetimos la instrucción, la respuesta es la esperada ya que los coeficientes son iguales.

Puede que el orden sea el mismo pero los coeficientes distintos, en ese caso la respuesta también sería false. Como se puede ver, lo que hace el código es simple, pero hay que entender bien el código JULIA, las estructuras que utiliza y las llamadas que hace.

3.3.2. Suma y Resta

Para llevar a cabo las sumas y las restas, JULIA hace uso de las funciones f y fc. Son dos funciones genéricas y a las que accede mediante el uso de tuplas.

```
1 for (f,fc)  in ((:+,:(add!)), (:-,:(subst!)))
```

Este bucle recorre la tupla f y fc de forma que, dependiendo de la llamada que se haga, accederá a una o a otra. En otras palabras, al sumar dos series, la tupla (f,fc) se asocia con

las llamadas (+,add!) mientras que al restarlas, lo hace con (-,subst!). Las funciones que en su nombre utilizan el signo de exclamación son aquellas que modifican algún parámetro de entrada y devuelven el resultado en la propia variable que ha sido modificada.

El código de la suma vale para la resta, y viceversa, ya que a la hora de operar series, estas operaciones hacen lo mismo sea cual sea la entrada. Se va a explicar el código para poder entenderlo mejor.

```
for T in (:Taylor1,:TaylorN)
```

Lo primero de todo decir que las entradas que estas funciones permiten son series de una variable (Taylor1) y series de más de una variable (TaylorN). Este trabajo se centrará en mejorar las de una única variable.

Lo primero de todo, se van a explicar las funciones add! y subs! (estas son las que cambian el dato), estas, a su vez, llaman a las funciones + y -.

```
## add! and subst! ##

           function ($fc)(v::$T{T}, a::$T{T}, k::Int) where {T}
               @inbounds v[k] = ($f)(a[k])
               return nothing
           end
           function ($fc)(v::$T, a::NumberNotSeries, k::Int)
               @inbounds v[k] = k==0 ? ($f)(zero(v[0]),a) : zero(v[k])
               return nothing
           end
           function ($fc)(v::$T, a::$T, b::$T, k::Int)
               @inbounds v[k] = ($f)(a[k], b[k])
               return nothing
           end
           function ($fc)(v::$T, a::$T, b::NumberNotSeries, k::Int)
               @inbounds v[k] = k==0 ? ($f)(a[0], b) : ($f)(a[k], zero(b))
               return nothing
           end
           function ($fc)(v::$T, a::NumberNotSeries, b::$T, k::Int)
               @inbounds v[k] = k==0 ? ($f)(a, b[0]) : ($f)(zero(a), b[k])
               return nothing
           end
```

Se puede observar que todas estas funciones son parecidas, de hecho todas tienen el mis-

mo nombre, pero si que hay una pequeña diferencia entre todas ellas. La diferencia se encuentra en los parámetros de entrada; es decir, en base a cuales sean accederá a una o a otra función.

Hay un tipo de dato en JULIA que se expresa tal de esta manera: T{T}. Sabemos que T es un tipo (en este caso Taylor1), pero entonces... ¿qué es T{T}? Como su propio nombre indica, es un tipo que a su vez contiene otro tipo, en este caso, series que contienen series.

En otras palabras, es una de serie de series que acaba formando una matriz. Cada coeficiente de esta serie es a su vez otra serie que puede ser del mismo o de distinto orden.

```

julia> x=Taylor1(Taylor1(3),7)

1.0t + O(t^{4}) + O(t^{8})

julia > x.coeffs

8-element Array{Taylor1{Float64},1}: \\

    1.0 t + O(t^{4})
    0.0 + O(t^{4})
    0.0 + O(t^{4})
    0.0 + O(t^{4})
    0.0 + O(t^{4})
    0.0 + O(t^{4})
    0.0 + O(t^{4}
    0.0 + O(t^{4})
```

Lo que se le dice a JULIA al crear este dato es que cree una serie de orden 7 cuyos coeficientes sean series de orden 4. Recordemos que los órdenes de estas series son inmutables, por lo que una vez creadas no se podrá modificar su tamaño, únicamente su contenido.

PRIMERA FUNCION

En este caso se realiza la llamada a f con un único operando y el resultado se deja en el índice k de v.

```
function (fc) (v::T{T},a::T{T} k::Int)

    @inbounds v[k] = (f)(a[k])

    return nothing  $

end
```

SEGUNDA FUNCION

En esta función aparece un concepto importante en JULIA: la interrogación. Lo que quiere decirnos esa línea es lo siguiente. Si se cumple la condición (k==0), hace la llamada a f con los coeficientes de v inicializados a 0 y con a. Por el contrario, si k!=0, devuelve directamente todos los coeficientes de la serie resultante a 0.

La pregunta es, ¿por qué únicamente suma el operando a si k==0? La respuesta es que a solo puede ir en el primer índice de v, ya que sería su término independiente. El primer coeficiente de una serie siempre hace referencia a este término.

```
function (fc) (v::T,k::Int) where  {T}

    @inbounds v[k] = k==0 ? (f)(zero(v[0]),a) : zero(v[k])

    return nothing

end
```

TERCERA FUNCION

Esta función llama a f cuando la entrada son dos operandos, ambos en forma de series de Taylor.

```
function (fc) (v::T,a::T,b::T k::Int)

   @inbounds v[k] = (f)(a[k],b[k])

   return nothing

end
```

CUARTA FUNCION

Al igual que antes, si k==0 hace la llamada a f con el primer coeficiente de a y con b; en caso contrario la hace con el coeficiente k de a y la variable b puesta a 0.

```
function (fc) (v::T,a::T, b::NumberNotSeries,k::Int)

    @inbounds v[k] = k==0 ? (f)(a[0], b) : (f)(a[k], zero(b))

    return nothing

end
```

QUINTA FUNCION

Esta función es muy parecida con la diferencia que los operandos se pasan en orden contrario. Sin embargo, la ejecución es la misma pero fijándonos en que la llamada en este caso se hace con el coeficiente 0 o k de b ya que es esta variable la que contiene la serie.

```
function (fc) (v::T,a::NumberNotSeries,b::T, k::Int)

    @inbounds v[k] = k==0  ? (f)(a, b[0]) : (f)(zero(a), b[k])

   return nothing

end
```

Como se ha dicho anteriormente, las funciones add!() y subst!() son las que llaman a +() y a -() por lo que se va a proceder a explicar estas dos últimas. Como antes, las variables entran en forma de T(serie simple de Taylor) y T{T} (serie de series).

PRIMERA FUNCION

Cuando los órdenes de las series entre las que se va a operar son distintos, se hace una llamada a la función fixorder() con el objetivo de hacer una especie de ajuste de estos. Una vez realizado, se crea una serie que almacenada en v con la llamada similar(). Esta función crea un array de la misma longitud pero con distintos coeficientes. Por último, se almacenan los coeficientes resultado de la suma de ambos operandos en y devuelve la serie resultante de estos coeficientes y el orden de a.

```
function f(a::T{T}, b::T{T}) where {T<:Number}

     if a.order != b.order

        a, b = fixorder(a, b)

     end

     v = similar(a.coeffs)

     @dot v = f(a.coeffs, b.coeffs)

     return T(v, a.order)

end
```

SEGUNDA FUNCION

Al igual que antes, crea un array de la misma longitud que a y asigna sus coeficientes a v. Por último crea la serie devolviendo sus coeficientes junto con el orden del operando.

```
function f(a::T)

    v = similar(a.coeffs)

    @dot v = f(a.coeffs)

    return T(v, a.order)

end
```

TERCERA FUNCION

Antes de explicar la siguiente función, es importante entender la manera en la que JULIA opera algunos datos. Si los parámetros entrantes son una serie de series y una serie simple, JULIA hace una especie de suma por "columnas"; es decir, la variable resultante va a ser una serie de series en la que solo varía su primera columna. El único coeficiente que cambia tras realizar la operación es el primero, (que es en el que se realiza la suma del primer coeficiente de la serie de series con el segundo operando que es una serie simple) aunque se verá mejor explicado paso a paso.

Se crea una variable coeffs en los que hace una copia de los coeficientes de a. La siguiente línea de código confirma lo explicado anteriormente. Solamente varía el primer coeficiente de coeffs en el que dejamos la suma del primer coeficiente de a(que es una serie) con b (que a su vez es otra serie) y devuelve la serie con estos coeficientes y el orden de a.

```
1 function f(a::T{T}, b::T) where {T<:Number}
2
3     coeffs = copy(a.coeffs)
4
5     @inbounds coeffs[1] = f(a[0], b)
6
7     return T(coeffs, a.order)
8 end
```

CUARTA FUNCION

Esta función es exactamente igual que la anterior pero con los operandos en orden inverso. Lo que hay que tener claro es que en una suma de este tipo la estructura que va a prevalecer es la grande, en este caso la serie de series. Es por eso por lo que se hace la copia de los coeficientes de a. Luego, el coeficiente que varía es el primero, resultado de la suma del primer índice de a con b.

Todas las funciones anteriores han sido explicadas para la operación suma pero también pueden aplicarse para la resta. Por esa razón van de la mano, el código es exactamente igual para ambas operaciones. Sin embargo, no pasa lo mismo con la multiplicación.

```
function f(b::T, a::T{T}) where {T<:Number}

    coeffs = similar(a.coeffs)

    @dot coeffs = f(a.coeffs)

    @inbounds coeffs[1] = f(b, a[0])

    return T(coeffs, a.order)

end
```

3.3.3. Multiplicación

En el caso de la multiplicación, JULIA distingue 2 casos principales. Estos casos los separa en función de los operandos que se le pasan al programa. Por un lado, está la multiplicación de un escalar por una serie.

La función incluída debajo tiene como parámetros un escalar (a) y una serie simple de Taylor (b). Lo primero que hace es crear una variable auxiliar (aux) a la que asigna el primer índice de la multiplicación de ambos operandos. En este caso cada índice será de tipo Float.

```
## Multiplication ##
for T in (:Taylor1)

    @eval begin

        function *(a::T, b::$T) where {T<:NumberNotSeries}

            @inbounds aux = a * b.coeffs[1]
            v = Array{typeof(aux)}(undef, length(b.coeffs))
            @__dot__ v = a * b.coeffs
            return $T(v, b.order)
        end

        *(b::$T, a::T) where {T<:NumberNotSeries} = a * b
    end
end
```

Después crea un array (v) de *Float* de longitud (b.coeffs) y asigna a cada índice de este nuevo array el resultado de multiplicar el escalar por todas las componentes de b. Este

array contendrá los coeficientes de la serie resultante que crea en la última línea de código junto con el orden de la nueva serie.

JULIA trae implementadas estas operaciones por lo que no es complicado programarlas. Sin embargo, cuando los operandos son dos series la cosa cambia. Lo que hace JULIA es tratar estas series como si fueran polinomios (al fin y al cabo es lo que son); es decir, sabiendo la forma en la que se multiplican 2 polinomios, también sabremos multiplicar dos series.

Al igual que antes T y W adquieren el valor dado por la tupla (Taylor1,Number) con la que se va a trabajar. En este caso las entradas (operandos) son series por lo que las operaciones son algo más complejas que las de una simple multiplicación de escalar por serie. Como he dicho previamente, se realiza la multiplicación de 2 polinomios. Lo único a destacar en este tipo de multiplicaciones es que una vez se realizan las multiplicaciones internas, el proceso requiere de la suma de los términos con mismo grado de la variable independiente.

```
1  for (T, W) in (:Taylor1, :Number)
2
3      @eval *(a::Taylor1, b::Taylor1)= *(promote(a,b)...)
4
5      @eval function *(a::Taylor1, b::Taylor1)
6          if a.order != b.order
7              a, b = fixorder(a, b)
8          end
9          c = Taylor1(zero(a[0]), a.order)
10         for ord in eachindex(c)
11             mul!(c, a, b, ord) # updates c[ord]
12         end
13         return c
14     end
15 end
```

Se ve que la función * tiene como entrada dos series de Taylor. Tras llamar al fixorder() para ajustar órdenes, es donde hace la reserva de memoria: crea una serie nueva (serie resultante) a la que le dice el orden que va a tener y cuyos coeficientes irá rellenando a medida que realicemos las operaciones internas.

El bucle recorre cada orden correspondiente a cada índice de la serie resultante que vamos

a llenar. Por poner un ejemplo, si el orden de c (serie resultante) es de 3, se harán 4 llamadas de * a mul! ya que el bucle principal se recorre de 0 a 3.

```
@eval @inline function mul!(c::Taylor1, a::Taylor1, b::Taylor1, k::Int)

    @inbounds c[k] = a[0] * b[k]
    @inbounds for i = 1:k

        c[k] += a[i] * b[k-i]

    end

    return nothing
end
```

Tras entrar en la llamada interna, se hacen las operaciones necesarias que permiten rellenar el contenido de los coeficientes de la nueva serie resultante de operar a y b.

4. CAPÍTULO

Soluciones investigadas

4.1. Problema propuesto

El problema que se plantea en este trabajo de fin de grado consiste en intentar mejorar los tiempos de ejecución de las operaciones entre series. Para conseguirlo, hay que fijarse en el código TaylorSeries.jl, concretamente en el fichero arithmetic.jl en el que están programadas todas las operaciones.

Una vez entendido el código tenemos que hacernos la siguiente pregunta, ¿cuál es el problema a abordar?. Mejorar la eficiencia de las operaciones mediante el ahorro de memoria. Si nos fijamos bien, la mayoría de las funciones programadas hacen un uso de memoria del que podríamos prescindir ya que lo único que consiguen es ralentizar el tiempo de las operaciones. Mejor verlo con el código concreto.

Por ejemplo, tanto en la suma como en la resta, JULIA hace un uso constante de las siguientes líneas de código:

1. La función *fixorder(a,b)* (auxiliary.jl): el programa llama a esta función cuando los parámetros de entrada son dos series de distinto orden. Veamos su código:

```
@inline function fixorder(a::T, b::T)

     a.order == b.order and return a, b

     minorder, maxorder = minmax(a.order, b.order)

     if minorder <= 0

        minorder = maxorder

     end

     return T(copy(a.coeffs), minorder), T(copy(b.coeffs), minorder)

end
```

En la última línea del programa, la función devuelve una copia de los coeficientes junto con su orden; es decir, hace una reserva de memoria para crear una serie. El orden que devuelve en ambas es el mismo ya que el objetivo de fixorder() es devolver el orden menor entre las series entrantes.

Esta es la razón por la que el resultado de las operaciones entre series de distintos órdenes contiene el orden menor entre ambas. Creo que es importante recalcar que esta llamada se hace únicamente cuando los dos parámetros de entrada son del mismo tipo ya que en caso de no ser así no es necesario realizarla.

2. Llamadas que realizan copias de los coeficientes que también realizan una reserva innecesaria de memoria como por ejemplo:

```
v = similar(a.coeffs)
coeffs = copy(a.coeffs)
```

3. Por último, al devolver la serie, también se hace una reserva de memoria:

```
return T(v,a.order)
return T(coeffs,a.order)
```

Hay que pensar que todas estas reservas se realizan cada vez que la función es llamada. En el caso de la multiplicación, como se ha explicado anteriormente, existen dos funciones principales. La primera, que realiza la multiplicación de un escalar por una serie y la segunda que opera una serie con otra serie. Al igual que antes, se van a mencionar las reservas de memoria que se realizan en ambas llamadas:

1. Escalar por serie

```
v = Array {typeof(aux)}(undef, length(b.coeffs))

return T(v, b.order)
```

La primera reserva es para el array de coeficientes que contiene el resultado de la operación y la segunda es para crear la serie con ese array y el orden de b.

2. Serie por serie

Esta función es algo más compleja, pero se ven rápidamente las reservas que realiza.

```

a, b = fixorder(a, b) (en caso de ordenes distintos)

c = T(zero(a[0]), a.order)

Estas dos instrucciones son dos claros ejemplos de las reservas de memoria realizadas. La primera
      con la llamada a la función fixorder() y la segunda en el momento en el que se está creando
      la serie.
```

4.2. Soluciones propuestas

Una vez entendido el código y comprendido cuales eran las partes que había que mejorar, se pensó en una solución que pudiera aplicarse a todas las operaciones. La idea era crear una estructura que guardara los resultados de las operaciones realizadas para que pudieran utilizarse posteriormente.

Lo que se pretendía hacer era lo siguiente. Crear una variable global de gran dimensión en la que se iban guardando todos los resultados de las operaciones. El objetivo de esto era ahorrar tiempo en las reservas de memoria ya que se realiza una única reserva para luego simplemente ir asignando los resultados en la nueva estructura.

Además, esta estructura es de gran utilidad ya que permite operar series nuevas con resultados de otras operaciones que se almacenan en la estructura. Se consiguió llegar a una solución después de probar muchas estructuras distintas:

- *Pila*
- *Struct(ARRAY(TUPLE)-CONTADOR)*
- *ESTRUCTURA(ARRAY(FLOAT)-ORDEN*
- *ARRAY-ESTRUCTURA ORDENES*
- *ARRAY-ORDEN*
- *UNICO ARRAY*

PILA

La primera idea fue crear una pila cuyos índices almacenaban series de Taylor (*Taylor1*. Sin embargo, se desechó rápidamente ya que implicaba saber de qué orden serían los resultados de las operaciones antes de realizarlas, lo que era imposible. Además, había que tener en cuenta que las series que ofrece JULIA son inmutables en su orden, una vez creadas su tamaño no puede variar, lo que supuso un gran problema.

ESTRUCTURA(ARRAY (TUPLAS)-CONTADOR)

Por eso, la siguiente idea fue crear una estructura que contuviera un par de campos. Por un lado, un array de gran dimensión cuyos índices almacenaban el coeficiente del resultado entre dos series. Pongamos un ejemplo sencillo:

$x = 1 + x + x^2$

$z = 1 + x$

Se quiere sumar estas dos series, la primera (x) de orden 2 y la segunda (z) de orden 1. La serie resultante va a ser de orden 1 por lo que en mi array solución (llamémosle y), el resultado ocuparía 2 posiciones. La primera posición (correspondiente al primer coeficiente de la suma) contendrá el resultado de x.coeffs[1] + z.coeffs[1]= 2 mientras que la segunda posición (correspondiente al segundo coeficiente de la suma) contendrá el resultado de x.coeffs[2] + z.coeffs[2]= 2.

En este momento surge la siguiente duda, era necesario saber donde empezaba y acababa la serie ya que de esa manera podía distinguir por un lado a qué coeficiente de la serie hacía referencia y por otro lado cual era el tamaño de la serie. Por esa razón, cree esta estructura:

```
mutable struct Point
    r:: Array (Tuple)
    cont::Int
end
```

Esta estructura contiene el array (r) donde se van introduciendo los resultados (coeficientes) de las operaciones y un contador que indica donde hay que almacenar el siguiente índice.

Para solucionar el problema de saber donde empezaba y acababa cada serie, la idea fue la siguiente. Hacer que el contenido de cada índice de este arreglo fueran tuplas. Estas tuplas almacenaban por un lado, el coeficiente resultante y por otro, la posición de dicho coeficiente. De esta manera se sabía el grado de la variable que acompañaba al coeficiente. Si la tupla era (5.0, 2) sabía que el segundo coeficiente de la serie era: 5 t. La inicialización de la estructura se hacía de la siguiente manera:

```
global y=Point((Array{Tuple{Float64,Int64}}(undef, 100)),1)
```

Por un lado, un array de 100 posiciones y por otro un contador inicializado a 1 que hace referencia a los índices del array que ya se han utilizado. Si ese contador esta a 6 (tras realizar la llamada) significa que en la siguiente operación que realice, el primer coeficiente

de la serie resultante irá en el índice 6 de la estructura y como primer coeficiente de la serie, el segundo valor de la tupla será de 1. Para acceder a las variables (array, contador) de la estructura, utilizaremos las siguientes instrucciones:

```
nombreDeLaEstructura.variable

y.r
y.cont
```

Hay que tener en cuenta que el empleo de la variable se hace de forma global, sin utilizar variables auxiliares que hacen referencia a la original. En la función, se utiliza directamente la *struct y* y su contenido va cambiando. Las primeras pruebas fueron con la suma ya que era el código más fácil de entender y de manipular.

```
function suma(a::Taylor1,b::Taylor1)

           h=min(a.order,b.order)

            if (h+1) <= (length(y.r)-y.cont+1)

               for i in 1:h+1

                   y.r[y.cont]=((a.coeffs[i]+b.coeffs[i]),i)
                   y.cont +=1
               end

            else
               y.cont=1
               return k
            end

end
suma (generic function with 1 method)
```

El código es bastante sencillo de entender. La entrada requiere como parámetros dos series de Taylor (a,b). Lo primero, se calcula el orden mínimo entre ambas ya que este será el orden de la serie resultante para depués almacenarlo en la variable h. La condición previa al bucle refleja si nuestra estructura podrá o no almacenar el resultado.

Supongamos que ya se han ocupado 96 posiciones de la estructura y que el orden de la serie resultante es 4. No hay suficiente espacio para almacenarla ya que se necesitan 5

espacios que la estructura no tiene. En ese caso, el contador se vuelve a poner a 1 y los resultados comenzarían a sobrescribirse.

En caso de poder almacenarlo, simplemente realizamos un bucle (que va desde 1 hasta el orden de la serie resultando más uno) en el que que se hace la suma de los coeficientes y van almacenando en la nueva estructura.

Sin embargo, esta propuesta no aportó cosas posas positivas ya que el acceso a mi estructura suponía un gran coste y por consiguiente, no se obtenía ninguna mejora. Por esta razón se pensó en algo distinto. Más adelante descubriría que lo que estaba costándome semejante coste era utilizar directamente la variable global sin parametrizarla.

Recalco que el código de la suma también vale para la operación de restar por lo que en la resta ocurre exactamente lo mismo.

ESTRUCTURA(ARRAY(FLOAT)-ORDEN)

Lo que se buscaba era que la estructura sirviera para sumar series con resultados de otras series. Para eso, hubo que programar 3 funciones. Por lo tanto, había que pensar en una estructura fácil de manejar y sobretodo, que mejorara los tiempos. Primero agrego la estructura para explicar la diferencia.

```
mutable struct Tay
   r:: Array
   orden::Int
end
```

La idea era crear una estructura similar a la que propone JULIA; es decir, una estructura que almacena únicamente una serie haciendo uso de dos campos: un array que contiene los coeficientes y un orden que informa del orden de la serie. Hay varias diferencias con respecto a la anterior estructura. Por un lado, los índices del Array ya no son tuplas, son de tipo *Float* por lo que nos olvidamos de saber donde empieza y acaba la serie. ¿Por qué? Porque a partir de ahora la nueva estructura únicamente almacena una serie. Por esa razón, el segundo campo hace referencia al orden de la serie que contiene la estructura.

En otras palabras, ahora solo se puede almacenar un resultado.

Lo siguiente que hice fue inicializar 3 estructuras de tipo Tay:

```
1 global y=Tay((Array{Float64}(undef, 100)),1)
2 global y1=Tay((Array{Float64}(undef, 100)),1)
3 global y2=Tay((Array{Float64}(undef, 100)),1)
```

A cada una de estas estructuras se le asocia un carácter (String) que les hace referencia (a, b o c) y donde se almacenan los resultados. Pongamos: ”a” con y, ”b” con y1 y ”c” con y2. Esto sirve para poder operar entre series y resultados de otras series. Para ello, se crearon 3 funciones:

• serie + serie: resultado siempre en el array y por lo que devuelve “a”

• serie + String (”a”, ”b” o ”c”)

• String + String (”a”+”b”, ”b”+”c” o ”a”+”c”)

Cuando en los siguientes apartados aparece el termino serie se refiere a la estructura que emplea JULIA (coeffs + orden) mientras que los Strings hacen referencia a las nuevas estructuras que se van a emplear (creadas por mí).

La primera función (serie + serie) es exactamente igual que la programada anteriormente con la diferencia de que vamos a usar la nueva estructura para almacenar los resultados. Esta función siempre almacenará el resultado en el primer vector y devolverá el carácter ”a” como resultado. De esta manera se sabe en cual de las estructuras se encuentra el resultado.

En la segunda función (serie + String), hay que realizar la operación entre una serie (JULIA) y algún resultado almacenado en una de las estructuras (y, y1 o y2). Es por eso que es necesario comprobar si el String que nos pasan como parámetro es ”a”, ”b” o ”c”.

Si es ”a”, el resultado de la operación irá a y1 y devuelve ”b”, si el parámetro que se pasa como String es ”b”, el resultado irá a y2 y devuelve “c” y así sucesivamente; es decir, devuelve un String que informa del lugar donde se encuentra el resultado de dicha operación.

Por último, **la tercera función** (String + String) guardará los resultados en la estructura que nos quede libre. Pongamos algún ejemplo para hacerlo más fácil de entender: si las entradas son ”a” y ”b”, el resultado se almacena en y2 y devuelve ”c”. Por otro lado, si las entradas son “c” y “a”, el resultado irá a y1 y devuelve b. Parece un poco rebuscado pero con el código se entiende mucho mejor. Las operaciones son sencillas pero requieren entender bien lo que se está haciendo para hacerlo correctamente.

Una vez programado y ejecutado con distintas series de diferentes órdenes, no hubo mejora con respecto al código original. Esta estructura tiene como punto a favor que es similar a la que emplea JULIA, es fácil de manejar. Sin embargo, tiene la pega de que utilizar una estructura de este tipo supone un gran coste y por consiguiente, la mejora de tiempos se hace complicada.

ARRAY-ESTRUCTURA DE ORDENES

La siguiente propuesta supuso un gran cambio ya que se comenzó a trabajar de forma independiente con coeficientes y orden. Por un lado, se utilizaban 3 arrays que almacenaban los coeficientes y por otro lado una estructura que almacenaba los órdenes correspondientes a estos arreglos. Para cada operación se crearían 3 arrays y su correspondiente estructura de órdenes. Veámoslo mejor con código:

```
1  mutable struct Ordenes
2     orden::Int
3     orden1::Int
4     orden2::Int
5
6  end
7
8  oS= Ordenes (0,0,0)
9
10 y= Array {Float64}(undef, 100)
11 y1= Array {Float64}(undef, 100)
12 y2= Array {Float64}(undef, 100)
```

Se crean los 3 arrays (todos de 100 posiciones cuyo contenido van a ser números en coma flotante) y la estructura en la que almacenamos los 3 órdenes. Estos se van a inicializar a 0. Una vez entendidos los datos de los que disponemos, plasmarlos en el código es sencillo. El código adjuntado a continuación corresponde a la primera función (serie + serie) cuyo resultado almacenamos siempre en la primera estructura (y) asociada con el String ”a”.

```
function sumaDefa(a::Taylor1,b::Taylor1)

    if (a.order<=b.order)

        h=a.order
    else
        h=b.order
    end

    t=h+1

    for i in 1:t
       @inbounds y[i]=a.coeffs[i]+b.coeffs[i]
    end

oS.orden=h
return "a"
end
sumaDefa (generic function with 1 method)
```

Destacaría la instrucción previa al return en la que se almacena el orden de la serie resultante en su correspondiente variable de la estructura. Siempre que las entradas sean *Taylor1 con Taylor1* los resultados se dejarán en el *Array ”y”* por lo que devolveremos el *String ”a”*.

Había que realizar la misma función para la multiplicación, aunque esta dio bastantes mas problemas que la suma, debido a que las operaciones internas que realiza son algo más complejas. El código inicial con mi estructura era el siguiente:

```
function mDefa(a::Taylor1{Float64}, b::Taylor1{Float64})

    if(a.order<=b.order)
        h=a.order
    else
        h=b.order
    end

    for ord in 0:h
        mul1!(a,b,ord,ord+1)
    end

oM.orden=h
return "a"
end

function mul1!(a::Taylor1, b::Taylor1,v::Int,s::Int)
    @inbounds k[s] = a[0] * b[v]

    @inbounds for i in 1:v

        k[s] += a[i] * b[v-i]

    end
end
```

El código adjuntado muestra la multiplicación utilizando la nueva estructura. Se puede ver que es muy parecida a la puesta anteriormente. Hay un par de cambios apreciables; el primero es que la llamada a la función mul1! se hace añadiendo un nuevo parámetro (ord+1). Este cambio es debido a que las series originales comienzan en índice 0 mientras que los arrays que yo uso como estructura comienzan en el 1. Por esa razón llevo esa variable extra.

El segundo cambio se encuentra antes del return, cuando asignamos a la variable oM.orden la variable h (igual que en la suma). Por último devolvemos *"a"* ya que el resultado queda guardado en el *Array "k"*.

Sin embargo, tras realizar muchas pruebas, la multiplicación que empleaba mi nueva estructura no mejoraba a la del código original por lo que hubo que cambiar algunas cosas de la operación interna que estaban ralentizando el tiempo de ejecución del programa.

```
function mul1!(a::Taylor1, b::Taylor1,v::Int,s::Int)
    @inbounds aux = a[0] * b[v]

    @inbounds for i in 1:v

        aux += a[i] * b[v-i]
    end
    k[s]=aux
end
```

Se ha adjuntado únicamente la función mul1! ya que es la única parte del código en la que se han realizado cambios. En esta función, a diferencia de la otra, se crea una variable auxiliar que hace el papel de acumulador. Esto consigue una mayor eficiencia en cuanto a tiempo ya que en vez de acceder continuamente a nuestra estructura, vamos acumulando el resultado en una variable. Después, hacemos la asignación para dejar lo acumulado en nuestro arreglo.

	MULTIPLICACION	
	t(ORIGINAL)	t(NUEVO)
orden 3 y 5	27.7(s)	16.14(s)
orden 5 y 7	36.7(s)	23.8(s)
orden 7 y 9	44.75(s)	31.71(s)
orden 9 y 11	49.2(s)	40.14(s)
orden 11 y 1	54.6(s)	49.85(s)
orden 13 y 1	60.6(s)	58.52(s)

	SUMA Y RESTA	
	t(ORIGINAL)	t(NUEVO)
orden 3 y 5	27.23(s)	17.45(s)
orden 5 y 7	34.8(s)	23.9(s)
orden 7 y 9	40.8(s)	29.73(s)
orden 9 y 11	43.3(s)	37.52(s)
orden 11 y 1	46.1(s)	44.06(s)
orden 13 y 1	49.7(s)	51.24(s)

	MULTIPLICACION
	t(ORIGINAL)
orden 3 y 3	10.11 (s)
orden 5 y 5	16.68 (s)
orden 7 y 7	19.03 (s)
orden 9 y 9	22.01 (s)
orden 11 y 1	25.42 (s)
orden 13 y 1	29.2 (s)
orden 15 y 1	32.66 (s)

	SUMA Y RESTA
	t(ORIGINAL)
orden 3 y 3	8.77 (s)
orden 5 y 5	13.91 (s)
orden 7 y 7	14.77 (s)
orden 9 y 9	15.93 (s)
orden 11 y 1	16.68 (s)
orden 13 y 1	17.89 (s)
orden 15 y 1	18.49 (s)

Figura 4.1: Tabla que muestra los tiempos obtenidos con entradas de órdenes distintos e iguales

Esta fue la primera estructura en la que se comenzaron a mejorar los tiempos. No suponía una gran mejora, pero era un comienzo. La primera columna muestra el tiempo que JULIA necesita para realizar un millón de operaciones, tanto en suma como en multiplicación mientras que la columna derecha refleja la pequeña mejoría obtenida con nuestra estructura. Las dos tablas de arriba muestran los tiempos obtenidos cuando las entradas son de distinto orden. Además, se puede apreciar que a medida que los órdenes crecen los

tiempos se van equiparando.

Las dos tablas de abajo reflejan que, cuando los operandos son series del mismo orden, el código original tarda bastante menos tiempo en realizarlas, aunque sigue siendo bastante más lento que nuestro código el cual invierte el mismo tiempo que cuando las entradas son de distinto orden. Esto es debido a la función fixorder(), que realiza un gran número de reservas. Esta función solo es llamada cuando los operandos son de órdenes distintos, de ahí el ahorro de tiempo.

Esta estructura tiene una gran pega con respecto a las anteriores y es que los coeficientes y el orden se almacenan en variables separadas; es decir, no comparten estructura. El objetivo final era emplear una estructura que almacenase ambos campos y que fuera fácil de utilizar.

Estas mejoras se conseguían al realizar la operación de serie (*Taylor1*) + serie (*Taylor1*). Sin embargo, al programar los códigos para realizar el serie (*Taylor1* + índice (*mi estructura*) y el índice + índice, los problemas comenzaron a llegar. Los tiempos se disparaban con respecto al serie + serie y como además, no mejoraban al código original que JULIA planteaba. Por esta razón, hubo que volver al proceso inicial y pensar en algo nuevo que mejorara los tiempos para estas 3 nuevas funciones que había que programar.

ARRAY-ORDEN

Como se ha explicado anteriormente, el método empleado crea variables globales que utiliza en la propia función de forma local, sin crear nuevas referencias a los objetos; es decir, no es necesario parametrizarlos ya que se han creado fuera de la función. Sin embargo, los arreglos son objetos mutables y si se pasan como parámetros en funciones, se crea el denominado alias que hace referencia al objeto (en este caso el array). Por esta razón, se pensó en cambiar la forma de uso de los arreglos y pasarlos como parámetros ya que, al ser una estructura mutable, si cambiamos el contenido del alias, también cambiará el contenido del objeto original. Esto obligó a reprogramar todas las funciones.

En primer lugar, hay que explicar la nueva estructura que se va a utilizar. Por cada operación vamos a crear 6 variables. 3 variables de tipo *Array* que almacenarán el resultado de

las operaciones y 3 variables de tipo *Int64* que contendrán el orden correspondiente a su array. Para la suma:

y (Array) ——— oS (Int)
y1 (Array) ——— oS1 (Int)
y2 (Array) ——— oS2 (Int)

Para la resta:

r (Array) ——— oR (Int)
r1 (Array) ——— oR1 (Int)
r2 (Array) ——— oR2 (Int)

Para la multiplicación:

k (Array) ——— oM (Int)
k1 (Array) ——— oM1 (Int)
k2 (Array) ——— oM2 (Int)

El gran cambio que sufren tanto la suma como la multiplicación de serie con serie reside en los parámetros de entrada. Esta operación siempre deja los resultados en el primero (a) de los 3 arrays de los que disponemos para almacenar los resultados. Por esta razón, es el único array que se va a utilizar en la función y que, por consiguiente, se pasa como parámetro de referencia.

```
1  y=Array{Float64}(undef,100)
2  y1=Array{Float64}(undef,100)
3  y2=Array{Float64}(undef,100)
4
5  oS=0
6  oS1=0
7  oS2=0
8
9  //Inicializamos todos los ordenes a 0.
10
11 function sumaDefa(a::Taylor1,b::Taylor1,sol::Array)
12
13     if (a.order<=b.order)
14
15         h=a.order
16     else
17         h=b.order
18     end
19
20     t=h+1
21
22     for i in 1:t
```

```
23
24        @inbounds sol[i]=a.coeffs[i]+b.coeffs[i]
25
26      end
27
28  global oS=h
29  return "a"
30  end
```

La variable *arr* se utiliza como referencia a nuestro objeto original. Cuando cambie nuestra referencia, también lo hará el array original. La instrucción previa al return, es necesaria para que cambie el contenido del orden de forma global y así poder utilizarlo en el resto de funciones. No se ha añadido el código de la multiplicación ya que es más de lo mismo. A partir de ahora, las llamadas las realizaremos de la siguiente manera:

```
1  x=Taylor1(3)
2  z=Taylor1(5)
3
4  sumaDefa(x,z,y)
5  mDefa(x,z,k)
```

Una vez programadas, la mejora de tiempos fue absoluta, no solo respecto del código original sino del código mejorado previamente. Utilizar referencias a las variables globales consiguió la mejora y el ahorro de espacio esperado. Aquí se vio que el principal problema era utilizar las variables globales directamente sin hacer uso de referencias. Lo siguiente era comprobar si también los mejoraba para las operaciones de serie con índice e índice con índice.

Para estas operaciones, se le pasan los 6 parámetros al programa ya que en función del String o los Strings que entren, necesitaremos información de una u otra estructura. En el caso de serie con índice, se hace de la siguiente manera:

```
@eval function sumaDefb(a::Taylor1,d::String,v::Int64,v1::Int64,v2::Int64,arr::Array,arr1::Array,
     arr2::Array)

    if (d=="a") && (v!=0)

        if (a.order<=v)

            h=a.order
        else
            h=v
        end
        t=h+1

        for i in 1:t

            @inbounds arr1[i]=a.coeffs[i]+arr[i]

        end

        global oS1=h
        return "b"
    else
            ...
            ...
            ...
    end
end
```

Se ha puesto únicamente el ejemplo de que el String==”a” para que el código no quede largo, ya que para los otros dos Strings se realiza lo mismo. Si la entrada es ”a”, necesitaremos las variables *y,oS*. La primera ya que su contenido es uno de los operandos y la segunda porque, al ser uno de los operandos entrantes, hay que comparar su orden con el del otro operando. En este caso también parametrizaremos el array (*y1*) ya que almacena el resultado de la operación y cuyo orden (*oS1* también habrá que actualizar.

	SUMA y RESTA			
	t(serie + serie)	t(serie + índice)	ti y tf(índice + índice)	t(Original)(serie+serie)
orden 3 y 5	2.75(s)	3.73(s)	4.2(s)	27.23(s)
orden 5 y 7	2.87(s)	3.83(s)	4.3(s)	34.8(s)
orden 7 y 9	2.99(s)	3.95(s)	4.38(s)	40.8(s)
orden 9 y 11	3.02(s)	4.06(s)	4.5(s)	43.3(s)
orden 11 y 1	3.13(s)	4.2(s)	4.61(s)	46.1(s)
orden 13 y 1	3.26(s)	4.33(s)	4.8(s)	49.7(s)

	MULTIPLICACION			
	t(serie + serie)	t(serie + índice)	t(índice + índice)	t(Original)(serie+serie)
orden 3 y 5	3.45(s)	4.52(s)	4.82(s)	27.7(s)
orden 5 y 7	3.99(s)	5.37(s)	5.47(s)	36.7(s)
orden 7 y 9	4.9(s)	6.53(s)	6.31(s)	44.75(s)
orden 9 y 11	6.03(s)	8.06(s)	7.25(s)	49.2(s)
orden 11 y 1	7.5(s)	9.76(s)	8.41(s)	54.6(s)
orden 13 y 1	8.85(s)	11.78(s)	9.8(s)	60.6(s)

Figura 4.2: Tiempos suma y multiplicación

Esta tabla muestra los tiempos obtenidos con esta nueva forma de pasar los parámetros. Las 3 primeras columnas hacen referencia a lo que tarda con la nueva estructura y la columna final a los tiempos calculados con la estructura original. Para ver la diferencia en tiempos se hizo un pequeño programa que hiciera un millón de operaciones para apreciar la diferencia.

La gran pega de esta estructura es que en cierta forma, nos obliga a disponer de un gran número de variables. Hay que pensar que para cada serie se crean 2 variables: un vector para los coeficientes y un entero para el orden. Además, tener por un lado los coeficientes y por otro lado el orden no es nada cómodo para el programador ya que tiene que estar pendiente de donde tiene cada dato. Por esta razón, al descubrir que parametrizando los tiempos mejoraban, el siguiente paso fue pensar una estructura que combinara ambos campos.

SOLUCION DEFINITIVA CON UN UNICO ARRAY

Este trabajo tiene como objetivo emplear estas nuevas estructuras en operaciones cuyas expresiones sean amplias y que por lo tanto, combinan suma, resta y multiplicación. Se crean 3 arrays para cada operación (3 para la suma, 3 para la resta,...), pero esto puede traer

problemas cuando las expresiones regulares sean muy grandes. La manera de organizar los arreglos nos limita ya que obliga a ejecutar la expresión de una manera determinada para que el resultado sea correcto.

En nuestra estructura, al multiplicar dos series, el resultado queda en el *array k* y se devuelve como resultado el *String a*. Además, se dispone de una función que opera serie con índice (String que hace referencia a nuestra estructura). Teniendo en cuenta lo explicado, si tenemos la siguiente expresión regular, tendremos un problema.

c=s+x*z

Con nuestro código, la llamada sería:

sumaDefb(s,mDefa(x,z,y),oS,oS1,oS2,y,y1,y2).

Se le suma a s el resultado de multiplicar x e y. Sin embargo, el problema viene al realizar la suma. La función sumaDefb() recibe como parámetros la serie y el índice (en este caso ”a”, resultado de multiplicar x y z) pero lo que no sabe es que hay que acceder a *k* (donde se ha dejado el resultado de la operación) y no a *y*.

La aparición de este problema obliga en cierta medida a cambiar la estructura o la forma de parametrizar las entradas. El hecho de que los arreglos estén asociados con un carácter limita la forma de trabajar. Por esa razón, la idea fue la siguiente: en vez de pasar como parámetro el String y en base a su valor asociarlo con el array, pasar directamente el array que queremos operar como parámetro.

Sin embargo, a diferencia de las soluciones anteriores en las que el array lo contenía una variable y el orden otra variable, en esta solución se utiliza un único array cuyo primer índice almacena el orden de la serie que va a contener. Como se ha dicho anteriormente, el objetivo era conseguir una estructura que contuviera ambos campos y esta lo conseguía. Además, se ahorran un gran número de parámetros de entrada. Pongamos un ejemplo sencillo:

julia>x=Taylor1(3)
$1{,}0t + O(t^4)$

julia>z=Taylor1(5)
$1{,}0t + O(t^6)$

```
julia>y=ArrayFloat64(undef,100)
100-element ArrayFloat64,1

julia >sumaDef1(x,z,y)
julia>y
100-element ArrayFloat64,1:
3.0
0.0
2.0
0.0
0.0
...
```

La entrada de esta llamada son dos series, una de orden 3 y otra de orden 5. Como ya sabemos, el orden de la serie resultante será 3 (y[1]) y los coeficientes (y[2],..,y[5]) contendrán los coeficientes resultantes. En otras palabras, os coeficientes de la serie comenzarán en el segundo elemento de la tabla y ocuparan y[1] + 1 elementos

Al igual que antes tenemos 3 funciones:

- *Taylor1 con Taylor1*
- *Taylor1 con Array*
- *Array con Array*

Los resultados de las 3 funciones programadas son arreglos que almacenan el contenido de las operaciones junto con su orden (en el primer índice). A continuación, se va a mostrar una comparativa de los dos tipos de llamadas: la programada con caracteres y la programada sin ellos:

Función	**Caracteres**	**Sin caracteres**
Taylor1,Taylor1	f(Taylor1,Taylor1,Array)	f(Taylor1,Taylor1, Array)
Taylor1, Array	f(Taylor1,String, Int,Int,Int, Array, Array, Array)	f(Taylor1, Array, Array)
Array, Array	f(String,String, Int,Int,Int, Array, Array, Array)	f(Array, Array, Array)

El hecho de olvidar esta asociación de carácter con vector reduce los parámetros que hay que pasarle a las funciones ya que el arreglo contiene toda la información necesaria. La propuesta anterior obligaba a parametrizar todos los arreglos y sus respectivos órdenes ya

que no se sabía con cual de ellos íbamos a operar, dependía del valor del *String*.

El último parámetro de todas las funciones refleja el arreglo donde se va a dejar la solución. Antes se devolvía un String que indicaba el sitio donde estaba el resultado pero ahora no hay necesidad de devolver nada.

La forma de almacenamiento es igual, los resultados siempre van en arreglos. Solo que ahora se le ofrece al programador un cierto número de Arrays para que los utilice a su manera. Si el programador dispone de 10 arrays donde almacenar los resultados de sus operaciones, deberá tenerlos bien organizados para saber donde tiene cada resultado. Sería raro y a la vez un problema encontrar expresiones regulares con más de 10 operandos. Si se diera el caso, el programador se vería obligado a sobrescribir resultados en alguno de ellos.

Se van a introducir una serie de códigos con la estructura final para que se vea mejor la forma en la que se ha realizado el proceso. Por un lado, las 3 funciones programadas para la suma (que también valen para la resta) y por el otro, las 3 programadas para la multiplicación.

```
1  function sDef1(a::Taylor1,b::Taylor1, sol::Array)
2
3          if (a.order<=b.order)
4
5              h=a.order
6          else
7              h=b.order
8          end
9
10         t=h+2
11         sol[1]=h
12
13         for i in 2:t
14
15             @inbounds sol[i]=a.coeffs[i-1]+b.coeffs[i-1]
16
17         end
18 end
19
20 function sDef2(a::Taylor1,b::Array,sol::Array)
21
22         if (a.order<=b[1])
23
```

```
24            h=a.order
25        else
26            h=convert(Int,b[1])
27        end
28
29        t=h+2
30        sol[1]=h
31
32
33        for i in 2:t
34
35            @inbounds sol[i]=a.coeffs[i-1]+b[i]
36
37        end
38 end
39
40 function sDef3(a::Array,b::Array,sol::Array)
41        if (a[1]<=b[1])
42
43            h=convert(Int,a[1])
44        else
45            h=convert(Int,b[1])
46
47        end
48
49        t=h+2
50        sol[1]=h
51
52        for i in 2:t
53
54            @inbounds sol[i]=a[i]+b[i]
55
56        end
57 end
```

sDef1() realiza la suma de 2 operandos tipo *Taylor1*. Lo único a destacar de este programa sería la instrucción *sol[1]=h*. Esta estructura almacena el orden en el primer índice de nuestro arreglo (pese a que asignamos un tipo *Int*, se almacenará como tipo *Float*). Esto obliga al primer coeficiente de la serie resultante a comenzar en el segundo índice, por esa razón el bucle recorre esa distancia. En cierta forma, los coeficientes se están desplazando un elemento a la derecha.

En **sDef2()** ocurre más de la mismo. Este programa realiza la suma de un operando tipo *Taylor1 con Array*. Sin embargo, hay una pequeña particularidad. El primer índice de los arreglos contiene el orden de la serie, pero es de tipo *Float*. Por esa razón, se llama a la función *convert()* para poder utilizarlo en otras operaciones.

sDef3() realiza la suma de *Array con Array*. Es muy parecida a las 2 funciones anteriores, e incluso más sencilla ya que el bucle es más fácil de programar. En el caso de las multiplicaciones, se explicará únicamente el código del programa que opera *Array con Array* ya que es el que tiene más complejidad.

```
function mDef3(a::Array, b::Array,sol::Array)

            if(a[1]<=b[1])

                h=convert(Int,a[1])

            else

                h=convert(Int,b[1])
            end

            t=h+2
            sol[1]=h
            for ord in 2:t

                 mul3!(a,b,ord,ord+1,sol)

            end


end


  @eval @inline function mul3!(a::Array, b::Array, v::Int,s::Int,vecSolucion::Array)

      @inbounds aux = a[2] * b[v]

      for i in 3:v

            @inbounds aux += a[i] * b[s-i+1]

      end

     vecSolucion[v]=aux

 end
```

La pequeña particularidad con respecto a lo programado en las anteriores multiplicaciones está en el bucle. Este comienza en 2 y va hasta orden+2. Esto ocurre porque los coeficientes de la nueva estructura comienzan en el índice 2.

	SUMA y RESTA			
	t(serie + serie)	t(serie + índice)	ti y tf(índice + índice)	t(Original)(serie+serie)
orden 3 y 5	2.24(s)	2.65(s)	2.7(s)	27.23(s)
orden 5 y 7	2.40(s)	2.79(s)	2.81(s)	34.8(s)
orden 7 y 9	2.48(s)	2.9(s)	2.89(s)	40.8(s)
orden 9 y 11	2.64(s)	3.12(s)	2.99(s)	43.3(s)
orden 11 y 1	2.68(s)	3.14(s)	3.09(s)	46.1(s)
orden 13 y 1	2.73(s)	3.20(s)	3.18(s)	49.7(s)

	MULTIPLICACION			
	t(serie + serie)	t(serie + índice)	t(índice + índice)	t(Original)(serie+serie)
orden 3 y 5	2.52(s)	4.11(s)	3.29(s)	27.7(s)
orden 5 y 7	3.14(s)	5.11(s)	3.91(s)	36.7(s)
orden 7 y 9	3.94(s)	6.3(s)	4.75(s)	44.75(s)
orden 9 y 11	5.05(s)	7.98(s)	5.76(s)	49.2(s)
orden 11 y 1	6.35(s)	9.67(s)	6.97(s)	54.6(s)
orden 13 y 1	7.94(s)	11.82(s)	8.44(s)	60.6(s)

Figura 4.3: Tiempos finales suma y multiplicación

La tabla de arriba muestra los tiempos que se han obtenido con la estructura final. Se puede apreciar que son muy parecidos a los vistos en la anterior estructura, incluso se consiguen mejorar algo más. Se ve que hemos pasado por muchas y distintas estructuras hasta llegar a la adecuada.

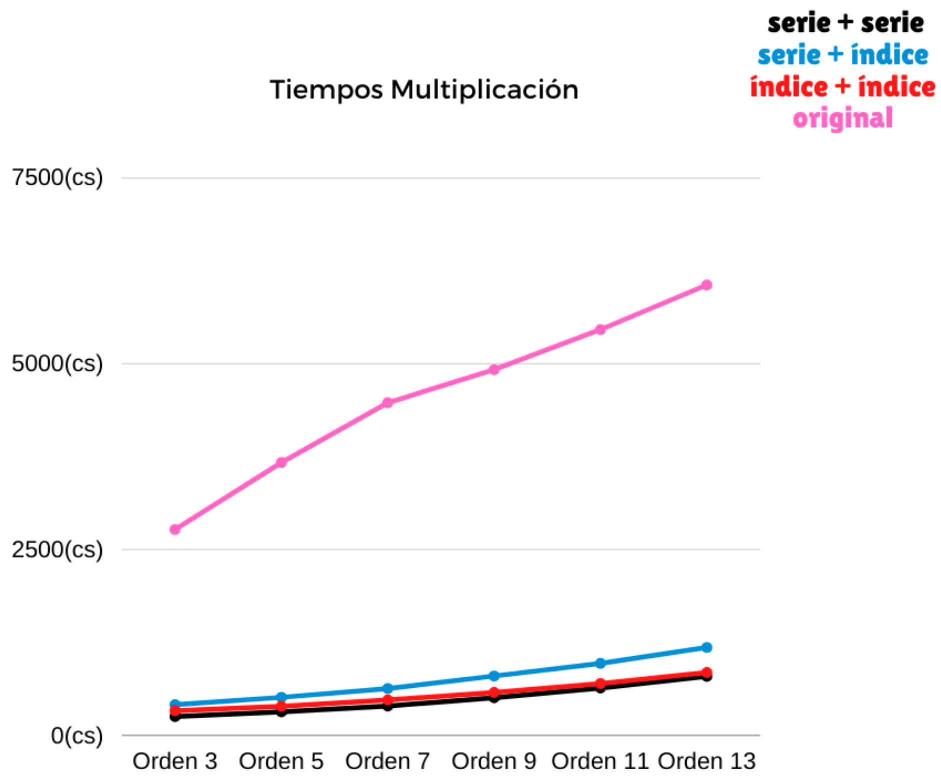

Figura 4.4: Gráfica de los tiempos obtenidos con la multiplicación

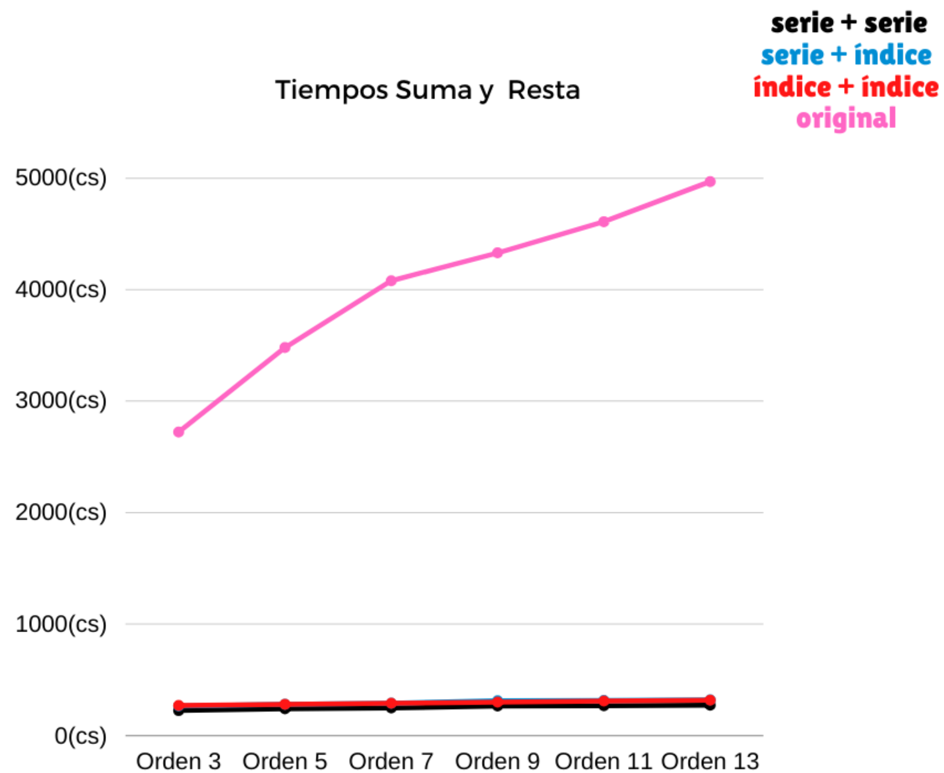

Figura 4.5: Gráfica de los tiempos obtenidos con la suma y la resta

Estas gráficas [Melanie Perkins, 2012] reflejan la comparativa entre los tiempos finales y los obtenidos con la estructura original. Se ve que hay una gran diferencia entre las rectas. La más alta hace referencia a los tiempos que se consiguen con la estructura que ofrece JULIA, mientras que las más bajas son los tiempos que se consiguen con la nueva estructura. La diferencia es evidente.

4.3. Series de Taylor para la resolución de ecuaciones diferenciales

Esta sección tiene como objetivo explicar la forma en la que se resuelven las ecuaciones diferenciales usando el método de Taylor [Sánchez, 2010]. Podremos resolverlo si conocemos el valor de una función en el instante t0 (y(t0)=y0) y si podemos conocer el valor de sus derivadas, entonces hallaremos el valor que toma la expresión de Taylor en el instante

t1.

$$y(t1) = y(t0) + (t1 - t0) * y'(t0) + \frac{1}{2}(t1 - t0)^2 * y''(t0) + \frac{1}{3!}(t1 - t0)^3 * y'''(t0) + ... + \tag{4.1}$$

Conociendo el valor de la función en el instante t0, tiene como objetivo aproximar la función en los siguientes instantes t, todo esto utilizando la expansión de Taylor. Podemos utilizar todos los términos que necesitemos para conseguir una mejor aproximación. El problema es que no conocemos las derivadas de y(t), únicamente la primera (y'(t)). Nuestro objetivo será obtener las siguientes derivadas.

Teniendo la siguiente definición recursiva:

y=f(t,y)

y sabiendo que su valor inicial es y(t0)=y0, podemos poner la solución de esta ecuación en forma de serie.

$$y(t) = y[0] + y[1] * (t - t0) + y[2] * (t - t0)^2 + y[3] * (t - t0)^3 + ... + y[k] * (t - t0)^k (3) \tag{4.2}$$

Teniendo en cuenta la condición inicial, sabemos que y[0]=y0, pero necesitamos el resto de coeficientes de nuestra serie. Para conseguirlos, utilizaremos f(t,y). Supongamos que conocemos la expansión de Taylor f(t,y) alrededor de t0. La escribiríamos así:

$$f(t, y(t)) = f[0] + f[1] * (t - t0) + f[2] * (t - t0)^2 + f[3] * (t - t0)^3 + ... + f[k] * (t - t0)^k (4) \tag{4.3}$$

En este caso f[0]=f(t0,y0) y el coeficiente f[k] de la serie son las derivadas en el instante t0, por lo que:

$$f[k] = \frac{1 * d^k}{k! dt^k}(t0)(5) \tag{4.4}$$

Suponemos que podemos conseguir el coeficiente f[k] y en eso nos puede ayudar el paquete TaylorSeries.jl. Sin embargo, para conseguir la expansión f(t,y(t)) necesitamos tener

la función y en forma de serie. En estos momentos solo tenemos el valor inicial que nos permite obtener únicamente el primer coeficiente de la expansión, pero necesitamos conocer los demás. De todas formas, ahora tenemos dos componentes de la ecuación número 3. En otras palabras, utilizando f(t,y) hemos obtenido un nuevo término de y(t).

$$y^{[1]} = y[0] + y[1] * (t - t0) + O(t^2)$$

(4.5)

Podríamos repetir ese proceso; es decir, calcular

$$f(t, y^{[1]}) \tag{4.6}$$

y esto tendrá dos términos conocidos:

$$f(t, y(t)) = f[0] + f[1] * (t - t0) + O(t^2) \tag{4.7}$$

Utilizando esta serie y teniendo en cuenta la ecuación número 2, vamos a derivar la serie 3 con respecto a t, la cual quedaría de la siguiente forma:

$$y[1] + 2y[2] * (t - t0) + O(t^2) = f[0] + f[1] * (t - t0) + O(t^2)$$

(4.8)

Una vez igualamos las dos series, hacemos lo mismo con sus potencias, y despejamos el valor de y[] que nos interesa, en este caso:

$$y[2] = \frac{f[1]}{2} \tag{4.9}$$

Ahora tenemos un nuevo término de la sucesión de y. Volviendo a repetir el proceso

podremos conseguir el siguiente. De forma genérica, quedaría así:

$$y[k] = \frac{f[k-1]}{k}$$

(4.10)

APLICACION

Tras conseguir una estructura eficiente que mejora el tiempo de ejecución del código original de taylorseries.jl, el siguiente paso es intentar aplicar estas nuevas estructuras en un problema real. En este caso un problema que hace uso de ecuaciones diferenciales en las que se realizan operaciones determinadas con series (sumas, restas, multiplicaciones). La tarea consiste en sustituir las operaciones que se realizan en el código por las operaciones programadas por mi que hacen uso de estructuras distintas.

Nosotros tenemos programadas 3 funciones que nos ofrecen la posibilidad de trabajar con distintos tipos de parámetros. Dicho esto, nos fijamos en el código original del programa que trabaja con ecuaciones diferenciales y procedemos a cambiar el código.

```
1  function NbodyODE0!(du,u,Gm,t)
2       #Time-reparametrization function s(u)=1 (no time-reparametrization)
3       N = length(Gm)
4       du[1,:,:] .= 0
5       for i in 1:N
6          xi = u[2,1,i]
7          yi = u[2,2,i]
8          zi = u[2,3,i]
9          Gmi = Gm[i]
10         for j in (i+1):N
11            xij = xi - u[2,1,j]
12            yij = yi - u[2,2,j]
13            zij = zi - u[2,3,j]
14            Gmj = Gm[j]
15            auxij = (xij*xij+yij*yij+zij*zij)^(-1.5)
16            du[1,1,i] -= Gmj*auxij*xij
17            du[1,1,j] += Gmi*auxij*xij
18            du[1,2,i] -= Gmj*auxij*yij
19            du[1,2,j] += Gmi*auxij*yij
20            du[1,3,i] -= Gmj*auxij*zij
21            du[1,3,j] += Gmi*auxij*zij
```

```
            end
        end
      du[2,:,:] .= u[1,:,:]
      return 1.
end
```

Esta función trabaja con planetas del sistema solar y resuelve problemas específicos relacionados con este tema. Concretamente lo que hace es calcular donde se encuentra cada planeta en determinados momentos; todo ello en base al valor de los parámetros que le pasemos. Las variables *xi,yi,zi,xij,yij,zij* y cada valor de la matriz *du[]* son series. Por lo cual, todas las operaciones que incluyan esos operandos tienen que sustituirse por las nuevas.

Cabe decir que la función NbodyODE0!() es una función a la que se llama desde TaylorSolve(). Esta última es la función que realiza todo el proceso del que hemos hablado en el apartado 3.6. El código de TaylorSolve() nos lo han dado ya hecho, por lo tanto, las únicas pruebas realizadas han sido sobre la función a la que llama, en este caso NbodyODE0!().

```
function func!(du,u,Gm,t)

        N = length(Gm)
        du[1,:,:] .= 0
        for i in 1:N
            xi = u[2,1,i]
            yi = u[2,2,i]
            zi = u[2,3,i]
            Gmi = Gm[i]
            for j in (i+1):N

                rDef1(xi,u[2,1,j],arr)
                rDef1(yi,u[2,2,j],arr1)
                rDef1(zi,u[2,3,j],arr2)
                mDef3(arr,arr,arr3)
                mDef3(arr1,arr1,arr4)
                mDef3(arr2,arr2,arr5)
                c=conversion(arr3)
                d=conversion(arr4)
                e=conversion(arr5)

                Gmj = Gm[j]
                auxij = (c+d+e)^(-1.5)
                auxijj=Gmj*auxij
                auxiji=Gmi*auxij



                du[1,1,i] -= conversion(mDef2(auxijj,arr,arr6))
```

```
29                du[1,1,j] += conversion(mDef2(auxiji,arr,arr7))
30                du[1,2,i] -= conversion(mDef2(auxijj,arr1,arr8))
31                du[1,2,j] += conversion(mDef2(auxiji,arr1,arr9))
32                du[1,3,i] -= conversion(mDef2(auxijj,arr2,arr10))
33                du[1,3,j] += conversion(mDef2(auxiji,arr2,arr11))
34            end
35        end
36        du[2,:,:] .= u[1,:,:]
37        return 1.
38    end
```

Se puede observar que, cuando de operaciones complejas se trata, es necesario tener un gran número de vectores que vayan guardando los resultados de las operaciones entre series. Clara está la importancia de saber donde se encuentra cada resultado. Lo que se ha hecho para hacerlo más sencillo es descomponer algunas partes del código en trozos más simples. En la parte donde hay que sumar los resultados de multiplicar, se almacenan las 3 multiplicaciones en 3 arrays para luego sumarlos. Sin embargo, es importante especificarle bien al programador la manera en la que hay que utilizar las estructuras y como hacer uso de las nuevas funciones programadas. Una vez cambiado todo el código, confiamos en mejorar también los tiempos que emplea este problema de resolución de ecuaciones diferenciales.

La función conversion() se utiliza unas cuantas veces en el código. Esta función, como su propio nombre indica, realiza la transformación de nuestra estructura tipo *Array* a una estructura tipo *Taylor1*. Para ello, se le pasa como parámetro el vector que se quiere transformar y nos devuelve la serie correspondiente. El problema de esta función es que, como TaylorSolve(), hace reservas de memoria para crear la serie. Funciones como la potencia no se han programado para nuestra estructura por lo que he procurado que esa operación se realizara con una serie original, de ahí la llamada.

```
1   function conversion(arr::Array)
2
3       h=convert(Int64,arr[1])
4       c=Taylor1(h)
5       t=h+1
6
7       for i in 1:t
8           c.coeffs[i]=arr[i+1]
9       end
10      return c
11  end
```

La llamada se realiza de la siguiente manera:

@time sol2=TaylorSolve(u0, t0, 360, delta/10., m,nmax, NbodyODE0!, Gm, order=4,variablestep=true)

En este ejemplo, se realiza la llamada con la función original NbodyODE0!(), junto con una serie de parámetros, entre ellos el periodo (360) o el orden (4). Para calcular el tiempo que le cuesta a nuestra función, basta con cambiar la función original por la reprogramada.

@time sol2=TaylorSolve(u0, t0, 360, delta/10., m,nmax, func!, Gm, order=4,variablestep=true)

Se han realizado varias pruebas para comprobar si se obtiene una mejora en el código que opera series para la resolución de ODEs. Se ha comprobado que los resultados son iguales y los tiempos que se han obtenido son similares a los recogidos por el código original. Hay que tener en cuenta que la función de resolución de ODEs TaylorSolve trabaja con series de Taylor originales, por lo que cada vez que se ejecute NBodyODE0! la función debe convertir los resultados de un formato a otro, con la consiguiente pérdida de tiempo y la consiguiente reserva de memoria.

```
t0 = 0.
delta = 0.01
m=20
nmax=20000
T = 360
```

	ODE(NbodyODE0!())	
	t(ORIGINAL)	espacio ocupado
orden 4	2.1 (s)	2.314 (GiB)
orden 10	9.59 (s)	9.594 (GiB)
orden 20	27.26 (s)	32.92 (GiB)

	ODE(func!())	
	t(ORIGINAL)	espacio ocupado(GiB)
orden 4	2.7 (s)	1.882 (GiB)
orden 10	9.85 (s)	7.49 (GiB)
orden 20	30.34 (s)	25.46 (GiB)

Figura 4.6: Comparación tiempos y espacio ocupado con estructura originsl y final en ecuaciones diferenciales

La tabla de arriba muestra algunos de los tiempos obtenidos (junto con el espacio ocupado) con el código original y con el nuevo. Se puede apreciar que se han hecho varias pruebas, cambiando el parámetro *order*. La fotografía que se va a introducir a continuación prueba que se obtienen los mismos resultados con la estructura original y con la nueva estructura. Para comprobarlo, se han obtenido las siguientes soluciones, la primera con la función original y la segunda con la reprogramada con las nuevas funciones.

@time sol2=TaylorSolve(u0, t0, 360, delta/10., m,nmax, NbodyODE0!, Gm, order=4,variablestep=true)

@time sol3=TaylorSolve(u0, t0, 360, delta/10., m,nmax, func!, Gm, order=4,variablestep=true)

```
julia> sol2.t                          julia> sol3.t
534-element Array{Float64,1}: 534-element Array{Float64,1}:
   0.0                                    0.0
   0.6358275912879001                     0.6358275912879001
   1.2612426987404564                     1.2612426987404564
   1.8771588008812594                     1.8771588008812594
   2.4844477839231933                     2.4844477839231933
   3.0839504149600927                     3.0839504149600927
   3.676485990416859                      3.676485990416859
   4.262861518767984                      4.262861518767984
   4.84388077085571                       4.84388077085571
   5.4203535300705505                     5.4203535300705505
   5.993105398687966                      5.993105398687966
   6.562988569538941                      6.562988569538941
   7.130894061778775                      7.130894061778775
   7.6977660595001955                     7.6977660595001955
   8.26461920507269                       8.26461920507269
   8.832560023001712                      8.832560023001712
   9.402814147831947                      9.402814147831947
   9.970637959559365                      9.970637959559365
  10.531777096523452                     10.531777096523452
  11.08788770457671                      11.08788770457671
   ⋮                                      ⋮
```

Figura 4.7: Prueba de que ambas soluciones dan el mismo resultado

Por último, se van a mostrar algunas de las gráficas obtenidas. Estas gráficas muestran las órbitas que realiza uno de los planetas. Para ello, se ha creado una función que obtiene por un lado, las componentes de la x y por otro lado, las componentes de la y. Una vez obtenidas, se dibuja la órbita.

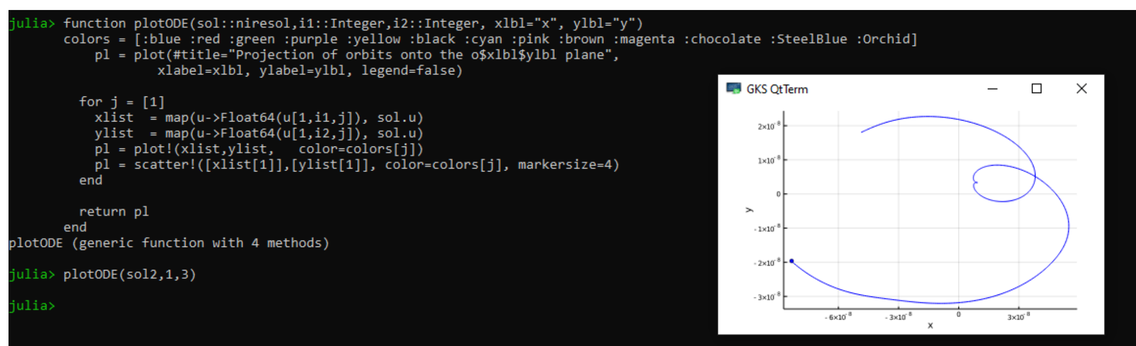

Figura 4.8: Órbita creada por uno de los planetas con el código original

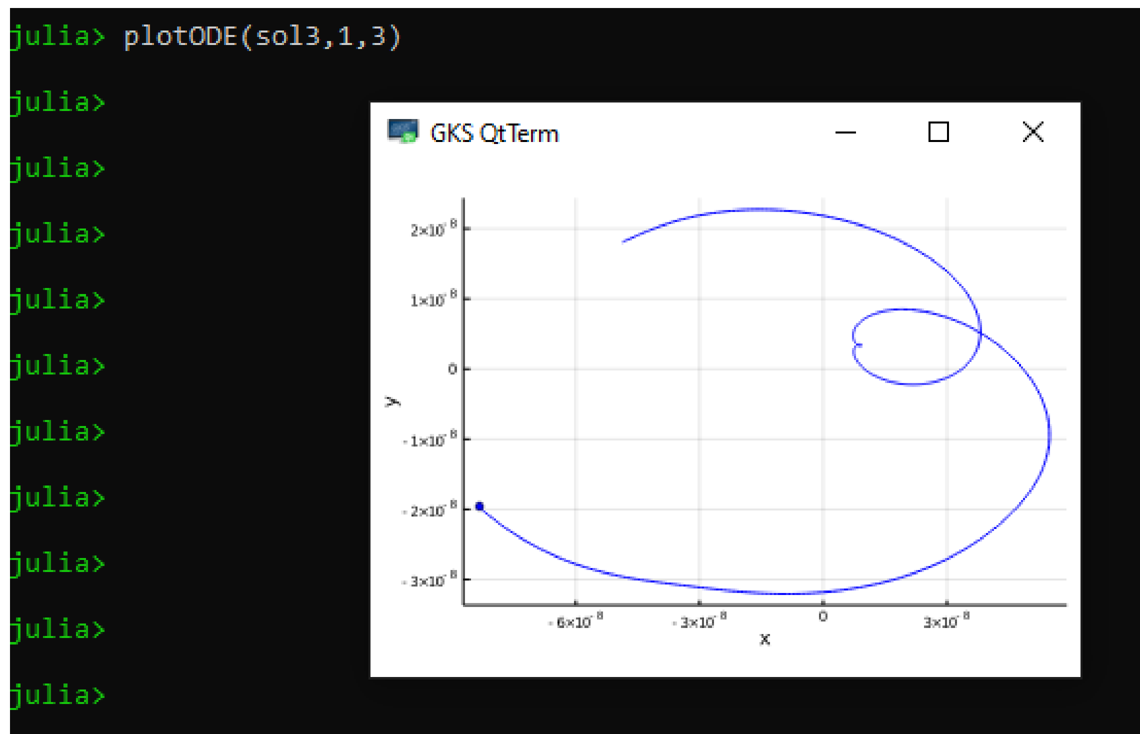

Figura 4.9: Órbita creada por uno de los planetas con el código nuevo

Sin embargo, no cabe la menor duda de que se llegarían a obtener mejoras si nuestras estructuras se utilizaran en la función principal *TaylorSolve()*. Al fin y al cabo, es únicamente la función NbodyODE0!() la que hace uso de nuestras estructuras y esta función es llamada desde TaylorSolve(). Esta última hace uso de las estructuras originales, por lo que realiza un gran número de reservas de memoria que podríamos ahorrarnos empleando nuestros vectores a modo de almacenamiento de resultados.

5. CAPÍTULO

Conclusiones

Tras realizar este trabajo de fin de grado, se ha llegado a una serie de conclusiones:

Parece evidente pero, es importante conocer y controlar bien las bases de un lenguaje de programación para comenzar a programar. Uno de los principales errores cometidos fue intentar entender el código que había que mejorar antes de haber estudiado el lenguaje JULIA y su sintaxis.

Una correcta gestión de la memoria es vital para que nuestros programas sean eficientes. Concretamente, el ahorro de memoria consigue que nos olvidemos de la creación de variables innecesarias que ralentizan el tiempo de ejecución de los programas.

Como se ha explicado en el desarrollo de la memoria, hubo que probar muchas estructuras hasta llegar a conseguir la que mejorara los tiempos. Los resultados comenzaron a llegar con la parametrización de las variables. En un principio se pensó que el problema de no mejorar los tiempos estaba en las estructuras que se estaban utilizando, por eso se desecharon. Sin embargo, lo que ralentizaba el proceso era usar esas variables globales de forma directa. Por lo tanto, se ha conseguido el objetivo principal de este trabajo que era mejorar los tiempos de cálculo en las operaciones aritméticas entre series de Taylor. Para ello ha hecho falta encontrar una estructura que usara menos memoria y que por consiguiente, mejorara los tiempos de ejecución en las operaciones entre series.

5.1. Trabajo futuro

El último capitulo explica cómo al sustituir las nuevas funciones en la función NbodyODE0!(), los tiempos obtenidos son similares a los del código original. El problema es que la función TaylorSolve() sigue haciendo uso de la estructura original (*Taylor1*) de serie, por lo que NBodyODE0! debe realizar los cálculos con la nueva estructura y convertir los resultados a (*Taylor1*) y debido a esta sobrecarga de trabajo extra es lógico que no se consiga una mejoría. Sin embargo, hemos visto que con nuestra estructura se han conseguido mejoras por lo que habría que programar un nuevo paquete que defina todas las funciones y operaciones entre series de taylor con la nueva estructura propuesta. De forma que tanto NBodyODE0!() como TaylorSolve() empleen el nuevo tipo de dato definido en el nuevo paquete como tipo de dato básico, evitando las conversiones innecesarias que hacen perder tiempo y recursos.

Definir un paquete completo basado en la nueva estructura de dato para serie de Taylor queda muy por encima de los objetivos planteados para este proyecto, por lo que se podría plantear otro Trabajo de Fin de Grado cuyo objetivo fuera llevar a cabo la implementación de este paquete.

Bibliografía

[Benet, 2014] Benet, S. (2014). Paquete taylorseries.jl. urlhttps://github.com/JuliaDiff/TaylorSeries.jl.

[Jeff Bezanson, 2017] Jeff Bezanson, Alan Edelman, S. K. V. B. S. (2017). Julia language. urlhttps://julialang.org/.

[Lauwens and Downey, 2019] Lauwens, B. and Downey, A. B. (2019). *THINK JULIA: How to think like a computer scientist*. O'Reilly Media.

[Melanie Perkins, 2012] Melanie Perkins, Cliff Obrecht, C. A. (2012). Gráficos. urlhttps://www.canva.com/es$_es/graficos/$.

[Sánchez, 2010] Sánchez, J. A. (2010). Método de las series de taylor para resolver ecuaciones diferenciales lineales y no lineales. urlhttps://www.academia.edu/19418531/MÉTODO$_DE_LAS_SERIES_DE_TAYLOR_PARA_RESOLVER_ECU$