UNIVERSITAT DE BARCELONA
Facultat de Matemàtiques i Informàtica

GRAU DE MATEMÀTIQUES

Treball final de grau

Cálculo del Ángulo entre Separatrices en el Sistema del Péndulo Rápidamente Forzado

Autor: Ignacio Rodríguez Via-Dufresne

Director: Dr. Angel Jorba
Realitzat a: Departament de Matemàtiques i Enginyeria Informàtica

Barcelona, 15 de enero de 2025

Abstract

The measure of the splitting of the separatrices of the rapidly forced pendulum

$$\ddot{x} + \sin x = \mu \ \sin\left(\frac{t}{\epsilon}\right)$$

is considered as a model problem that has been studied by different authors.

In this work, we study the angle that separates the separatrices. We observe that the splitting angle in this system is of very small orders, and thus, for its calculation, we have been compelled to use high-precision arithmetic.

Specifically, for this study, we have created a program in C that uses **MPFR** (high-precision arithmetic) to analyze how the angle varies with changes in the parameters ϵ and μ.

Resumen

La medida de las separatrices divididas del péndulo forzado rápidamente

$$\ddot{x} + \sin x = \mu \ \sin\left(\frac{t}{\epsilon}\right)$$

es considerado como un problema que ha sido estudiado por diferentes autores.

En este trabajo, estudiamos el ángulo entre las dos separatrices. Observamos que el ángulo de escisión en este sistema es de órdenes muy bajos y, por ello, para su cálculo nos hemos visto obligados a utilizar aritmética de alta precisión.

Concretamente, para su estudio, hemos creado un programa en C que hace uso de **MPFR** (aritmética de alta precisión) para analizar cómo varía el ángulo con cambios en los parámetros ϵ y μ.

2020 Mathematics Subject Classification. 37J46, 37M21, 65P99

Agradecimientos

En primer lugar, quisiera agradecer a mi tutor del trabajo, el Dr. Àngel Jorba Monte, su propuesta de trabajo y su ayuda y correcciones durante todos estos meses. Además, quiero agradecer también al resto de profesores que me han impartido clases durante todo el grado, su tiempo y su disponibilidad.

En segundo lugar, quisiera dedicarlo a mis padres y a mis hermanos, que siempre han sido mi mayor soporte en todo.

Quisiera también agradecer a todos los amigos que he conocido en la carrera y con los que he compartido estos cuatro años. Ha sido para mí un regalo compartir este tiempo con ellos, disfrutar de los mejores momentos y apoyarnos en los más difíciles.

Índice general

1. Introducción **1**

2. Contextualización del Problema **4**

2.1. Definiciones y conceptos básicos 4

2.1.1. Sistemas no lineales 5

3. Aritmética de alta precisión **9**

3.1. Definiciones y conceptos básicos 9

3.1.1. Representación de reales en punto fijo 9

3.1.2. Representación de reales en punto flotante 10

3.2. Codificación de números reales en punto flotante 10

3.3. Estándar IEEE 754 11

3.4. MPFR 11

3.4.1. Funcionamiento 12

3.4.2. Representación de datos 12

3.4.3. MPFR C++ 13

4. Método de Taylor **15**

4.1. Definiciones y conceptos básicos 15

4.2. Diferenciación Automática 16

4.3. Grado y control de paso 17

4.4. Estimación del Orden y del paso 18

4.5. Controles de paso 19

4.5.1. Primer control de paso 19

4.5.2. Segundo control de paso 19

4.6. Computación de alta precisión 19

4.7. Implementación del software 20

5. Cálculo del ángulo de escisión **21**

5.1. Cálculo órbitas periódicas 21
5.2. El método de Newton 22
5.3. Función de Newton 23
5.4. Función update_F 24
5.5. Intervalo fundamental 27
5.6. Método de la bisección 28
5.7. Vector tangente 30
5.7.1. Método de extrapolación de Richardson 31
5.7.2. Método Algebraico 33
5.8. Cálculo del ángulo y comprovación 34
5.9. Funciones auxiliares 35
5.9.1. Función calcula_veps 35
5.10. Main 36

6. Ejecución y Resultados **40**

7. Conclusiones **42**
7.1. La importancia del ángulo entre separatrices 42
7.2. El reto computacional: la necesidad de alta precisión 42
7.3. Relevancia de los resultados para la teoría de sistemas dinámicos 43
7.4. Limitaciones y posibles mejoras 43
Anexo 46

Capítulo 1

Introducción

Un péndulo sin fricción en un campo gravitacional constante es quizás el ejemplo más conocido en el campo de las matemáticas de un oscilador no lineal. La ecuación que lo rige es:

$$\ddot{x} + \sin x = 0 \tag{1.0.1}$$

donde x(t) es el ángulo entre el brazo del péndulo y una línea vertical, y t representa el tiempo.

Este es un sistema hamiltoniano con un grado de libertad

$$H(\dot{x}, x) = \dot{x}^2 + (1 - \cos x) \tag{1.0.2}$$

y es integrable.

Observemos que si perturbamos el movimiento del péndulo añadiendo una pequeña fuerza periódica, de modo que la ecuación a considerar sea:

$$\ddot{x} + \sin x = \mu \sin\left(\frac{t}{\epsilon}\right) \tag{1.0.3}$$

nos encontramos con que el sistema sigue siendo hamiltoniano:

$$H = [\frac{\dot{x}^2}{2} + 1 - \cos x] + \mu\,[-x \sin\frac{t}{\epsilon}] \tag{1.0.4}$$

pero el Hamiltoniano ahora depende explícitamente del tiempo.

Una pregunta interesante es si el sistema sigue siendo integrable o se vuelve caótico ante pequeñas perturbaciones.

Cuando el sistema es perturbado, la intersección transversal de las separatrices puede generar dinámica caótica, de acuerdo al mecanismo de Smale-Birkhoff[1]. El ángulo entre ellas ayuda a cuantificar el grado de transversalidad y la sensibilidad del sistema a perturbaciones. Por otra parte, este ángulo, es un indicador de integrabilidad, ya que en sistemas integrables, las separatrices no se cruzan transversalmente.

Sabemos por el famoso artículo de Poincaré[2] que la división de separatrices da lugar a un comportamiento complicado de las trayectorias en sistemas dinámicos hamiltonianos.

Esta ecuación se ha convertido en un problema paradigmático y, durante la historia de las matemáticas, se han aplicado diversos métodos de investigación.

El problema de medir el ángulo de las separatrices del péndulo rápidamente forzado radica en que el ángulo es de un orden muy pequeño. En particular, al tratarse con los parámetros ϵ y μ con valores relativamente pequeños provoca que el ángulo de escisión tienda a 0. Por ejemplo, con los valores $\epsilon = 0{,}001$ y $\mu = 0{,}001$ el ángulo es de orden de 10^{-690} y por ello es necesario utilizar aritmética de alta precisión para calcular el ángulo de escisión.

Si fijamos ϵ y consideramos un μ pequeño, entonces la simple aplicación de la teoría de Poincaré-Arnold-Melnikov[3] muestra que las separatrices se intersecan transversalmente con un ángulo de $O(\mu)$.

Lo que veremos en este trabajo es la creación de un código basado en aritmética de gran precisión para calcular de forma experimental el ángulo que divide las separatrices. También compararemos este resultado con métodos teóricos para el cálculo de este ángulo, observando entre ellos cuál se acerca más al método experimental propio.

Alguna de las aplicaciones clave de calcular este ángulo es el diseño de trayectorias espaciales, ya que los sistemas dinámicos que gobiernan estas trayectorias pueden modelarse con ecuaciones muy similares al péndulo forzado. Por otra parte, modelos como el péndulo forzado aparecen en sistemas de control como el robot Segway o en la estabilización de edificios frente a sismos. Finalmente, el comportamiento de partículas cargadas en campos magnéticos oscilantes puede describirse mediante sistemas análogos al péndulo forzado, lo cual tiene una gran aplicabilidad en los aceleradores de partículas. Por todo ello, podemos concluir que calcular este ángulo tiene aplicaciones prácticas en áreas tan diversas como la exploración espacial, el control de sistemas mecánicos y la física, entre otras, permitiendo comprender y optimizar esos sistemas.

Nuestro objetivo es estudiar de forma experimental y con alta precisión la medida del ángulo entre las dos separatrices del péndulo rápidamente forzado mediante la creación y aplicación de un programa en C, aprovechando las grandes oportunidades que aporta la programación científica al campo de las matemáticas.

Estructura de la Memoria

En este trabajo veremos, en primer lugar, una pequeña introducción al material teórico que necesitaremos para explicar el proceso del cálculo del ángulo, basándonos en la existencia de ciertos objetos matemáticos propios de sistemas dinámicos, como son las variedades invariantes en sistemas no lineales. En este capítulo introduciremos de forma clara y mediante ejemplos nociones básicas usadas a lo largo de todo este trabajo, como la noción de órbitas periódicas, de sección transversal, la aplicación de Poincaré y nociones utilizadas en el estudio de sistemas no lineales, como son las variedades invariantes y los teoremas que nos permiten conocer más de cerca el comportamiento del sistema a partir de estas.

En el siguiente capítulo nos concentraremos en explicar conceptos necesarios para el uso de la programación científica, como las distintas representaciones de datos y su codificación siguiendo el estándar IEEE 754.

Tras esta explicación, nos centraremos en la mejora de calidad en el campo de la programación científica al utilizar aritmética de alta precisión, como es la biblioteca MPFR, junto con el uso del wrapper MPFR C++ que mejora en gran medida la sintaxis de esta biblioteca.

El capítulo cuatro consiste en una explicación del método de Taylor para conseguir un integrador numérico de un sistema de ecuaciones diferenciales ordinarias. En este capítulo se hace un análisis detallado del paquete de software Taylor, que implementa el método con su mismo nombre, explicando detalladamente la lógica y las proposiciones matemáticas en los que se basa esta implementación y por qué motivo se ha decidido utilizar este integrador numérico y no otro, como dop853, un código explícito de Runge-Kutta de orden 8, y odex, un método de extrapolación de orden variable basado en el algoritmo de Gragg-Bulirsh-Stoer.

En el capítulo cinco, nos centraremos en la herramienta desarrollada. Hablaremos sobre nuestra implementación y creación del código, toda la lógica y el flujo de trabajo del código, conformando toda la teoría matemática que hay detrás en un programa escrito en C++, que, haciendo todo lo explicado anteriormente, calcula el ángulo que hay entre las dos separatrices principales del sistema del péndulo rápidamente forzado.

Por último, terminaremos este trabajo estudiando los diferentes resultados que da el programa según calibremos de forma distinta los parámetros ϵ y μ propios del sistema del péndulo rápidamente forzado, junto con el parámetro DIGITS, que determina el número de dígitos con los que opera el programa. Además, añadiremos al final de esta sección las conclusiones que se pueden sacar según los resultados del programa y el valor añadido que aporta este trabajo al campo de las matemáticas.

Capítulo 2

Contextualización del Problema

En esta sección introduciremos todos los conceptos básicos relacionados que sean necesarios para el objetivo de este trabajo. La mayoría de definiciones y resultados se encuentran en [4], [5] y [6].

Sabemos, a partir del famoso artículo de Poincaré[2], que la separación de separatrices da lugar a un comportamiento complicado de las trayectorias en sistemas dinámicos hamiltonianos. La ecuación más simple que proporciona un ejemplo de separación de separatrices es la ecuación del péndulo con forzamiento periódico

$$\ddot{x} + \sin x = \mu \sin \frac{t}{\epsilon}$$

Considerando el cambio de variable $\tau = \frac{t}{\epsilon}$, se puede reescribir como un sistema casi-integrable:

$$\ddot{x} + \epsilon^2 \sin x = \mu\epsilon^2 \sin \tau \ \ (= \frac{\partial}{\partial \tau})$$

Como la fuerza en la ecuación del péndulo rápidamente forzado es $2\pi\epsilon$-periódica, las propiedades dinámicas de esta ecuación se visualizan de mejor forma con la ayuda del mapa de Poincaré asociado.

2.1. Definiciones y conceptos básicos

Definición 2.1.1. *Sea* $\boldsymbol{X}$*:* $U \to \mathbb{R}^n$ *un campo vectorial de clase* $\mathbb{C}^r$*,* $r \geq 1$ *y sea* ϕ *el flujo asociado a* $\boldsymbol{X}$*. Decimos que* $\gamma(x_0)$ *es una órbita periódica de período* $\boldsymbol{T} > 0$ *de* $\boldsymbol{X}$ *si* $\phi(\boldsymbol{T}, x_0) = x_0$ *y* $\phi(t, x_0) \neq x_0 \ \forall t \in (0, \boldsymbol{T})$

Definición 2.1.2. *Sea* $r \geq 1$ *o* $r = \omega$*. Sea* $\boldsymbol{X}$*:*$U \subset \mathbb{R}^n \to \mathbb{R}^n$ *un campo vectorial de clase* $\mathbb{C}^r$*, sea* $\mathbb{A} \subset \mathbb{R}^{n-1}$*.Una aplicación* $f : \mathbb{A} \to U$ *de clase* $\mathbb{C}^r$ *se llama* ***sección transversal(local) de*** $\boldsymbol{X}$ *si para todo* $a \in \mathbb{A}$*:*

$$\det(Df(a)\boldsymbol{X}(f(a))) \neq 0$$

Decimos que $\sum := f(\mathbb{A})$ *es una* ***sección transversal de*** $\boldsymbol{X}$

Definición 2.1.3. *Sea* $\boldsymbol{X}$*:* $U \to \mathbb{R}^n$ *un campo vectorial de clase* $\mathbb{C}^r$*,* $r \geq 1$ *y sea* $\gamma(x_0)$ *una órbita periódica de período* $\boldsymbol{T} > 0$*. Sea* $\sum$ *una sección transversal a* $\boldsymbol{X}$ *en el punto* x_0*. La aplicación*

$$\pi : \sum_0 \subset \sum \to \sum$$

definida como el primer punto en tiempo positivo de la órbita de x *que corta con* $\sum$ *se llama* ***Aplicación de Poincaré***

De forma usual y como se usará a partir de ahora en este trabajo, consideraremos el mapa de Poincaré de esta forma: si $z = (x_0, \dot{x}_0)$ es un punto del plano y $x(t)$ es la solución del sistema del péndulo rápidamente forzado con valor inicial: $x(0) = x_0, \dot{x}(0) = \dot{x}_0$, entonces $\pi(z)$ se define simplemente como $(x(2\pi\epsilon), \dot{x}(2\pi\epsilon))$.

Si $\mu = 0$ el retrato de fase de $\pi(z)$ es conocido. Podemos considerar x definida mod 2π y entonces su retrato de fase es un cilindro, lleno de curvas integrales del péndulo simple, que consisten en órbitas cerradas, excepto por un punto fijo elíptico en 0 $mod(2\pi\epsilon)$, un punto hiperbólico en $-\pi$ $mod(2\pi\epsilon)$ y dos separatrices asociadas al punto hiperbólico dadas por las soluciones homoclínicas del péndulo simple:

$$L_\pm = \{(x_0(t), y_0(t))\},$$

$$x_0(t) = 2\arctan(\sinh t),$$

$$y_0(t) = \dot{x}_0(t) = \frac{2}{\cosh t}$$

El retrato de fase se complica más si $\mu \neq 0$, sobre todo cerca de las separatrices. Si μ es lo suficientemente pequeño, todavía hay un punto fijo e hiperbólico cerca de $(-\pi, 0)$, el punto correspondiente a la órbita hiperbólica $2\pi\epsilon$-periódica, así como a las variedades estable e inestable que se encuentran cercanas a las dos separatrices.

2.1.1. Sistemas no lineales

Definición 2.1.4. *Sea* $S \subset \mathbb{R}^n$ *un conjunto, entonces*

1. *(Tiempo continuo)* S *es llamado* ***invariante*** *bajo el campo vectorial* $\dot{x} = f(x)$ *si para algún* $x_0 \in S$ *tenemos que* $x(t, 0, x_0) \in S$ *para todo* $t \in \mathbb{R}$*. Donde* $x(0, 0, x_0) = x_0$*.*

2. *(Tiempo discreto)* S *es llamado invariante bajo el mapa* $x \to g(x)$ *si para algún* $x_0 \in S$ *tenemos que* $g^{(n)}(x_0) \in S$ *para todo* n*.*

Si nos restringimos únicamente al tiempo positivo (ie. $t \geq 0, n \geq 0$*) entonces nos referimos a* S *como conjunto positivamente invariante y para el tiempo negativo, como conjunto negativamente invariante.*

Observación 2.1.5. Observamos que los conjuntos invariantes tienen la propiedad de que las trayectorias que comienzan con el conjunto invariante permanecen en él para todo su futuro y todo su pasado.

Cabe destacar que si g no es invertible, entonces solo tiene sentido considerar $n \geq 0$(aunque en algunos casos puede ser útil considerar g^{-1}, que sí tiene un significado en teoría de conjuntos).

Definición 2.1.6. *Un conjunto invariante $S \subset \mathbb{R}^n$ es llamado como una* ***variedad invariante de clase*** $\mathbb{C}^r$ *si S tiene la estructura de una variedad diferenciable de clase $\mathbb{C}^r$ $(r \geq 0)$. Similarmente, un conjunto positivamente(resp. negativamente) invariante $S \subset \mathbb{R}^n$ es llamado variedad invariante $\mathbb{C}^r$ $(r \geq 0)$ positiva (resp. negativa) si S tiene la estructura de una variedad diferenciable de clase $\mathbb{C}^r$*

Ejemplo 2.1.7. Sea $\{s_1, .., s_n\}$ la base estándar en $\mathbb{R}^n$. Sean $\{s_{i_1}, ..., s_{i_j}\}$ $j < n$, cualesquiera j vectores base de este conjunto; entonces, el espacio generado por $\{s_{i_1}, ..., s_{i_j}\}$ forma un subespacio de dimensión j de $\mathbb{R}^n$, que es, trivialmente , una variedad $\mathbb{C}^\infty$ de dimensión j. Para una introducción detallada a la teoría de variedades con una perspectiva orientada a aplicaciones, véase Abraham, Marsden y Ratiu [1988].

La razón principal para elegir estos ejemplos es que, en este trabajo, cuando se use el término "variedad", será suficiente pensar en una de estas dos situaciones:

1. Sistema lineal: un vector lineal subespacio de $\mathbb{R}^n$.
2. Sistema no lineal: una superficie incrustada en $\mathbb{R}^n$ que puede ser localmente representada mediante un gráfico (que puede ser justificada mediante el teorema de la función implícita).

Por otra parte, poco podemos decir de sistemas no lineales del tipo $\dot{x} = f(x)$. Recordando el teorema de existencia y unicidad de las soluciones para las ecuaciones diferenciales ordinarias, implica que para las funciones *analíticas* $f(x)$, la solución del problema con valor inicial

$$\dot{x} = f(x); \quad x \in \mathbb{R}^n, \quad x(0) = x_0$$

está definida al menos localmente en algún entorno $t \in (-\omega, \omega)$ de $t = 0$. Entonces, un flujo local $\phi_t : \mathbb{R}^n \to \mathbb{R}^n$ está definido como $\phi_t = x(t, x_0)$ de una manera análoga al caso lineal, aunque no podamos dar una fórmula general del tipo $\exp(At)$.

Un buen lugar para empezar el estudio de un sistema no lineal $\dot{x} = f(x)$ es encontrando los ceros de f o los puntos fijos del sistema. Supongamos que tenemos un punto fijo $\overline{x}$ tal que $f(\overline{x}) = 0$ y queremos entender el comportamiento de las soluciones en un entorno cercano a $\overline{x}$. Hacemos esto mediante la linearización del sistema en $\overline{x}$, es decir, estudiando el sistema:

$$\dot{y} = Df(\overline{x})y, \quad y \in \mathbb{R}^n$$

donde $Df = [\frac{\partial f_i}{\partial x_i}]$ es la matriz jacobiana de las primeras derivadas parciales de la función $f = (f_1(x_1, ..., x_n), f_2(x_1, ..., x_n), ..., f_n(x_1, ..., x_n))^T$ y $x = \overline{x} + \epsilon \quad \|\epsilon\| \ll 1$. Como el sistema linealizado es un sistema lineal, esto se puede hacer de forma sencilla. En particular, el mapa del flujo linealizado $D\phi_t(\overline{x})y$ se obtiene mediante:

$$D\phi_t(\overline{x})y = e^{tDf(\overline{x})}y$$

Finalmente todo se reduce a que podemos decir del sistema a partir de lo que conocemos de su correspondiente sistema linealizado.

Teorema 2.1.8. *Si $Df(\overline{x})$ no tiene ningún cero o únicamente valores propios imaginarios, entonces existe un homeomorfismo h definido en un entorno U de $\overline{x}$ en $\mathbb{R}^n$ local que convierte órbitas del flujo no lineal ϕ_t del sistema a órbitas del flujo linealizado $e^{tDf(\overline{x})}$ del sistema linealizado. Este homeomorfismo preserva el sentido de las órbitas y puede ser escogido de tal forma que preserve también su parametrización por el tiempo.*

Cuando $Df(\overline{x})$ no tiene valores propios con la parte real igual a 0, $\overline{x}$ se llama un punto fijo **hiperbólico o no degenerado** y el comportamiento asintótico de las soluciones alrededor del punto(y por lo tanto su estabilidad) se puede conocer mediante la linearización.

Definición 2.1.9. *Definimos la variedad local estable e inestable de* $\overline{x}$, $W^s_{loc}(\overline{x}), W^u_{loc}(\overline{x})$ *de la siguiente forma:*

$$W^s_{loc}(\overline{x}) = \{x \in U \text{ tal que } \phi_r(x) \to \overline{x} \text{ cuando } t \to \infty \text{ y } \phi_t(x) \in U \ \forall\, t \geq 0\}$$

$$W^u_{loc}(\overline{x}) = \{x \in U \text{ tal que } \phi_r(x) \to \overline{x} \text{ cuando } t \to -\infty \text{ y } \phi_t(x) \in U \ \forall\, t \leq 0\}$$

donde $U \subset \mathbb{R}^n$ *es un entorno del punto fijo* $\overline{x}$.

Observación 2.1.10. Las variedades invariantes $W^s_{loc}(\overline{x}), W^u_{loc}(\overline{x})$, proveen de forma análoga a los planos (espacios vectoriales) estables e inestables $\mathbf{E}^s, \mathbf{E}^u$ de un problema de tipo lineal.

Teorema 2.1.11. *Supongamos que* $\dot{x} = f(x)$ *tiene un punto hiperbólico en* $\overline{x}$. *Entonces existe una variedad local estable y una variedad local inestable* $W^s_{loc}(\overline{x}), W^u_{loc}(\overline{x})$ *de la misma dimensión* n_s, n_u *que los espacios vectoriales* $\boldsymbol{E}^s, \boldsymbol{E}^u$ *del sistema linealizado respectivo, y tangentes a* $\boldsymbol{E}^s, \boldsymbol{E}^u$ *en* $\overline{x}$, $W^s_{loc}(\overline{x}), W^u_{loc}(\overline{x})$ *son tan suaves como la función* f.

Estas variedades locales invariantes $W^s_{loc}(\overline{x}), W^u_{loc}(\overline{x})$ tienen sus análogas globales obtenidas mediante *propagar* puntos en $W^s_{loc}(\overline{x})$ hacia atrás en el tiempo, y *propagar* puntos en $W^u_{loc}(\overline{x})$ hacia adelante en el tiempo.

Definición 2.1.12. *Definimos las variedades invariantes globales de la siguiente forma:*

$$W^s(\overline{x}) := \bigcup_{t \leq 0} \phi_t(W^s_{loc}(\overline{x})),$$

$$W^u(\overline{x}) := \bigcup_{t \geq 0} \phi_t(W^u_{loc}(\overline{x})),$$

En este trabajo consideramos que una variedad es un conjunto que, localmente, tiene la estructura del espacio euclidiano. En las aplicaciones, las variedades se encuentran más comúnmente como superficies de dimensión m incrustadas en $\mathbb{R}^n$. Si la superficie no tiene puntos singulares, es decir, si la derivada de la función que representa la superficie tiene rango máximo, entonces, según el teorema de la función implícita, puede representarse localmente como un gráfico. La superficie es una variedad de clase $\mathbb{C}^r$ si los gráficos (locales) que la representan son de clase $\mathbb{C}^r$ (nota: para un tratamiento detallado de esta representación particular de una variedad, véase Dubrovin, Fomenko y Novikov [1985]).

En el sistema del péndulo rápidamente forzado, las variedades estable e inestable intersecan en el punto conocido como **punto homoclínico**. A causa de la forma de la ecuación del péndulo rápidamente forzado, tienen que intersecar en un punto z_h en el eje $\dot{x}$ así como en la órbita $\{\pi^n(z_h),\ n \in \mathbb{Z}\}$. Llamemos α a este ángulo.

Si $\alpha \neq 0$ las variedades invariantes intersecan de forma transversal en el punto z_h y encierran bucles cuya área es un invariante, es decir, no depende del punto z_h elegido.

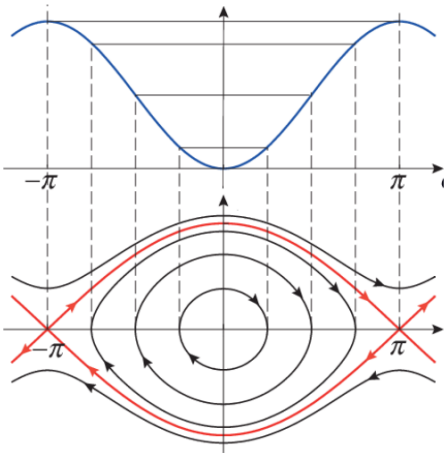

Figura 2.1: Retrato de fase del péndulo simple.Fuente: web.mat.upc.edu

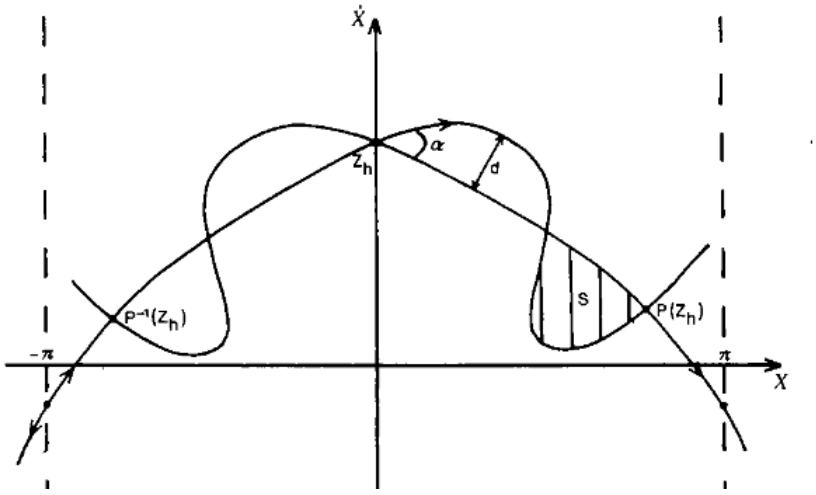

Figura 2.2: Retrato de fase del péndulo rápidamente forzado con el punto homoclínico y el ángulo de escisión.Fuente: [4]

Capítulo 3

Aritmética de alta precisión

Los ordenadores permiten representar los datos con elementos básicos que únicamente pueden estar en dos estados. La información que se puede representar con combinaciones de estos elementos básicos es información binaria. Cada elemento de información binaria se llama *bit* y se codifica usando los símbolos 0 y 1. La mayoría de definiciones y resultados de esta sección se encuentran en [7], [8], [9] y [10].

3.1. Definiciones y conceptos básicos

Para representar un número se pueden utilizar distinto número de dígitos según la base en la que se esté representando ese número.

Ejemplo 3.1.1. Por ejemplo, en el sistema hexadecimal(de base 16) se usan los dígitos $0, ..., 9, A, ..., F$ que representan las cantidades $zero, ..., nueve, diez, ..., quince$ respectivamente.

3.1.1. Representación de reales en punto fijo

Dada una base $b \geq 2$, todo número real x se puede representar en la forma

$$\pm d_n d_{n-1} .... d_1 d_0 . d_{-1} d_{-2} ...{}_{(b}$$

donde $d_j \in \mathbb{N}$ con $0 \leq d_j < b \quad (j = n, n-1, ...)$ La expresión decimal de este número se obtiene sumando los valores de los dígitos teniendo en cuenta su posición:

$$d_n b^n + d_{n-1} b^{n-1} + ... + d_1 b^1 + d_0 b^0 + d_{-1} b^{-1} + d_{-2} b^{-2} + ...$$

Ejemplo 3.1.2.

$$1101{,}11_{(2} = 1{\cdot}2^3 + 1{\cdot}2^2 + 0{\cdot}2 + 1{\cdot}2^0 + 1{\cdot}2^{-1} + 1{\cdot}2^{-2} = 13{,}75$$

Para obtener la expresión de un número en base b, separamos la parte entera y la parte decimal. La expresión de la parte decimal se obtiene haciendo productos sucesivos por b y cogiendo la parte entera de estos productos.

Ejemplo 3.1.3. Para obtener la expresión 23,15 en binario, separamos la parte entera 23 y la parte decimal 0,15. Hemos visto que $23 = 10111_{(2}$. Para la parte decimal calculamos

$$0{,}15{\cdot}2 = 0{,}3 \qquad 0{,}4{\cdot}2 = 0{,}8$$
$$0{,}3{\cdot}2 = 0{,}6 \qquad 0{,}8{\cdot}2 = 1{,}6$$
$$0{,}6{\cdot}2 = 1{,}2 \qquad 0{,}6{\cdot}2 = 1{,}2$$
$$0{,}2{\cdot}2 = 0{,}4 \qquad 0{,}2{\cdot}2 = ...$$

es decir $0{,}15 = 0{,}0010011..._{(2}$ y por tanto $23{,}15 = 10111{,}0010011..._{(2}$

3.1.2. Representación de reales en punto flotante

Dada una base $b \geq 2$, una representación en punto flotante de un número real $x \neq 0$ viene dada por

$$x = (-1)^s M b^E$$

donde E es un número entero, $s \in \{0, 1\}$, y

$$M = D_0 D_1 D_2 D_3 ..._{(b}$$

con $D_j \in \mathbb{N}$ y $0 \leq D_j \leq b$ para todo j.

s representa el signo, E es el exponente y M, la mantisa que puede tener infinitos dígitos

Ejemplo 3.1.4. La representación en punto flotante normalizada de 23,15 es $2{,}315{\cdot}10^1$ en base 10 y $1{,}01110010011..._{(2}{\cdot}2^4$ en binario

3.2. Codificación de números reales en punto flotante

Si se dispone de n bits para codificar el número, hemos de separarlo en tres partes: el signo,el exponente y la mantisa. Cada sistema de punto flotante fija la longitud de estas tres partes.

Observación 3.2.1. El número de dígitos de la mantisa es la precisión.

Un sistema de números en punto flotante de base b, precisión $p + l$ y valores del exponente en $[E_{min}, E_{max}] \in \mathbb{Z}$ contiene los números reales:

$$(-1)^s m b^E$$

donde $s \in \{0, 1\},\ E \in \{E_{min}, E_{min} + 1, ..., E_{max}\},\ m = d_0 d_1 d_2 d_3 .. d_p\ d_j \in \{0, 1, ..., b - 1\},\ j = 0, ..., p$

Para que la codificación sea única sin reducir el rango del sistema, se añade la condición $d_0 \neq 0$ que provoca una biyectividad entre los posibles valores y los números normalizados.

Los números perdidos a causa de la normalización se pueden recuperar sin destruir la unicidad de la representación haciendo $d_0 = 0$ solo para el valor mínimo del exponente, a estos se les llama números *no normalizados*.

Si solo se disponen de $p + 1$ dígitos y el valor real $x = (-1)^s M b^E$ tiene una mantisa $M = d_0 D_1 D_2 D_3 ..._{(b}$ con más dígitos, no podremos guardar el número exactamente.

El paso del valor real de x a su aproximación se hace mediante el proceso:

1. Suprimiendo los dígitos a partir de D_{p+1}.
2. Se redondea, codificando en función del dígito D_{p+1}:

$$\begin{cases} \text{si } D_{p+1} & \leq \frac{1}{2}b, \text{ tomando } m = d_0D_1D_2D_3...D_p \\ \text{si } D_{p+1} & \geq \frac{1}{2}b, \text{ tomando } m = d_0D_1D_2D_3...D_p + b^{-p} \end{cases}$$

Ejemplo 3.2.2. En la codificación de π con $p = 9$, tomaremos los valores

$$3{,}141592653 \cdot 10^0 \text{si cortamos}$$

$$3{,}141592654 \cdot 10^0 \text{ si redondeamos}$$

3.3. Estándar IEEE 754

La implementación del estándar IEEE 754 utiliza la base 2 y tiene dos formatos básicos: precisión simple ($float$) y precisión doble ($double$). El exponente E se guarda sesgado, en la forma $e = E + sesgo$.

La precisión simple usa 4 bytes, de los cuales el primer bit de la izquierda es para el signo, los siguientes 8 bits son para el exponente sesgado y los últimos 23 para la parte decimal de la mantisa. Se coge como sesgo $2^{8-1} - 1 = 127$.

La precisión doble usa 8 bytes, de los cuales el primer bit es para el signo, los siguientes 11 para el exponente sesgado y los otros 52 para la parte decimal de la mantisa. Se coge como sesgo $2^{11-1} - 1 = 1023$.

Esta implementación usa un bit oculto, ya que los números normalizados tienen $d_0 \neq 0$ y por lo tanto $d_0 = 1$.

Ejemplo 3.3.1. Codificamos con precisión simple el número 10,2. Tenemos que

$$x = 10{,}2 = 1{,}275 \cdot 2^3 = 1{,}010\overline{0011}_{(2} \cdot 2^3$$

El número es positivo, por lo tanto $s = 0$; el exponente $E = 3$, por lo tanto $e = 3 + 127 = 130 = 10000010_{(2}$. Por lo tanto su codificación es:

$$01000001001000110011001100110011$$

Observamos que el número exacto que representa esta codificación es 10,19999981

3.4. MPFR

La biblioteca MPFR es una biblioteca para aritmética de punto flotante de precisión múltiple con semántica clara, que extiende el estándar IEEE 754. Esta biblioteca está escrita en lenguaje C sobre la biblioteca GNU MP(GMP).

MPFR proporciona redondeo correcto para todas las operaciones y funciones matemáticas que implementa, con una eficiencia más rápida que otros softwares.

Como consecuencia, los programas que usan esta biblioteca heredan las mismas propiedades destacables de los programas que utilizan IEEE 754: portabilidad, semántica bien definida, posibilidad de diseñar programas robustos y probar su corrección, sin una desaceleración con respecto a bibliotecas de precisión múltiple con peor semántica.

3.4.1. Funcionamiento

La biblioteca MPFR es una extensión del estándar IEEE 754, con base 2. Esta elección de base 2 se fundamenta en buscar la eficiencia y la portabilidad.

Respecto a la eficiencia se usa la capa *GMP mpn* que requiere una base del tipo 2^k. Por otra parte, en aras de la portabilidad, si se tomase un $k \neq 1$ las mantisas de punto flotante que usan un número impar de palabras en una máquina de 32 bits, no tendrían equivalente en un ordenador de 64 bits y tampoco sería posible emular los formatos IEEE 754 de 24 y 53 bits.

La idea principal que aporta la biblioteca MPFR es que cualquier número en punto flotante tiene su propia precisión en bits, la cual puede variar desde 2 hasta la mayor precisión posible con la memoria disponible.

Consideremos un número flotante x con precisión p, otro número y con precisión q, y un modo de redondeo r. Sea $mpfr\_f$ la función de la biblioteca correspondiente a una función matemática f. El resultado de $mpfr\_f(y, x, r)$ consiste en asignar el valor de $round(f(x), q, r)$ al número en punto flotante y, lo que significa que el resultado exacto de $f(x)$ se redondea a la precisión q.

Ejemplo 3.4.1. Sea $x = 601 \cdot 2^{10} = 0{,}1001011001_{(2}$ entonces el redondeo correcto de $exp(x)$ con redondeo al más cercano y una precisión objetivo de 17 bits es $58931 \cdot 2^{-15} = 1{,}1100110001100110_{(2}$

Como ocurre con cualquier operación aritmética que sigue el estándar IEEE 754, cada entrada de función es considerada como exacta por MPFR. En otras palabras, MPFR proporciona redondeo correcto solo para operaciones atómicas; no se conserva información sobre la "precisión" de los resultados intermedios. Por lo tanto, para cualquier secuencia de operaciones, es responsabilidad del usuario calcular los límites de error correspondientes. Este trabajo se simplifica debido a que cada operación atómica genera límites de error rigurosos.

Observación 3.4.2. Cada función de MPFR devuelve un valor ternario, llamado ïndicador de inexactitud", que indica la dirección del redondeo con respecto al valor exacto: el indicador de inexactitud es negativo (respectivamente positivo o cero) cuando la salida redondeada es menor (respectivamente mayor o igual) que el valor exacto.

3.4.2. Representación de datos

La representación interna de los datos utilizada por MPFR es la siguiente. Un número en punto flotante x se representa mediante una mantisa m, un signo s y un exponente firmado e. Los números especiales como NaN, infinitos o ceros tienen una representación especial.

La mantisa m se representa con una lista de "limbs"(partes) de GMP (entero-máquina sin signo) y se interpreta como $\frac{1}{2} \leq m < 1$. El bit más significativo de la mantisa siempre es 1. El bit más significativo de la mantisa corresponde al bit más significativo del "limb"más significativo. En otras palabras, cuando la precisión no es un múltiplo del número de bits por palabra, los bits no utilizados se encuentran en el "limb"menos significativo y siempre son cero.

Ejemplo 3.4.3. La mantisa del número de 17 bits $1{,}1100110001100110_{(2}$ se almacenaría en un ordenador de 5 bits de la siguiente forma:

limb 3: 11100 limb 2: 1100 limb 1: 11001 limb 0: 10000

3.4.3. MPFR C++

MPFR define un tipo de datos personalizado en C para representar números de punto flotante: *mpfr_t*. Las manipulaciones matemáticas con variables de tipo *mpfr_t* se realizan mediante funciones que se asemejan a instrucciones ensamblador.

Por ejemplo, para sumar dos números x e y con el resultado en z se debe llamar a la función especial $mpfr\_add(z, x, y, GMP\_RNDN)$.

Ejemplo 3.4.4. Consideremos esta función escrita en aritmética doble y aritmética mpfr:

```
// precision double
double f(double x)
{
    return  1-x*sin(sqrt(abs(x)));
}

//precision mpfr
void mpfr_f(mpfr_t y, mpfr_t x)
{
   mpfr_t t;
   mpfr_init(t);
mpfr_abs(t,x,GMP_RNDN);
mpfr_sqrt(t,t,GMP_RNDN);
mpfr_sin(t,t,GMP_RNDN);
mpfr_mul(t,t,x,GMP_RNDN);
mpfr_set_str(y,'1',10,GMP_RNDN);
mpfr_sub(y,y,t,GMP_RNDN);
   mpfr_clear(t);
}
```

Tras ver este ejemplo, se puede entender de forma clara que se haya desarrollado un wrapper que simplifique la sintaxis de escritura utilizando MPFR.

El wrapper en C++ para MPFR introduce un nuevo tipo en C++ para números de punto flotante de alta precisión: *mpreal* que encapsula el nivel bajo de *mpfr_t*.

Todos los operadores aritméticos y booleanos (+, -, *, /, ¿, !=, etc.) están implementados para números *mpreal* mediante la técnica de sobrecarga de operadores. Las funciones matemáticas (sqrt, pow, sin, cos, etc.) también son compatibles. Esto hace posible usar cálculos de MPFR de manera tan sencilla como los cálculos con números de los tipos integrados *double* o *float*.

Cálculos internos

Algo que distingue este wrapper de otras bibliotecas que extienden MPFR es la forma de tratar los resultados intermedios durante el cálculo de una expresión matemática.

C++ descompone las expresiones en operaciones atómicas como + o · y almacena los resultados en variables temporales para su posterior uso.

Ejemplo 3.4.5. La expresión $x = ab + cd$ se calcula en C++ de la siguiente forma:

```
t1 = a * b
t2 = c * d
t3 = t1 + t2
x = t3
```

donde $t1, t2, t3$ son variables temporales que están ocultas al usuario.

Para obtener un valor final correcto, estos resultados intermedios deben tratarse cuidadosamente con la suficiente precisión.

La regla principal de MPFR C++ para los resultados intermedios es:

$$prec(resultado) = max(prec(op1), prec(op2))$$

El resultado de una operación se almacena con la máxima precisión entre las precisiones de los operandos. En otras palabras, las operaciones intermedias se realizan con la máxima precisión razonable definida por las precisiones de los argumentos y no dependen de la precisión de la variable objetivo (en este ejemplo, x)

Otras bibliotecas calculan los resultados intermedios con la precisión de la variable final o los redondean a una precisión no evidente para el usuario, que es independiente de la precisión de los argumentos y de la variable final. Esto puede conducir a una reducción significativa en la precisión del resultado final [11].

Capítulo 4

Método de Taylor

En esta sección comenzaremos explicando cómo funciona el método de Taylor, un paquete de software para la integración numérica de EDOs mediante métodos de Taylor de orden alto. La mayoría de definiciones y resultados de este capítulo se encuentran en [12], [13] y [14].

4.1. Definiciones y conceptos básicos

El objetivo principal de este paquete de software es generar un integrador numérico específico para un conjunto dado de EDOs. El código generado incluye una función para calcular el flujo de derivadas de la solución hasta un orden determinado, además de la sección adaptativa del orden y el tamaño del paso para el problema de valor inicial. Este paquete ofrece soporte para distintas aritméticas de gran precisión, incluyendo tipos definidos por el propio usuario.

Si consideramos el problema de valor inicial

$$\begin{cases} x'(t) = f(t, x(t)), \\ x(a) = x_0 \end{cases}$$

donde asumimos que f es analítica en su dominio definido y que $x(t)$ está definida para todo $t \in [a, b]$.

La idea del método de Taylor es muy simple, dada la condición inicial $x_{m+1} = x_0$ $(t_0 = a)$, el valor de $x(t_0 + h)$ es aproximado mediante las series de Taylor de la solución $x(t)$ en $t = t_0$,

$$\begin{aligned} x_0 &= x(t = 0), \\ x_{m+1} &= x_m + x'(t_m)h + \frac{x''(t_m)}{2!}h^2 + ... + \frac{x^p(t_m)}{p!}h^p, \quad m = 0, ..., M - 1, \end{aligned}$$

donde $t_m = a + mh$ y $h = (b - a)/M$

Por lo tanto, el primer paso para aplicar este método es calcular estas derivadas hasta un orden adecuado. Entonces, para cada paso de integración, evaluar estas expresiones para obtener los coeficientes de las series $x(t)$ en $t = t_m$.

Para resolver este problema se utiliza la técnica llamada *diferenciación automática*

4.2. Diferenciación Automática

Definición 4.2.1. *La diferenciación automática es un procedimiento recursivo que calcula el valor de las derivadas de ciertas funciones en un punto dado.*

Las funciones consideradas son aquellas que pueden ser obtenidas mediante sumas, productos, cocientes y composiciones de funciones elementales (se incluyen polinomios, funciones trigonométricas, potencias, exponenciales y logaritmos).

En pos de simplificar la explicación vamos a introducir cierta notación: si $f : t \in I \subset \mathbb{R} \to \mathbb{R}$ denota una función continuamente diferenciable, llamamos a su n-ésima derivada normalizada

$$f^{[n]}(t) = \frac{1}{n!}\, f^{(n)}(t) \tag{4.2.1}$$

donde $f^{(n)}(t)$ denota la derivada n-ésima de f respecto de t

Asumimos que $f(t) = F(b(t), c(t))$ y que sabemos los valores de $b^{[j]}(t)$ y de $c^{[j]}(t)$, $j = 0, ..., n$, para un t dado.

La siguiente proposición da la n-ésima derivada de f en t para algunas funciones F.

Proposición 4.2.2. *Si las funciones b y c son de clase $\mathbb{C}^n$, y $\alpha \in \mathbb{R} \setminus 0$, entonces tenemos:*

1. *Si $f(t) = b(t) \pm c(t)$, entonces*

$$f^{[n]}(t) = b^{[n]}(t) \pm c^{[n]}(t)$$

2. *Si $f(t) = b(t)c(t)$,entonces*

$$f^{[n]}(t) = \sum_{j=0}^{n} b^{[n-j]}(t)c^{[j]}(t)$$

3. *Si $f(t) = \frac{b(t)}{c(t)}$, entonces*

$$f^{[n]}(t) = \frac{1}{c^{[0]}(t)}[\, b^{[n]}(t) - \sum_{j=1}^{n} c^{[j]}(t)f^{[n-j]}(t)\,]$$

4. *Si $f(t) = b(t)^{\alpha}$, entonces*

$$f^{[n]}(t) = \frac{1}{nb^{[0]}(t)} \sum_{j=0}^{n-1} (n\alpha - j(\alpha + 1))b^{[n-j]}(t)f^{[j]}(t)$$

5. *Si $f(t) = e^{b(t)}$, entonces*

$$f^{[n]}(t) = \frac{1}{n} \sum_{j=0}^{n-1} (n - j)f^{[j]}(t)b^{[n-j]}(t)$$

6. *Si $f(t) = \ln b(t)$,entonces*

$$f^{[n]}(t) = \frac{1}{b^{[0]}(t)}[b^{[n]}(t) - \frac{1}{n} \sum_{j=1}^{n-1} (n - j)b^{[j]}(t)f^{[n-j]}(t)]$$

7. *Si* $f(t) = \cos c(t)$ *y* $b(t) = \sin c(t)$, *entonces*

$$f^{[n]}(t) = \frac{-1}{n} \sum_{j=1}^{n} j b^{[n-j]}(t) c^{[j]}(t)$$

$$b^{[n]}(t) = \frac{1}{n} \sum_{j=1}^{n} j f^{[n-j]}(t) c^{[j]}(t)$$

Corolario 4.2.3. *El número de operaciones aritméticas necesarias para evaluar las derivadas normalizadas de una función hasta el orden n es* $O(n^2)$.

Observación 4.2.4. Estas reglas se aplican de forma recursiva para conseguir fórmulas recursivas para las derivadas de una función descrita por combinaciones de estas funciones básicas.

4.3. Grado y control de paso

La expansión en potencias de la solución $x(t)$ en $t = t_m$ tendrá radios de convergencia distintos para diferentes t_m, esto causa que, en cada paso (es decir, para cada m), se deba calcular valores adecuados para el orden $p = p_m$ y el tamaño de paso $h = h_m$.

Denotemos por $\{x_m^{[j]}(t_m)\}_j$ el flujo de derivadas normalizadas en t_m de la solución del problema de valor inicial que satisface $x_m(t_m) = x_m$. Entonces, si $h = t - t_n$ es suficientemente pequeño, tenemos

$$x_m(t) = \sum_{j=0}^{\infty} x_m^{[j]}(t_m) h_m^j \tag{4.3.1}$$

Por lo tanto, hemos de seleccionar un valor de h_m suficientemente pequeño y un valor de p_m suficientemente grande tal que

$$t_{m+1} \equiv t_m + h_m \ , \quad x_{m+1} \equiv \sum_{j=0} p_m x_m^{[j]}(t_m) h_m^j$$

cumplan que

$$\|x_m(t_{m+1} - x_{m+1})\| \leq \epsilon$$

Proposición 4.3.1. *Asumiendo que la función* $z \to x(t_m + z)$ *es analítica en el disco de radio* p_m. *Sea* A_m *una constante positiva tal que*

$$\|x_m^j\| \leq \frac{A_m}{p_m^j}, \quad \forall j \in \mathbb{N}. \tag{4.3.2}$$

Entonces si la precisión requerida ϵ *tiende a* 0, *los valores de* h_m *y* p_m *que dan la precisión requerida y minimizan el número de operaciones tienden a*

$$h_m = \frac{p_m}{e^2}, \quad p_m = -\frac{1}{2} \ln \frac{\epsilon}{A_m} - 1$$

Observación 4.3.2. Se implementa en el método de Taylor un algoritmo basado en esta proposición.

4.4. Estimación del Orden y del paso

Para implementar esta proposición se necesita el radio de convergencia de la serie de Taylor o el valor de A_m.

Denotamos por ϵ_a y ϵ_r las tolerancias absolutas y relativas para el error. Si $\epsilon_r\|xM\|_\infty \leq \epsilon_a$ controlamos el error absoluto usando ϵ_a.

En primer lugar calculamos el orden de p_m para el método de Taylor de la siguiente forma: se define ϵ_m como

$$\epsilon_m = \begin{cases} \epsilon_a & \text{si } \epsilon_r\|x_m\|_\infty \leq \epsilon_a, \\ \epsilon_r & \text{de lo contrario.} \end{cases} \tag{4.4.1}$$

y entonces,

$$p_m = [-\frac{1}{2}\ln\epsilon_m + 1] \tag{4.4.2}$$

donde [x] es la función que redondea un número x al entero más pequeño tal que sea mayor o igual a x.

Comparando con la proposición anterior, es como si consideráramos $A_m = 1$ y p_m dos unidades mayores.

Para derivar el paso, consideramos los dos mismos casos. Si $\epsilon_r\|x_m\|_\infty \leq \epsilon_a$ definimos

$$\delta_m^{(j)} = (\frac{1}{\|x_m^{[j]}\|_\infty})^{\frac{1}{j}}, \quad 1 \leq j \leq p, \tag{4.4.3}$$

y si $\epsilon_a < \epsilon_r\|x_m\|_\infty$,

$$\delta_m^{(j)} = (\frac{\|x_m\|_\infty}{\|x_m^{[j]}\|_\infty})^{\frac{1}{j}}, \quad 1 \leq j \leq p \tag{4.4.4}$$

Por lo tanto, en los dos casos estimamos que el radio de convergencia es el mínimo de los dos últimos términos

$$\delta_m = \text{mín}\,(\delta_m^{(p-1)}, \delta_m^{(p)}) \tag{4.4.5}$$

y el estimado paso es

$$h_m = \frac{\delta_m}{e^2} \tag{4.4.6}$$

Para los casos en los que $A_m \neq 1$ necesitamos acotar el error de truncamiento correspondiente al orden p_m y al paso h_m; para ello, usaremos la siguiente proposición.

Proposición 4.4.1. *1. Si $\epsilon_r\|x_m\|_\infty \leq \epsilon_a$, entonces*

$$\|x_m^{[p_m-1]}h_m^{p_m-1}\|_\infty \leq \epsilon_a, \quad \|x_m^{[p_m]}h_m^{p_m}\|_\infty \leq \frac{\epsilon_a}{e^2}$$

2. Si $\epsilon_r\|x_m\|_\infty > \epsilon_a$, entonces

$$\frac{\|x_m^{[p_m-1]}h_m^{p_m-1}\|_\infty}{\|x_m\|_\infty} \leq \epsilon_r, \quad \frac{\|x_m^{[p_m]}h_m^{p_m}\|_\infty}{\|x_m\|_\infty} \leq \frac{\epsilon_r}{e^2}$$

Observación 4.4.2. Nótese que aunque la fórmula para el orden utiliza $A_m = 1$, su valor real se tiene en cuenta en las fórmulas (4.4.3) y (4.4.4), por ello esta última proposición se cumple con $A_m \neq 1$.

Estos resultados se han utilizado para implementar dos controles de paso.

4.5. Controles de paso

4.5.1. Primer control de paso

Equivale a usar las fórmulas (4.4.1) y (4.4.2) para el orden y (4.4.3)(4.4.4)(4.4.5) para el radio de convergencia. Finalmente, se añade un factor (4.4.6) para asegurar el control de paso:

$$h_m = \frac{\delta_m}{e^2} \exp(-\frac{0{,}7}{p_m - 1})$$

4.5.2. Segundo control de paso

Es una corrección del anterior método que evita tamaños de paso excesivamente grandes, los cuales podrían llevar a cancelaciones al sumar la serie de Taylor.

Sea $\overline{h}_m$ el control de paso encontrado usando el primer método, definimos z como:

$$z = \begin{cases} 1 & \text{si } \epsilon_r \|x_m\|_\infty \leq \epsilon_a, \\ \|x_m\|_\infty & \text{de lo contrario.} \end{cases} \tag{4.5.1}$$

Sea $h_m \leq \overline{h}_m$ el valor más grande tal que:

$$\|x_m^{[j]}\|_\infty h_m^j \ \leq \ z, \quad j = 1, ..., p$$

4.6. Computación de alta precisión

Como hemos comentado al principio, este paquete de software tiene la propiedad de que es compatible con aritmética de alta precisión.

Asumiendo que estamos resolviendo el problema de valor inicial y que, en un paso, usamos un tamaño de paso $h \ll 1$ y un orden p para obtener un error $\epsilon \ll 1$. El número de operaciones necesarias para el cálculo de derivadas es $O(p^2)$(Corolario 4.2.3). Como el número de operaciones para la suma de la serie de potencias es $O(p)$, el número de operaciones totales para un paso es $O(p^2)$. Por ello, si quisiéramos incrementar la precisión hasta ϵ^l con $(2 \leq l)$ podríamos simplemente incrementar el orden del método de Taylor hasta lp, para que el número de operaciones se incremente por un factor l^2. Si quisiéramos incrementar la precisión sin incrementar el orden, reduciríamos el tamaño del paso h. Esta última opción significaría que el número total de operaciones se incrementaría por un factor de $\frac{1}{h^{l-1}}$, mayor que l^2.

En consecuencia, requiere bastante menos trabajo incrementar el orden que reducir el paso. Por ello, los métodos de orden fijo son fuertemente penalizados en cuanto se requiere alta precisión en comparación con los métodos de orden variable.

Por esta razón, hemos utilizado este método para calcular el ángulo entre las dos separatrices del sistema del péndulo rápidamente forzado, ya que se requiere una precisión muy alta al trabajar con dígitos muy pequeños, de órdenes de $O(10^{-60})$ o más según los valores de ϵ y de μ.

4.7. Implementación del software

Taylor actúa como traductor, lee un sistema de EDOs desde un archivo de texto plano y genera un conjunto de rutinas en C que implementan el método de Taylor para el sistema dado.

Taylor trabaja en distintas fases, donde en cada fase se genera una salida y se pasa como entrada a la próxima fase.

La primera fase es la fase léxica. Esta consiste en la traducción de los caracteres del archivo de entrada en unidades léxicas llamadas tokens, mediante el uso de escáneres (analista léxico). Este escáner está generado por *Lex*.

La siguiente fase es la fase sintáctica. En esta fase, un analizador agrupa los tokens en unidades sintácticas y verifica que la entrada sea sintácticamente válida de acuerdo con un conjunto de reglas de gramática libre de contexto. La salida del analizador es el árbol sintáctico, una representación gráfica de la entrada. Este analizador está generado por Yacc.

La siguiente fase es la optimización. En esta fase, el árbol sintáctico se analiza y modifica utilizando transformaciones que preservan la semántica. En esta fase se llevan a cabo tareas como:

1. La identificación y el marcaje de expresiones constantes. Las expresiones constantes son triviales de manejar al calcular derivadas de alto orden.

2. La eliminación de subexpresiones comunes. En este caso, la tarea de buscar subexpresiones comunes se realiza casi por completo a nivel "léxico". En este paso, recorremos la rama derecha del árbol sintáctico para cada ecuación diferencial y anotamos los nodos con sus subexpresiones definitorias. Para los nodos no terminales, se introducen variables temporales cuyas expresiones definitorias se registran como sus atributos. Luego, el conjunto de variables temporales se compara por pares utilizando sus expresiones definitorias. Si se encuentra que dos variables son iguales, una de ellas se elimina. Este proceso continúa hasta que no se encuentren variables redundantes.

3. La introducción de variables auxiliares para algunas funciones elementales. Por ejemplo, se agrega una nueva variable $v = \cos x$ a la tabla de símbolos si $\sin x$ aparece en el árbol sintáctico.

4. La construcción de gráficos de dependencia entre todas las variables y se ordenan las variables según el gráfico de dependencias.

5. La expansión de las potencias como series de productos.

La última fase consiste en la generación de código. Se aplican las fórmulas de la proposición 4.2.2 al código producido por el optimizador, con el fin de generar procedimientos que calculen los coeficientes de Taylor de forma secuencial.

Capítulo 5

Cálculo del ángulo de escisión

En esta sección explicaremos la parte principal de este Trabajo de Fin de Grado: la creación del programa que calcula con alta precisión la medida del ángulo entre las dos separatrices del péndulo rápidamente forzado.

Explicaremos la lógica de todo el programa, junto con las funciones utilizadas en todo momento.

Todo ello, acompañando la explicación del razonamiento matemático que fundamenta el flujo de trabajo del programa.

5.1. Cálculo órbitas periódicas

Todo nuestro programa parte de la definición de la ecuación del péndulo rápidamente forzado,

$$\ddot{x} + \sin x = \mu \ \sin\left(\frac{t}{\epsilon}\right) \tag{5.1.1}$$

Podemos observar que esta ecuación se puede reescribir a

$$\ddot{x} + \sin x = f(x,t)$$

donde $f(x,t)$ es una función periódica en t ya que incluye $\sin(\frac{t}{\epsilon})$.

Por ello podemos calcular su **T-periodo**.

Observación 5.1.1. El período de una función senoidal depende del argumento del seno:

- Si el argumento es $\omega t = \frac{t}{\epsilon}$, la frecuencia angular es $\omega = \frac{1}{\epsilon}$
- Sustituyendo $\omega = \frac{1}{\epsilon}$, el T-período es: $T = 2\pi\epsilon$

Esto significa que $f(x,t)$ se repite cada $T = 2\pi\epsilon$ unidades de tiempo, independientemente de x.

En este momento seguimos un razonamiento en términos de la **definición de una función de retorno o mapa de Poincaré** y el uso del método de Newton para encontrar soluciones periódicas.

Definimos $F : \mathbb{R}^n \to \mathbb{R}^n$, correspondiente al flujo del sistema, evaluado después del periodo $T = 2\pi\epsilon$. Es decir:

$$F(x_0) = \phi(x_0, T)$$

donde $\phi(x_0, t)$ es la solución del sistema que parte de la condición inicial $x(0) = x_0$ y evoluciona con el flujo del sistema del péndulo rápidamente forzado.

Para encontrar soluciones periódicas del sistema, buscamos un punto x_0 tal que $F(x_0) = x_0$. Esto se traduce en resolver:

$$G(x) = F(x) - x = 0 \tag{5.1.2}$$

Para resolver $G(x) = 0$ vamos a usar el método de Newton.

5.2. El método de Newton

Definición 5.2.1. *Sea $F = (f_1, f_2, ..., f_n) : U \subset \mathbb{R}^n \to \mathbb{R}^n$ función $\mathbb{C}^1$ y $\overline{x} \in U$ tal que $F(\overline{x}) = 0$. Recordemos que si $x \in U$:*

$$DF(x) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \dots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \dots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \dots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}.$$

Sea $x^{(k)}$ cercano a $\overline{x}$. Usando Taylor:

$$0 = F(\overline{x}) \approx F(x^{(k)}) + DF(x^{(k)})(\overline{x} - x^{(k)})$$

Resolviendo el sistema $F(x^{(k)}) + DF(x^{(k)})(\overline{x} - x^{(k)}) = 0$ obtenemos una aproximación x^{k+1} de $\overline{x}$ lo cual, si lo generalizamos, encontramos la función de iteración del método de Newton:

$$x^{k+1} = x^k - [DF(x^k)]^{-1} F(x^k) \quad \forall k \geq 0$$

Volviendo a nuestro sistema en particular, queremos usar el método de Newton para resolver $G(x) = 0$, lo cual es válido ya que $G(x)$ es diferenciable gracias a que $f(x, t) = \mu \sin \frac{t}{\epsilon} - \sin x$ lo es.

El método de Newton iterativo está dado por:

$$x_{n+1} = x_n - [DG(x_n)]^{-1} G(x_n)$$

donde $DG(x)$ es la derivada de $G(x)$, es decir:

$$DG(x) = DF(x) - I$$

siendo $DF(x)$ la derivada del flujo $F(x)$

5.3. Función de Newton

En nuestro programa principal hemos desarrollado una función que implemente todo este razonamiento.

Esta función llamada **newton** recibe como parámetros los valores *solution[2],DF[2][2],F[2],meps* donde el parámetro **solution[2]** guardará la solución que encuentre el método de Newton, el parámetro **DF[2][2]** es la matriz diferencial $DF(x)$, al parámetro **F[2]** se le pasa el punto inicial y se guarda en él $\phi(F(x))$, finalmente el parámetro **meps** guarda el valor de ϵ.

Dentro de esta función se llaman a otras funciones como **subtract_vectors, invert_matrix, multiply_matrices y update_F**, esta última tiene gran relevancia y se utiliza mucho a lo largo del programa.

```
void newton(mpreal solution[2], mpreal DF[2][2], mpreal F[2], mpreal meps){
        mpreal tolerance,x_n[2], x_n1[2], F_minus_x[2],step[2];
        mpreal identity[2][2] = {{1.0, 0.0}, {0.0, 1.0}};
        mpreal DF_minus_I[2][2], inverse_DF_minus_I[2][2],t=0;
        int max_iterations = 1000000, iteration = 0,k;
        k=mpreal::get_default_prec();
        tolerance=pow(0.5,k);
        F[0]=-3.14;
        F[1]=0.0;
        x_n[0]=F[0];
        x_n[1]=F[1];

        while (1) {
                //Calculamos F(x_n) y DF(x_n)
                update_F(F,DF,1, meps);
                //Calculamos F(x_n)-x
                subtract_vectors(F, x_n, F_minus_x);
                //Calculamos DF(x_n)-I
                for (int i = 0; i < 2; i++) {
                    for (int j = 0; j < 2; j++) {
                        DF_minus_I[i][j] = DF[i][j] - identity[i][j];
                    }
                }

                //Invertimos DF_minus_I
                if (!invert_matrix(DF_minus_I, inverse_DF_minus_I)) {
                    std::cerr << "Matriz DF(x_n) - I is singular. Cannot proceed." <<
            std::endl;
                    return;
                }

                // Calculamos paso: (DF(x_n) - I)^(-1) * (F(x_n) - x_n)
                multiply_matrices(inverse_DF_minus_I, F_minus_x, step);

                //Restamos x_n al step y lo guardamos en x_n1
```

```
                for (int i = 0; i < 2; i++) {
                    x_n1[i] = x_n[i] - step[i];
                }

                //Comprobamos convergencia
                mpreal error = sqrt(pow(x_n1[0] - x_n[0], 2) + pow(x_n1[1] - x_n[1], 2));
                if (error < tolerance) {
                        solution[0]=x_n1[0];
                        solution[1]=x_n1[1];
                        break;
                }

            //Actualizamos x_n=x_n1 y actualizamos F(x_n)
                for (int i = 0; i < 2; i++) {
                    x_n[i] = x_n1[i];
                }
                F[0]=x_n1[0];
                F[1]=x_n1[1];
                // Comprobamos límite de iteración
                iteration++;
                if (iteration > max_iterations) {
                     std::cerr << "Failed to converge within " << max_iterations
             << " iterations." << std::endl;
                    break;
                }
        }
        return;
}
```

5.4. Función update_F

La función **update_F** tiene gran importancia en el programa y se usa de forma repetida.

Es en esta función donde utilizamos el **método de Taylor** para encontrar el valor de **F(x)** y **DF(x)**.

Podemos observar que partiendo únicamente del sistema del péndulo rápidamente forzado no obtendremos la matriz $DF(x)$ la cual nos es necesaria para aplicar el método de Newton. Para resolver este problema, calculamos $DF(x)$ mediante las ecuaciones variacionales.

En concreto $DF(x_0)$ representa la matriz jacobiana del flujo $F(x_0)$, que relaciona la forma en que pequeñas perturbaciones en x_0 afectan al estado del sistema después de un tiempo **T**. Esto implica que estamos buscando cómo evoluciona una variación inicial δx_0 bajo el sistema dinámico. Para calcular esta matriz, se usan las ecuaciones variacionales respecto a x_0.

Teorema 5.4.1. *Sea $f : \Omega \subset \mathbb{R} \times \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n$, Ω abierto, una familia de funciones que dependen de un parámetro $\lambda \in \mathbb{R}^m$. Supongamos que f es una función continua y de*

clase $\mathbb{C}^1(\Omega)$ *respecto de* x *y* λ. *Consideramos el problema de Cauchy definido por*

$$\begin{cases} \dot{x} = f(t, x, \lambda) \\ x(t_0) = x_0 \end{cases}$$

Entonces $\phi(t; t_0, x_0, \lambda)$ *es de clase* $\mathbb{C}^1$ *respecto de* (t, t_0, x_0, λ).

Definición 5.4.2. *Sea* $\phi(t) = (\phi_1(t), \phi_2(t), ..., \phi_n(t))$ *la solución del problema de Cauchy (es decir,* $\phi(t) := \phi(t; t_0, x_0, \lambda)$*)*

$$\begin{cases} \dot{x} = f(t, x, \lambda) \\ x(t_0) = x_0 \end{cases}$$

donde suponemos que f *está en las hipótesis del resultado anterior.*

Definimos la matriz $M(t)$ *como*

$$M(t) := (\frac{\partial \phi(t)}{\partial x_0});\ M(t) \in \mathbb{M}_n$$

la matriz de las derivadas parciales de $\phi(t)$ *respecto de la condición inicial* $x_0 = (x_0^1, x_0^2, x_0^3, ..., x_0^n)$.

Teorema 5.4.3. *La matriz M(t) es la solución del problema de Cauchy (matricial, lineal, homogéneo)*

$$\begin{cases} \dot{X} = A(t)X \\ X(t_0) = Id_{\mathbb{R}^n} \end{cases}$$

donde $A(t) := D_x f(t, \phi(t; t_0, x_0, \lambda), \lambda)$. *Es decir, la matriz M(t) es la matriz fundamental principal del anterior sistema, y este sistema se llama* ***ecuación variacional para la condición inicial*** x_0.

Para que el paquete de software de **Taylor** nos genere un integrador numérico adecuado, le hemos pasado como parámetro el sistema del modelo del péndulo rápidamente forzado junto con el sistema de la ecuación variacional respecto de x_0. Le pasamos este texto plano:

x'= y;

y'= mu*sin(t/epsylon)-sin(x);

a'= c;

b'=d;

c'=(-1)*a*cos(x);

d'=(-1)*b*cos(x);

donde las dos primeras ecuaciones representan el sistema del péndulo rápidamente forzado, donde hemos introducido la variable y para reformular el sistema como un sistema de primer orden. Las cuatro últimas representan el sistema de la ecuación variacional respecto de x_0.

Finalmente, otro tema a considerar en esta función es que se aplican conversiones a los distintos tipos de datos. En nuestro programa principal usamos el wrapper de Pavel Holoborodko **MPFR C++** con el objetivo de aliviar la gramática de escribir expresiones matemáticas usando MPFR. En esta función convertimos datos *mpreal* a datos $mpfr\_t$ que son los que acepta el método de *Taylor*, al ser este compatible con MPFR.

Esta conversión se hace mediante la función *AssignMyFloat()*.

```
int update_F(mpreal F[2],mpreal DF[2][2], int direct, mpreal meps){

    MY_FLOAT punto[6], endtime, t;
    double log10abs_err, log10rel_err;
    int i, nsteps = 10000, order=10, l=0;
    int step_ctrl_method=2,k;
    static int flag=0;
    mpreal tf, mmu=pow(meps,1);
    k=mpreal::get_default_prec();

    //Reservamos memoria
    InitMyFloat(t);
    InitMyFloat(endtime);
    InitMyFloat(epsylon);
    InitMyFloat(mu);
    for (int j=0; j<6; j++) InitMyFloat(punto[j]);

    //Inicializamos valores
    mpfr_set_str(t,"0",10,GMP_RNDN);
    AssignMyFloat(epsylon,meps.mpfr_ptr());
    AssignMyFloat(mu,mmu.mpfr_ptr());
    tf = const_pi() * 2 * meps;
    AssignMyFloat(endtime,tf.mpfr_ptr());
    log10abs_err = k*log10(0.5);
    log10rel_err = k*log10(0.5);
    AssignMyFloat(punto[0],F[0].mpfr_ptr());
      AssignMyFloat(punto[1],F[1].mpfr_ptr());
      mpfr_set_str(punto[2],"1",10,GMP_RNDN);
      mpfr_set_str(punto[3],"0",10,GMP_RNDN);
      mpfr_set_str(punto[4],"0",10,GMP_RNDN);
      mpfr_set_str(punto[5],"1",10,GMP_RNDN);

    //Calculamos F(x) y DF(x)
    while(l!=1) {
            l=taylor_step_tfg(&t, punto, direct,2,log10abs_err, log10rel_err,
    &endtime, NULL, NULL,NULL);
    }

    //Devolvemos el resultado
    F[0]=punto[0];
    F[1]=punto[1];
    DF[0][0]=punto[2];
    DF[0][1]=punto[3];
    DF[1][0]=punto[4];
    DF[1][1]=punto[5];

    //Liberamos memoria
    for (int j=0; j<6; j++) ClearMyFloat(punto[j]);
    ClearMyFloat(t);
```

```
        ClearMyFloat(endtime);
        ClearMyFloat(epsylon);
        ClearMyFloat(mu);
        return 0;
}
```

5.5. Intervalo fundamental

Definición 5.5.1. *Definimos un intervalo fundamental de una variedad invariante como un segmento de curva perteneciente a dicha variedad que sirve como unidad inicial para describir la totalidad de la variedad a través de iteraciones del sistema dinámico considerado.*

Al encontrarnos con una variedad invariante, podemos definir toda la variedad de la siguiente forma.

Partiendo de un segmento de la variedad $\mathbb{I}(p_0, p_1)$, al que llamaremos **el intervalo fundamental**, se puede calcular de forma iterada las imágenes de este intervalo por la función F y como la variedad es invariante por F, en particular, tendremos que $F(\mathbb{I}(p_0, p_1)) := \mathbb{I}(F(p_0), F(p_1))$, pertenecerá a la variedad.

Definición 5.5.2. *Un intervalo fundamental propagado es el resultado de aplicar iterativamente el flujo del sistema dinámico a un intervalo fundamental, extendiendo así su alcance dentro de la variedad invariante.*

Observación 5.5.3. Este concepto se utiliza para determinar cómo una porción inicial de la variedad se deforma y distribuye bajo las transformaciones inducidas por el sistema.

En particular, el intervalo propagado se considera clave para identificar puntos de interés como intersecciones con otras estructuras dinámicas, como separatrices o ejes de coordenadas.

Al ser F en nuestro caso el flujo del sistema del péndulo rápidamente forzado, llamaremos a la iteración del cálculo de las imágenes de $\mathbb{I}(p_0, p_1)$ por F la *propagación* del intervalo fundamental.

En esta sección definimos una función *interval_fonamental* que *propaga* el intervalo fundamental $\mathbb{I}(p_0, p_1)$ hasta que la última imagen contenga el punto de corte entre las separatrices del sistema. Es decir, se van calculando las imágenes por el flujo F del intervalo $\mathbb{I}(p_0, p_1)$ hasta que el intervalo fundamental *propagado* $F^{(n)}(\mathbb{I}(p_0, p_1)) = \mathbb{I}(F^{(n)}(p_0), F^{(n)}(p_1))$ cumpla que $F^{(n)}(p_0)F^{(n)}(p_1) < 0$, que contenga el eje y.

Notación 5.5.4. *$F^{(n)}(x)$ denota la composición n veces de x por F.*

Ejemplo 5.5.5.

$$F^{(2)}(x) := F(F(x))$$
$$F^{(3)}(x) := F(F(F(x)))$$

Basándonos en esta lógica, hemos desarrollado una función que nos encuentre un intervalo, el **intervalo fundamental propagado**, que contenga el punto de corte de las separatrices. Tras ello, aplicaremos el método de la bisección para encontrar el punto de corte.

```
    int interval_fonamental(mpreal punta[2],mpreal puntb[2],  mpreal matrix[2][2],
    mpreal meps){

        int max_iterations = 1000000, iteration = 0;

        while(puntb[0]<0){
                update_F(punta,matrix,1,meps);
                update_F(puntb,matrix,1,meps);

                // Comprobamos límite de iteración
                iteration++;
                if (iteration > max_iterations) {
                     std::cerr << "Failed to converge within " << max_iterations <<
            " iterations." << std::endl;
                     break;
                }
        }

        return iteration;

}
```

5.6. Método de la bisección

Recordemos el Teorema de Bolzano

Teorema 5.6.1. *Dada una función f continua en [a,b], si $f(a)f(b) \leq 0$ entonces existe $c \in (a,b)$ con $f(c) = 0$*

Definición 5.6.2. *El método de la bisección se basa en el Teorema de Bolzano. La idea es la siguiente: Se construye una sucesión de intervalos [a,b]=$[a_0, b_0] \supset [a_1, b_1] \supset ... \supset [a_n, b_n]$... siempre con $f(a_i)f(b_i) < 0$.*

Empecemos considerando $[a,b] = [a_0, b_0]$ con $f(a_0)f(b_0) < 0$. Calculamos el punto medio $c_0 = \frac{a_0+b_0}{2}$. Ahora el intervalo $[a_1, b_1]$ viene definido de la siguiente forma:

$$[a_1, b_1] = \begin{cases} [a_0, c_0] & si\ f(a_0)f(c_0) < 0, \\ [c_0, b_0] & si\ f(c_0)f(b_0) < 0 \end{cases}$$

Para cada i haremos, $c_i = \frac{a_i+b_i}{2}$. Si $f(c_i) = 0$ entonces c_i es el cero. En general:

$$[a_{i+1}, b_{i+1}] = \begin{cases} [a_i, c_i] & si\ f(a_i)f(c_i) < 0, \\ [c_i, b_i] & si\ f(c_i)f(b_i) < 0 \end{cases}$$

La longitud del intervalo la encontraremos haciendo $\|[a_{i+1}, b_{i+1}]\| = \frac{b_0-a_0}{2^{i+1}}$

Después de explicar el método de la bisección, pasamos a explicar la función. Esta función es de gran importancia ya que en ella se ejecutan dos pasos necesarios para el objetivo final del programa.

En primer lugar, *propagamos* el intervalo fundamental por la variedad invariante hasta tener un intervalo que contenga el eje y. Esto lo hacemos para encontrar un intervalo que contenga el punto de corte entre las dos separatrices, ya que sabemos por simetría que este punto de corte se da en el eje y del plano.

En segundo lugar vamos aplicando el método de la bisección al intervalo fundamental con el fin de encontrar el punto x_k, y_k tal que $\Phi(x_k, y_k)$ sea el punto de corte de las dos separatrices.

Observación 5.6.3. Observemos que se ha de aplicar el método de la bisección en el intervalo fundamental, pero se utiliza como criterio para escoger una mitad el intervalo fundamental propagado. Es decir, si el intervalo fundamental es (a,b) utilizamos $\Phi(a) * \Phi(\frac{a+b}{2})$ vemos si contiene algún punto de la forma (0,x). Si es así, pasamos a reducir el intervalo a $(a, \frac{a+b}{2})$, si se diera el caso contrario nos quedaríamos con el intervalo $(\frac{a+b}{2}, b)$.

```
int bisection_on_variety(mpreal a[2], mpreal b[2], mpreal DF[2][2],
mpreal f_result[2], mpreal result[2], mpreal meps) {
mpreal mid[2]; // Punto medio
mpreal f_a[2], f_b[2], f_mid[2], tolerance; // Imagenes de x para a, b y mid
int iterations = 0, num_prop=0,k;
k=mpreal::get_default_prec();
tolerance=10*pow(0.5,k/2);
f_a[0]=a[0];
f_a[1]=a[1];
f_b[0]=b[0];
f_b[1]=b[1];

//Propagamos intervalo fundamental para que contenga el punto de corte con el eje y
num_prop=interval_fonamental(f_a,f_b,DF,meps);
printf("\n\nIntervalo fundamental propagado:\n");

//Imprimimos punto incial y punto final del intervalo fundamental propagado
cout << "Punto P1: ( " << f_a[0] << " , " << f_a[1] << ")" <<endl;
cout << "Punto P2: ( " << f_b[0] << " , " << f_b[1] << ")\n" <<endl;

// Verifica que el intervalo fundamental propagado contiene la raíz
if (f_a[0] * f_b[0] > 0) {
    printf("Error: El intervalo inicial no contiene la raíz (f(a) * f(b) > 0).\n");
    return 1;
}

// Método de bisección
while ( sqrt(pow(f_b[0] - f_a[0], 2) + pow(f_b[1] - f_a[1], 2)) > tolerance &&
iterations < 10000) {
    // Calcula el punto medio
    mid[0] = (a[0] + b[0]) / 2.0;
    mid[1] = (a[1] + b[1]) / 2.0;

    // Propaga el punto medio con la aplicación de Poincaré
    f_mid[0]=mid[0];
```

```
        f_mid[1]=mid[1];
        for(int i=0; i<num_prop;i++){update_F(f_mid, DF, 1, meps);}

        // Decide en qué subintervalo continuar
        if (f_mid[0] * f_a[0] < 0) {
            // La raíz está en f([a, mid])
            b[0] = mid[0];
            b[1] = mid[1];
            f_b[0] = f_mid[0];
            f_b[1] = f_mid[1];
        } else {
            // La raíz está en f([mid, b])
            a[0] = mid[0];
            a[1] = mid[1];
            f_a[0] = f_mid[0];
            f_a[1] = f_mid[1];
        }
        iterations++;
    }

    if (iterations >= 10000) {
        printf("Advertencia: Alcanzado el máximo de iteraciones sin convergencia.\n");
    }

    printf("Iteracio: %d\n",iterations);
    cout << "Aproximación de coordenada x: " << (f_a[0] + f_b[0]) / 2.0 <<endl;

    // El punto medio es una aproximación del cruce con el eje Y
    result[0]=(a[0] + b[0]) / 2.0;
    result[1]=(a[1] + b[1]) / 2.0;
    f_result[0] = 0.0; // La intersección ocurre en x = 0
    f_result[1] = (f_a[1] + f_b[1]) / 2.0; // Aproximación de y
    return num_prop;

}
```

5.7. Vector tangente

Tras haber calculado ya el punto de corte entre las separatrices principales, pasamos a calcular el vector tangente a la separatriz que pasa por el punto cercano a $(-\pi, 0)$.

Para ello vamos a utilizar dos formas de calcular el vector. La primera va a ser mediante la aplicación del método de Richardson y la segunda va a ser mediante la *propagación* de una curva de manera algebraica.

5.7.1. Método de extrapolación de Richardson

Teorema 5.7.1. *Sea $F \in \mathbb{C}(\mathbb{I})$ con $x_i \in \mathbb{I}$ y $a_i \in \mathbb{I}$ con $a_i \geq 0$. Entonces existe $y \in \mathbb{I}$ tal que $\sum_{i=1}^{m} a_i F(x_i) = F(y) \sum_{i=1}^{m} a_i$*

Proposición 5.7.2. *Sea ahora $f \in \mathbb{C}^8([c-h, c+h])$. Usando la fórmula centrada para $f''(c)$, obtenemos:*

$$F(h) = f''(c) + 2\frac{f^{(4)}(c)}{4!}h^2 + 2\frac{f^{(6)}(c)}{6!}h^4 + \frac{f^{(8)}(\alpha) + f^{(8)}(\beta)}{8!}h^6 \quad \alpha, \beta \in (c, c+h)$$

Esto muestra que $F(h) - f''(c) = a_1 h^2 + a_2 h^4 ...$

Definición 5.7.3. *Sea una función $F(h)$ que podamos evaluar para valores $h \neq 0$ aunque presenta dificultades para $h \to 0$. Supongamos que queremos calcular $f'(c)$. Usaremos la fórmula centrada:*

$$F(h) = \frac{f(c+h) - f(c-h)}{2h}$$

Hemos visto en esta sección que $F(h) = f'(c) + a_1 h^2 + a_2 h^4 + ...$ Ahora, si evaluamos f para $2h$:

$$F(2h) = f'(c) + a_1(2h)^2 + a_2(2h)^4 + \ ...$$

Sin tener en cuenta los términos con grado mayor que 2. Utilizando la siguiente resta:

$$F(2h) - F(h) = 3a_1 h^2$$

o de manera equivalente

$$\frac{F(2h) - F(h)}{3} = a_1 h^2 + \ ...$$

Si volvemos a la expresión anterior, ya conocida:

$$F(h) = f'(c) + \frac{F(2h) - F(h)}{3} + \ ...$$

La generalización de este proceso se hace introduciendo la notación siguiente:$F_1(h) = F(h)$ y $F_2(h) = F_1(h) + \frac{F_1(h) - F_1(2h)}{3}$. De manera más general:

$$F(h) = f'(c) + a_1 h^{p_1} + a_2 h^{p_2} + \ ... \quad p_1 < p_2 < p_3$$

Reiterando el proceso partiendo de $h = h_0$ y $\frac{h_0}{q}$, Reiterando el proceso partiendo de $h = h_0$ y $\frac{h_0}{q^2}, ...$ con $q > 1$, considerando $F_1(h) = F(h)$ obtenemos:

$$F_{k+1}(h) = F_k(h) + \frac{F_k(h) - F_k(2h)}{q^{p_k} - 1} \quad k = 1, 2...$$

Volviendo a nuestro programa, aplicamos el método de Richardson para calcular el vector tangente. En nuestra función **richardson** se ejecutan tres iteraciones del método de Richardson para la aproximación del vector tangente.

En esta función llamamos varias veces a la función **calcul_rich** que nos calcula las correspondientes $F_1(h), F_1(2h), F_2(h), F_2(2h)$ y $F_3(h)$.

Finalmente, en esta misma función normalizamos el vector tangente y lo imprimimos por pantalla.

```
  void richardson(mpreal solution[2], mpreal DF[2][2], int num_prop, mpreal vec_tan[2],
  mpreal meps){

        mpreal tolerance, solutionp[2], solutionm[2], F_1[2], F_2[2], F_1h[2], F_2h[2];
        int k=mpreal::get_default_prec();
        tolerance=pow(0.5,k);

        //F_1(h)
        calcul_rich(solution, DF, num_prop, tolerance, F_1, meps);

        //F_1(2*h)
        calcul_rich(solution, DF, num_prop, 2*tolerance, F_1h, meps);

        //F_2(h)
        F_2[0]=F_1[0]+(F_1[0]-F_1h[0])/3;
        F_2[1]=F_1[1]+(F_1[1]-F_1h[1])/3;

        //F_2(2h)
        calcul_rich(solution, DF, num_prop, 2*tolerance, F_1, meps);
        calcul_rich(solution, DF, num_prop, 2*tolerance, F_1h, meps);
        F_2h[0]=F_1[0]+(F_1[0]-F_1h[0])/3;
        F_2h[1]=F_1[1]+(F_1[1]-F_1h[1])/3;

        //Calcul F_3(h)
        vec_tan[0]=F_2[0]+(F_2[0]-F_2h[0])/15;
        vec_tan[1]=F_2[1]+(F_2[1]-F_2h[1])/15;
        mpreal norm = sqrt(vec_tan[0] * vec_tan[0] + vec_tan[1] * vec_tan[1]);
        vec_tan[0] /= norm
        vec_tan[1] /= norm;
        cout << "Converged to vector: f_vec = [ " << vec_tan[0] << " , " << vec_tan[1]
    << "]\n" <<endl;

}
void calcul_rich(mpreal entrada[2], mpreal DF[2][2], int num_prop, mpreal h,
mpreal result[2], mpreal meps){

        mpreal solutionp[2], solutionm[2];

        solutionp[0]=entrada[0]+h;
        solutionp[1]=entrada[1]+h;
        solutionm[0]=entrada[0]-h;
        solutionm[1]=entrada[1]-h;

        for(int i=0; i<num_prop;i++){
                update_F(solutionp,DF, 1, meps);
                update_F(solutionm,DF, 1, meps);
        }

        result[0]=(solutionp[0]-solutionm[0])/(2*h);
```

```
        result[1]=(solutionp[1]-solutionm[1])/(2*h);

        return;

}
```

5.7.2. Método Algebraico

Definición 5.7.4. *Un vector tangente u en $\boldsymbol{T}_p\mathbb{R}^2$ puede representarse como una derivada direccional*

$$u = \sum u_i \frac{\partial}{\partial x_0}$$

actuando sobre funciones $f : \mathbb{R}^2 \to \mathbb{R}$.

Proposición 5.7.5. *Sea $F : \mathbb{R}^2 \to \mathbb{R}^2$ una aplicación diferenciable. Sea $DF(p)$ la matriz jacobiana de F evaluada en el punto p. Sea $u \in \boldsymbol{T}_p\mathbb{R}^2$ un vector tangente en el punto p. La derivada de F actúa sobre u para transformar el vector tangente de p al vector tangente de $F(p)$:*

$$v = DF(p)u$$

Esto se fundamenta en que $DF(p)$ describe cómo las derivadas parciales en p se transforman bajo F. En coordenadas:

$$v_i = \sum_{j=1}^{2} \frac{\partial F_i}{\partial x_j}(p)u_j$$

Observación 5.7.6. La interpretación geométrica es que en el espacio tangente, $DF(p)$ da una transformación lineal que lleva vectores tangentes en $\mathbf{T}_p\mathbb{R}^2$ a vectores tangentes en $\mathbf{T}_{F(p)}\mathbb{R}^2$

Basados en esta última proposición podemos calcular el vector tangente en el punto de corte de las separatrices mediante la *propagación* del vector tangente en el punto del intervalo fundamental.

Observamos que cerca del punto fijo calculado mediante el método de Newton, las dos separatrices se comportan como rectas de pendientes 1 y -1 respectivamente.

Por lo tanto, sea el intervalo fundamental $[p_1, p_2]$, el vector tangente en el intervalo fundamental es $u = p_2 - p_1$. Mediante la *propagación* de este vector y del intervalo fundamental a través de la aplicación de Poincaré podemos determinar el vector tangente en el punto donde se cortan las separatrices.

Aplicando esta lógica es como hemos programado la función vector_prop.

```
void vector_prop(mpreal solution[2], mpreal DF[2][2], mpreal tan[2], mpreal f_tan[2],
int num_prop, mpreal meps){

    mpreal norm;

    for(int i=0; i<num_prop;i++){
            update_F(solution,DF, 1, meps);
```

```
                multiply_matrices(DF, tan, f_tan);
                tan[0]=f_tan[0];
                tan[1]=f_tan[1];

        }

        norm= sqrt(f_tan[0] * f_tan[0] + f_tan[1] * f_tan[1]);
        f_tan[0]=f_tan[0]/norm;
        f_tan[1]=f_tan[1]/norm;

        cout << "Converged to vector: f_vec = [ " << f_tan[0] << " , " << f_tan[1]
    << "]\n" <<endl;

        return;
}
```

5.8. Cálculo del ángulo y comprovación

Finalmente tras haber calculado dos vectores tangentes mediante diferentes métodos, uno de aproximación diferencial y otro basado en el álgebra, podemos calcular el ángulo que separa las dos separatrices principales.

Para ello, aprovechándonos de la simetría de todo el sistema del péndulo rápidamente forzado, calculamos este ángulo mediante dos pasos:

1. Calculamos el ángulo del vector calculado con el eje de las x, es decir con el vector (1,0).
2. Multiplicamos este ángulo por 2 aprovechándonos de la simetría del sistema.

Observación 5.8.1. Sea v el vector calculado, para calcular el ángulo con el vector (1,0) podemos utilizar la igualdad:

$$\alpha = [\tan \frac{v_1}{v_0}]^{-1}$$

Tras el cálculo del ángulo de forma experimental, finalmente, comprovamos el ángulo basándonos en distintos artículos teóricos que han desarrollado ecuaciones para este ángulo según el valor de los parámetros.

En concreto, en este programa nos basamos en el artículo de Amadeu Delshams (Departamento de Matemática Aplicada de la Universidad Politécnica de Cataluña) y Teresa M. Seara (Departamento de Matemática Aplicada de la Universidad Politécnica de Cataluña).

Este artículo concluye que

$$\alpha = \frac{\pi}{2\epsilon} \frac{\mu}{\cosh \frac{\pi}{2\epsilon}} [1 + O(\mu, \epsilon^2)]$$

Para ello hemos desarrollado la función **comprovacio_angle**:

```
    void comprovacio_angle(mpreal meps){

        mpreal angle,mmu;

        mmu=pow(meps,1);

        angle = M_PI/(2*meps) * mmu/cosh(M_PI/(2*meps));

        cout << "Angle de diferencia entre separatrius segons el abstract: " << angle
    << " * (1 + O( " << mmu << ", "<< meps*meps <<")" <<endl;

}
```

5.9. Funciones auxiliares

A lo largo de todo el programa hemos ido creando funciones auxiliares para ayudar en el cálculo y la estructura del código.

Considero que estas funciones auxiliares también tienen su importancia ya que sin ellas sería imposible la ejecución correcta del código.

5.9.1. Función calcula_veps

Esta función se utiliza para calcular los vectores propios de una matriz.

Al estar todo englobado con el sistema del péndulo forzado, esta función ha sido programada en particular para este sistema y por ello solo se considera que el tamaño de la matriz sea de 2×2.

Para calcular de forma sencilla los vectores propios de la matriz pasada por parámetro, resolvemos el sistema mediante el método de Cramer.

Aplicando este método, hemos programado la función:

```
    void calcula_veps(mpreal matrix[2][2], mpreal eigenvalue, mpreal eigenvector[2]){

    mpreal a = matrix[0][0];
    mpreal b = matrix[0][1];
    mpreal c = matrix[1][0];
    mpreal d = matrix[1][1];

    // Matriz resultante después de restar I
    mpreal new_matrix[2][2] = {
        {a - eigenvalue, b},
        {c, d - eigenvalue}
    };
        mpreal x1, x2;

        // Elegimos una ecuación del sistema
```

```
        x1 = -new_matrix[0][1];

        // Supongamos que b no es 0
        x2 = new_matrix[0][0];

        // Normalización del vector propio
        mpreal norm = sqrt(x1 * x1 + x2 * x2);
        eigenvector[0] = x1 / norm;
        eigenvector[1] = x2 / norm;

}
```

5.10. Main

Finalmente, todo este trabajo no tendría sentido sin un hilo conductor que lo unificara todo y lo guiara para llegar al cálculo del ángulo y la comprobación de este.

Para ello, hemos desarrollado la función main:

```
int main() {
    mpreal::set_default_prec(mpfr::digits2bits(DIGITS));
    mpreal DF[2][2], F[2], solution[2], p1[2], h, p2[2],
vacia[2][2]={{0.0, 0.0}, {0.0, 0.0}}, f_solution[2];
    mpreal vap1, vap2;
    mpreal meps;
        mpreal vep1[2], vep2[2];
        mpreal tan[2], f_tan[2], angle, vec_rich[2];
        int num_prop, k;
        k=mpreal::get_default_prec();
        meps=0.1;
        h=pow(0.5,k/2);
        printf("%d\n",k);
        cout << "Paso para calcular variedades: " << h <<endl;
        printf("Parte del punto (-3.14,0)\n");

        //Calculamos solucion de F(x)-x para encontrar puntos fijos
    //del sistema(orbitas periodicas)
        newton(solution,DF, F, meps);
        cout << "Converged to solution: x = [ " << solution[0] << " , "
    << solution[1] << "]" <<endl;

        //Imprimimos F
    printf("Matriz F(x):\n");
    cout << F[0] <<endl<< F[1] <<endl;

    //Imprimimos matriz DF
    printf("Matriz DF:\n");
    cout << DF[0][0] << "    " << DF[0][1] << "\n" <<DF[1][0]<<"    "<<
    DF[1][1]<<endl;
```

```
        //Calculamos vaps
        vaps(DF, &vap1, &vap2);
        printf("Valores propios:\n");
        cout<<"Vap1:"<<vap1<<" \n";
        cout<<"Vap2:"<<vap2<<" \n";
        mpreal modulo1 = calculate_modulus(vap1);
        mpreal modulo2 = calculate_modulus(vap2);
        printf("modulos de los valores propios:\n");
        cout<<"Vap1:"<<modulo1<<"\n";
        cout<<"Vap2:"<<modulo2<<"\n";

        //Decimos si son estables o inestables
        //Punto estable(atractivo)
        if(modulo1<1 && modulo2<1){printf("Punto estable\n");}

        //Punto inestable
        else if(modulo1>1 || modulo2>1 ){printf("Punto inestable\n");}

        else if (modulo1==1 && modulo2==1){printf("Punto centrico\n");}

        //Imprimimos veps asociados a los vaps
        cout <<"Vector propio asociado al vap( " << vap1 << " ):\n";

        calcula_veps(DF, vap1, vep1);

        cout << "(" << vep1[0] << " , " << vep1[1] << ")" <<endl;

        cout <<"Vector propio asociado al vap( " << vap2 << " ):\n";

        calcula_veps(DF, vap2, vep2);

        cout << "(" << vep2[0] << " , " << vep2[1] << ")" <<endl;

        //Calculamos intervalo fundamental

        //Calculamos p1=p0 + h*v
        p1[0]=solution[0]+h*(-vep1[0]);
        p1[1]=solution[1]+h*(-vep1[1]);

        //Calcumos p2
update_F(p1,vacia,1,meps);

        p2[0]=p1[0];
        p2[1]=p1[1];
        p1[0]=solution[0]+h*(-vep1[0]);
        p1[1]=solution[1]+h*(-vep1[1]);
        tan[0]=p2[0]-p1[0];
        tan[1]=p2[1]-p1[1];
```

```
        //Imprimimos p1 y p2, este es el intervalo fundamental con el segmento recto
        printf("\n\nIntervalo fundamental:\n");

        cout <<"Punto P1: ( " << p1[0] << " , " << p1[1] << " )\n";

        cout <<"Punto P2: ( " << p2[0] << " , " << p2[1] << " )\n";

        //Bucle para aplicar el metodo de la biseccion y la propagacion del intervalo
        printf("\n\nEntramos en el metodo de la biseccion:\n");

num_prop=bisection_on_variety(p1, p2, vacia, f_solution, solution, meps);

        cout <<"Converged to solution: sol = [ " << solution[0] << " , "
        << solution[1] << " ]\n";

        cout <<"Converged to imaged solution: f_sol = [ " << f_solution[0] << " , "
        << f_solution[1] << " ]\n\n";

        cout <<"Vector tangente: vec = [ " << tan[0] << " , " << tan[1] << " ]\n";

        //Calculamos el vector tangente en el eje y mediante extrapolacion de Richardson

        richardson(solution, DF, num_prop,vec_rich, meps);

        //Calculo del angulo
angle=2*acos(angle_cos(vec_rich));

        cout <<"Angle de diferencia entre separatrius segons el metode de derivacio
    Richardson i el cos: " << angle <<"\n";

angle=2*atan2(vec_rich[1],vec_rich[0]);

        cout <<"Angle de diferencia entre separatrius segons el metode de derivacio
    Richardson i la tan: " << angle <<"\n\n";

        //Calculo de la imagen del vector tangente vector tangente en el eje y,
        //iteramos tantas veces como se ha iterado para encontrar el intervalo
        //fundamental
        cout <<"Vector tangente: vec = [ " << tan[0] << " , " << tan[1] << " ]\n";

        vector_prop(solution, DF, tan, f_tan, num_prop, meps);

        //Calculo del angulo angle=2*acos(angle_cos(f_tan));

        cout <<"Angle de diferencia entre separatrius segons el metode de propagacio
    i el cos: " << angle <<"\n";

        angle=2 * atan2(f_tan[1],f_tan[0]);
```

```
    cout <<"Angle de diferencia entre separatrius segons el metode de propagacio
i la tan: " << angle <<"\n\n";

    comprovacio_angle(meps);

    return 0;
}
```

Capítulo 6

Ejecución y Resultados

El análisis de los resultados obtenidos al calcular el ángulo entre las separatrices del sistema del péndulo rápidamente forzado demuestra claramente la necesidad de utilizar aritmética de alta precisión. A lo largo de la ejecución del programa, implementado en C++ con soporte de la biblioteca MPFR, se observaron diferencias significativas en la estabilidad y exactitud de los cálculos cuando se variaron los parámetros de precisión y los valores de entrada del sistema.

Los cálculos realizados en este trabajo se basan en parámetros como ϵ y μ, cuyos valores pequeños generan ángulos extremadamente reducidos.

Uno de los aspectos más reveladores es cómo pequeñas diferencias en los valores iniciales o en los parámetros ϵ y μ afectan el cálculo del ángulo de separación. Por ejemplo, la tabla de resultados muestra que, a medida que se incrementa el número de dígitos de precisión con el parámetro DIGITS, las soluciones convergen a valores significativamente más consistentes, mientras que con una precisión más baja, los resultados fluctúan de manera errática o incluso divergen.

A continuación, se detalla la relación entre los parámetros, los dígitos de precisión requeridos y los ángulos calculados:

ϵ	μ	Dígitos	Ángulo
0.1	0.1	60	$4{,}73438 \times 10^{-7}$
0.01	0.01	100	$4{,}7538 \times 10^{-68}$
0.01	0.0001	100	$4{,}78119 \times 10^{-70}$
0.01	0.000000001	100	$4{,}76125 \times 10^{-72}$
0.001	0.001	800	$5{,}0534 \times 10^{-690}$

Cuadro 6.1: Tabla de resultados.

Los resultados reflejan cómo los valores más pequeños de ϵ y μ hacen que los cálculos sean sensibles al error numérico, lo que requiere un mayor número de dígitos de precisión para mantener la exactitud.

Por ejemplo:

- Para $\epsilon = 0{,}01$ y $\mu = 0{,}01$, los resultados son del orden de 10^{-68}, lo que exige al menos 100 dígitos de precisión para evitar errores de redondeo.
- En el caso de $\epsilon = 0{,}001$ y $\mu = 0{,}001$, el resultado decrece en gran medida debido a

la dinámica del sistema, alcanzando valores del orden de 10^{-690}.

Esto demanda hasta 800 dígitos de precisión, mostrando que las herramientas tradicionales de punto flotante no son adecuadas para este problema.

La aritmética de alta precisión utilizada en este trabajo, implementada con la biblioteca MPFR, permite manejar números de gran escala con exactitud y reduce significativamente los errores numéricos acumulativos. Esto es crucial, ya que el cálculo del ángulo entre las separatrices implica operaciones complejas, como iteraciones de propagación y derivación numérica, donde pequeños errores podrían amplificarse y llevar a resultados incorrectos o inconsistentes.

Por último, cabe destacar el impacto del método de Taylor utilizado en el cálculo. Este método, al permitir el control dinámico del orden y tamaño del paso, complementa perfectamente la aritmética de alta precisión, adaptándose a las necesidades de cada intervalo de integración y garantizando un equilibrio entre la eficiencia y la precisión computacional. Sin estas herramientas, el cálculo preciso del ángulo habría sido inviable.

El uso de herramientas de alta precisión no es solo una mejora, sino una necesidad en el análisis del péndulo rápidamente forzado. Sin estas herramientas, los resultados serían inalcanzables o carecerían de la fiabilidad necesaria para aplicaciones prácticas, como la modelización de sistemas físicos sensibles o el estudio de fenómenos no lineales en la dinámica caótica.

Capítulo 7

Conclusiones

En este trabajo se ha abordado el análisis de las separatrices del sistema del péndulo rápidamente forzado, con especial atención al cálculo del ángulo entre estas. Este problema, además de ser un desafío computacional, aporta un enfoque significativo al estudio de sistemas dinámicos no lineales y sus propiedades en contextos altamente sensibles. A continuación, se sintetizan las conclusiones obtenidas a partir de este estudio:

7.1. La importancia del ángulo entre separatrices

El ángulo entre las separatrices es un parámetro crucial para comprender la dinámica del sistema. Este ángulo determina la geometría de las variedades invariantes asociadas a los puntos de equilibrio, proporcionando información clave sobre la estructura caótica del sistema. Además, su cálculo permite estimar con mayor precisión la sensibilidad del sistema a las condiciones iniciales y predecir su comportamiento a largo plazo.

En el caso del péndulo rápidamente forzado, el ángulo entre separatrices revela la influencia de los parámetros ϵ y μ, que modulan la amplitud y la frecuencia del forzamiento externo. Este conocimiento es de gran relevancia tanto para la teoría de sistemas dinámicos como para aplicaciones prácticas en física y otras disciplinas.

7.2. El reto computacional: la necesidad de alta precisión

Uno de los principales hallazgos de este trabajo es la identificación de la necesidad de emplear aritmética de alta precisión para resolver problemas con parámetros muy pequeños. En este sistema, la combinación de valores extremadamente reducidos de ϵ y μ genera ángulos entre separatrices de un orden de magnitud que desborda la capacidad de la aritmética estándar en punto flotante, como el tipo double. Por ejemplo:

Con $\epsilon = 0{,}01$ y $\mu = 0{,}01$, el ángulo calculado es del orden de 10^{-68}. Para $\epsilon = 0{,}001$ y $\mu = 0{,}001$, el valor alcanza magnitudes cercanas a 10^{-690}.

Estos resultados demuestran cómo los errores de redondeo en las representaciones numéricas estándar pueden acumularse, volviendo los resultados imprecisos o inservibles. En consecuencia, el uso de herramientas como la biblioteca MPFR, que permite cálculos con precisión arbitraria, es esencial para garantizar resultados fiables.

7.3. Relevancia de los resultados para la teoría de sistemas dinámicos

El estudio del ángulo entre separatrices no solo permite describir con mayor detalle el sistema del péndulo rápidamente forzado, sino que también abre nuevas perspectivas para analizar fenómenos caóticos en sistemas no lineales. Este análisis es aplicable en áreas como:

- Modelos físicos: Por ejemplo, en sistemas de partículas cargadas en campos oscilantes o en el estudio de resonancias en sistemas mecánicos y electrónicos.
- Astrofísica y dinámica orbital: Donde las separatrices determinan zonas de estabilidad e inestabilidad en la trayectoria de cuerpos celestes.
- Ingeniería y diseño de sistemas robustos: Al analizar sistemas sujetos a forzamientos externos para prever posibles fallos en condiciones críticas.

7.4. Limitaciones y posibles mejoras

Aunque el trabajo realizado proporciona resultados consistentes y fiables, existen áreas en las que se pueden realizar mejoras para futuros estudios. Entre ellas:

- Optimización computacional: La implementación de algoritmos más eficientes para cálculos de alta precisión podría reducir el tiempo de ejecución sin comprometer la exactitud.
- Explorar valores más extremos de ϵ y μ podría ofrecer una visión más completa de las transiciones entre comportamiento regular y caótico.
- Simulaciones visuales: Representar gráficamente la evolución de las separatrices y los puntos de cruce en el espacio de fases facilitaría la interpretación de los resultados y su comunicación a una audiencia más amplia.

Este trabajo ha contribuido a la comprensión del sistema del péndulo rápidamente forzado al destacar la importancia del ángulo entre separatrices como un indicador de su dinámica subyacente. Además, ha demostrado la utilidad práctica de la aritmética de alta precisión en problemas matemáticos y físicos complejos, estableciendo un marco sólido para futuras investigaciones.

La combinación de modelos teóricos, herramientas numéricas avanzadas y enfoques computacionales rigurosos ha permitido explorar un problema complejo con un nivel de detalle que sería inalcanzable mediante métodos convencionales. Este enfoque no solo enriquece la teoría de sistemas dinámicos, sino que también subraya la importancia de utilizar tecnología puntera para abordar problemas de frontera en la ciencia y la ingeniería.

Bibliografía

[1] Hirsch, Morris W.; Smale, Stephen; Devaney, Robert L. Differential equations, dynamical systems and an introduction to chaos, 2a edición. Elsevier Academic Press, 2004. Pure and Applied Mathematics, Vol. 60.

[2] Poincaré, Henri. Sur le problème des trois corps et les équations de la dynamique. Acta Mathematica, vol. 13, 1890, pp. 1–270.

[3] Arnold, Vladimir I. Instability of dynamical systems with several degrees of freedom. Soviet Mathematics Doklady, vol. 5, 1964, pp. 581–585. Melnikov, Valery K. On the stability of the center for time periodic perturbations. Transactions of the Moscow Mathematical Society, vol. 12, 1963, pp. 1–57.

[4] Delshams, Amadeu; Seara, Teresa M. An asymptotic expression for the splitting of separatrices of the rapidly forced pendulum. Communications in Mathematical Physics, vol. 150, 1992, pp. 433–463.

[5] Wiggins, Stephen. Introduction to applied nonlinear dynamical systems and chaos (2nd ed.). Texts in Applied Mathematics, vol. 2. Springer, 2003, pp. 28–76.

[6] Guckenheimer, John; Holmes, Philip. Nonlinear oscillations, dynamical systems, and bifurcations of vector fields. Springer-Verlag, 1983. Applied Mathematical Sciences, vol. 42, pp. 1–60.

[7] Gustafson, John. Floating-point computation. IEEE Computer Society Press, 1992.

[8] Wilkinson, James H. The representation of real numbers in computer arithmetic. Mathematics of Computation, vol. 17, no. 81, 1963, pp. 263–276.

[9] Barton, Michael M. D.; Brown, Kenneth L. IEEE 754: A Standard for Binary Floating-Point Arithmetic. ACM Computing Reviews, 1984.

[10] Goldberg, David. The IEEE 754 standard for binary floating-point arithmetic. ACM Computing Surveys, vol. 23, no. 1, 1991, pp. 5–22.

[11] Holoborodko, Pavel. MPFR - Multiple Precision Floating-Point Reliable Library. (n.d.). Recuperado de http://www.holoborodko.com/pavel/mpfr/.

[12] Jorba, Àngel; Zou, Maorong. A software package for the numerical integration of ODEs by means of high-order Taylor methods. (n.d.).

[13] Barton, Derrick; Willers, Ian M.; Zahar, Ralph V. M. The Automatic Solution of Ordinary Differential Equations by the Method of Taylor Series. Computer Journal, vol. 14, no. 3, 1970, pp. 243–248.

[14] Kirlinger, Gerald; Corliss, George F. On Implicit Taylor Series Methods for Stiff ODEs. In Computer Arithmetic and Enclosure Methods, edited by L. Atanassova and J. Herzberger, Amsterdam: North-Holland, 1992, pp. 371–379.

Anexo

Programa completo

El programa completo quedaría de la siguiente forma:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mpreal.h"
#include <iostream>
#include "taylor.h" // Aquí se incluye el archivo que puede tener funciones C

using namespace std;
using namespace mpfr;

#define DIGITS 60

/*External variables*/
MY_FLOAT epsylon,mu;

// Función para calcular el determinante de una matriz 2x2

mpreal determinant_2x2(mpreal a, mpreal b, mpreal c, mpreal d) {
    return a * d - b * c;
}

// Función para calcular los vectores propios

void calcula_veps(mpreal matrix[2][2], mpreal eigenvalue, mpreal eigenvector[2]){
    mpreal a = matrix[0][0];
    mpreal b = matrix[0][1];
    mpreal c = matrix[1][0];
    mpreal d = matrix[1][1];

    // Matriz resultante después de restar I
    mpreal new_matrix[2][2] = {
        {a - eigenvalue, b},
        {c, d - eigenvalue}
```

```
    };
        mpreal x1, x2;

        // Elegimos una ecuación del sistema
        x1 = -new_matrix[0][1]; // Supongamos que b no es 0
        x2 = new_matrix[0][0];

        // Normalización del vector propio
        mpreal norm = sqrt(x1 * x1 + x2 * x2);
        eigenvector[0] = x1 / norm;
        eigenvector[1] = x2 / norm;
}

// Funcion para multiplicar una matriz 2x2 por un vector del plano

void multiply_matrices(mpreal A[2][2], mpreal B[2], mpreal result[2]) {
    for (int i = 0; i < 2; i++) {
        result[i] = 0.0;
        for (int j = 0; j < 2; j++) {
            result[i] += A[i][j] * B[j];
        }
    }
}

// Funcion para restar dos vectores

void subtract_vectors(mpreal A[2], mpreal B[2], mpreal result[2]) {
    for (int i = 0; i < 2; i++) {
        result[i] = A[i] - B[i];
    }
}

//Funcion para calcular vaps

int vaps(mpreal matrix[2][2], mpreal *eigenvalue1, mpreal *eigenvalue2) {
    mpreal a = matrix[0][0];
    mpreal b = matrix[0][1];
    mpreal c = matrix[1][0];
    mpreal d = matrix[1][1];

    // Calcular la traza (suma de diagonales) y el determinante
    mpreal trace = matrix[0][0] + matrix[1][1];
    mpreal determinant = matrix[0][0]* matrix[1][1] - matrix[0][1] * matrix[1][0];

    // Calcular el discriminante del polinomio característico
    mpreal discriminant = trace * trace - 4 * determinant;

    // Valores propios reales
    *eigenvalue1 = (trace + sqrt(discriminant)) / 2.0;
```

```
    *eigenvalue2 = (trace - sqrt(discriminant)) / 2.0;
    return 0;
}

// Función para calcular el módulo de un valor propio complejo

mpreal calculate_modulus(mpreal eigenvalue) {
    return sqrt(eigenvalue * eigenvalue);
}

// Funcion para calcular la inversa de una matriz 2x2

int invert_matrix(mpreal A[2][2], mpreal inverse[2][2]) {
    mpreal determinant = A[0][0] * A[1][1] - A[0][1] * A[1][0];

    if (fabs(determinant) < 1e-12) {
        return 0; // La matriz es singular
    }

    inverse[0][0] = A[1][1] / determinant;
    inverse[0][1] = -A[0][1] / determinant;
    inverse[1][0] = -A[1][0] / determinant;
    inverse[1][1] = A[0][0] / determinant;

    return 1;
}

//funcion para calcular F(x) y DF(x)

int update_F(mpreal F[2],mpreal DF[2][2], int direct, mpreal meps){
        MY_FLOAT punto[6], endtime, t;
        double log10abs_err, log10rel_err;
        int i, nsteps = 10000, order=10, l=0;
        int step_ctrl_method=2,k;
        static int flag=0;
        mpreal tf, mmu=pow(meps,1);
        k=mpreal::get_default_prec();
        InitMyFloat(t);
        InitMyFloat(endtime);
        InitMyFloat(epsylon);
        InitMyFloat(mu);
        for (int j=0; j<6; j++) InitMyFloat(punto[j]);

        mpfr_set_str(t,"0",10,GMP_RNDN);
        AssignMyFloat(epsylon,meps.mpfr_ptr());
        AssignMyFloat(mu,mmu.mpfr_ptr());
        tf = const_pi() * 2 * meps;
        AssignMyFloat(endtime,tf.mpfr_ptr());
        log10abs_err = k*log10(0.5);
```

```
        log10rel_err = k*log10(0.5);
        AssignMyFloat(punto[0],F[0].mpfr_ptr());
          AssignMyFloat(punto[1],F[1].mpfr_ptr());
          mpfr_set_str(punto[2],"1",10,GMP_RNDN);
          mpfr_set_str(punto[3],"0",10,GMP_RNDN);
          mpfr_set_str(punto[4],"0",10,GMP_RNDN);
          mpfr_set_str(punto[5],"1",10,GMP_RNDN);

        while(l!=1) {
                l=taylor_step_tfg(&t, punto, direct,2,log10abs_err, log10rel_err,
        &endtime, NULL, NULL,NULL);
        }

        F[0]=punto[0];
        F[1]=punto[1];
        DF[0][0]=punto[2];
        DF[0][1]=punto[3];
        DF[1][0]=punto[4];
        DF[1][1]=punto[5];

        for (int j=0; j<6; j++) ClearMyFloat(punto[j]);
        ClearMyFloat(t);
        ClearMyFloat(endtime);
        ClearMyFloat(epsylon);
        ClearMyFloat(mu);
        return 0;
}

//Funcion que utiliza el metodo de Newton para encontrar el punto del sistema que corte
//con el eje x(orbitas periódicas)

void newton(mpreal solution[2], mpreal DF[2][2], mpreal F[2], mpreal meps){
        mpreal tolerance,x_n[2], x_n1[2], F_minus_x[2],step[2];
        mpreal identity[2][2] = {{1.0, 0.0}, {0.0, 1.0}};
            mpreal DF_minus_I[2][2], inverse_DF_minus_I[2][2],t=0;
            int max_iterations = 1000000, iteration = 0,k;
            k=mpreal::get_default_prec();
            tolerance=pow(0.5,k);
        F[0]=-3.14;
        F[1]=0.0;
        x_n[0]=F[0];
        x_n[1]=F[1];

        while (1) {
                //Calculamos F(x_n) y DF(x_n)
                update_F(F,DF,1, meps);

                //Calculamos F(x_n)-x
                subtract_vectors(F, x_n, F_minus_x);
```

```
                //Calculamos DF(x_n)-I
                for (int i = 0; i < 2; i++) {
                    for (int j = 0; j < 2; j++) {
                        DF_minus_I[i][j] = DF[i][j] - identity[i][j];
                    }
                }

                //Invertimos DF_minus_I
                if (!invert_matrix(DF_minus_I, inverse_DF_minus_I)) {
                    std::cerr << "Matrix DF(x_n) - I is singular. Cannot proceed."
            << std::endl;
                    return;
                }

                // Calculamos paso: (DF(x_n) - I)^(-1) * (F(x_n) - x_n)
                multiply_matrices(inverse_DF_minus_I, F_minus_x, step);

                //Restamos x_n al step y lo guardamos en x_n1
                for (int i = 0; i < 2; i++) {
                    x_n1[i] = x_n[i] - step[i];
                }

                //Comprobamos convergencia
                mpreal error = sqrt(pow(x_n1[0] - x_n[0], 2) + pow(x_n1[1] - x_n[1], 2));
                if (error < tolerance) {
                        solution[0]=x_n1[0];
                        solution[1]=x_n1[1];
                        break;
                }

                //Actualizamos x_n=x_n1 y actualizamos F(x_n)
                for (int i = 0; i < 2; i++) {
                    x_n[i] = x_n1[i];
                }
                      F[0]=x_n1[0];
                   F[1]=x_n1[1];
                // Comprobamos límite de iteración
                iteration++;
                if (iteration > max_iterations) {
                     std::cerr << "Failed to converge within " << max_iterations <<
             " iterations." << std::endl;
                    break;
                }
        }
        return;
}

//función para calcular la imagen del intervalo fundamental, propaga el intervalo
```

```
int interval_fonamental(mpreal punta[2],mpreal puntb[2],  mpreal matrix[2][2],
    mpreal meps){

        int max_iterations = 1000000, iteration = 0;

        while(puntb[0]<0){
                update_F(punta,matrix,1,meps);
                update_F(puntb,matrix,1,meps);

                // Comprobamos límite de iteración
                iteration++;
                if (iteration > max_iterations) {
                     std::cerr << "Failed to converge within " << max_iterations <<
            " iterations." << std::endl;
                    break;
                }
        }

        return iteration;

}

// Método de Bisección para encontrar el punto de cruce con el eje Y

int bisection_on_variety(mpreal a[2], mpreal b[2], mpreal DF[2][2], mpreal f_result[2],
mpreal result[2], mpreal meps) {
    mpreal mid[2]; // Punto medio
    mpreal f_a[2], f_b[2], f_mid[2], tolerance; // Imagenes de x para a, b y mid
    int iterations = 0, num_prop=0,k;
    k=mpreal::get_default_prec();
    tolerance=10*pow(0.5,k/2);
    f_a[0]=a[0];
    f_a[1]=a[1];
    f_b[0]=b[0];
    f_b[1]=b[1];

    //Propagamos intervalo fundamental para que contenga el punto de corte con el eje y
    num_prop=interval_fonamental(f_a,f_b,DF,meps);
    printf("\n\nIntervalo fundamental propagado:\n");

    //Imprimimos punto incial y punto final del intervalo fundamental propagado
    cout << "Punto P1: ( " << f_a[0] << " , " << f_a[1] << ")" <<endl;
    cout << "Punto P2: ( " << f_b[0] << " , " << f_b[1] << ")\n" <<endl;

    // Verifica que el intervalo fundamental propagado contiene la raíz
    if (f_a[0] * f_b[0] > 0) {
        printf("Error: El intervalo inicial no contiene la raíz (f(a) * f(b) > 0).\n");
        return 1;
```

```
    }

    // Método de bisección
    while ( sqrt(pow(f_b[0] - f_a[0], 2) + pow(f_b[1] - f_a[1], 2)) > tolerance &&
    iterations < 10000) {

        // Calcula el punto medio
        mid[0] = (a[0] + b[0]) / 2.0;
        mid[1] = (a[1] + b[1]) / 2.0;

        // Propaga el punto medio con la aplicación de Poincaré

        f_mid[0]=mid[0];
        f_mid[1]=mid[1];
        for(int i=0; i<num_prop;i++){update_F(f_mid, DF, 1, meps);}

        // Decide en qué subintervalo continuar
        if (f_mid[0] * f_a[0] < 0) {

            // La raíz está en f([a, mid])
            b[0] = mid[0];
            b[1] = mid[1];
            f_b[0] = f_mid[0];
            f_b[1] = f_mid[1];
        } else {

            // La raíz está en f([mid, b])
            a[0] = mid[0];
            a[1] = mid[1];
            f_a[0] = f_mid[0];
            f_a[1] = f_mid[1];
        }
        iterations++;
    }
    if (iterations >= 10000) {
        printf("Advertencia: Alcanzado el máximo de iteraciones sin convergencia.\n");
    }
    printf("Iteracio: %d\n",iterations);
    cout << "Aproximación de coordenada x: " << (f_a[0] + f_b[0]) / 2.0 <<endl;

    // El punto medio es una aproximación del cruce con el eje Y
    result[0]=(a[0] + b[0]) / 2.0;
    result[1]=(a[1] + b[1]) / 2.0;
    f_result[0] = 0.0; // La intersección ocurre en x = 0
    f_result[1] = (f_a[1] + f_b[1]) / 2.0; // Aproximación de y
    return num_prop;
}

//calculo del vector propio mediante la aplicacion algebraica(DF*v=u) y
```

```
//la propagacion de DF

void vector_prop(mpreal solution[2], mpreal DF[2][2], mpreal tan[2], mpreal f_tan[2],
int num_prop, mpreal meps){
        mpreal norm;

        for(int i=0; i<num_prop;i++){
                update_F(solution,DF, 1, meps);
                multiply_matrices(DF, tan, f_tan);
                tan[0]=f_tan[0];
                tan[1]=f_tan[1];

        }
        norm= sqrt(f_tan[0] * f_tan[0] + f_tan[1] * f_tan[1]);
        f_tan[0]=f_tan[0]/norm;
        f_tan[1]=f_tan[1]/norm;

        cout << "Converged to vector: f_vec = [ " << f_tan[0] << " , " << f_tan[1] <<
    "]\n" <<endl;
        return;

}

//formula para comprobar el angulo con el paper

void comprovacio_angle(mpreal meps){
        mpreal angle,mmu;
        mmu=pow(meps,1);
        angle = M_PI/(2*meps) * mmu/cosh(M_PI/(2*meps));

        cout << "Angle de diferencia entre separatrius segons el abstract: " << angle
    << " * (1 + O( " << mmu << ", "<< meps*meps <<")" <<endl;

}

//funcion F(h) del metodo de richardson

void calcul_rich(mpreal entrada[2], mpreal DF[2][2], int num_prop, mpreal h,
mpreal result[2], mpreal meps){
        mpreal solutionp[2], solutionm[2];
        solutionp[0]=entrada[0]+h;
        solutionp[1]=entrada[1]+h;
        solutionm[0]=entrada[0]-h;
        solutionm[1]=entrada[1]-h;

        for(int i=0; i<num_prop;i++){
                update_F(solutionp,DF, 1, meps);
                update_F(solutionm,DF, 1, meps);
        }
```

```
        result[0]=(solutionp[0]-solutionm[0])/(2*h);
        result[1]=(solutionp[1]-solutionm[1])/(2*h);
        return;
}

//funcion para calcular el vector propio mediante el metodo de Richardson

void richardson(mpreal solution[2], mpreal DF[2][2], int num_prop, mpreal vec_tan[2],
mpreal meps){

        mpreal tolerance, solutionp[2], solutionm[2], F_1[2], F_2[2], F_1h[2], F_2h[2];
        int k=mpreal::get_default_prec();
        tolerance=pow(0.5,k);

        //F_1(h)
        calcul_rich(solution, DF, num_prop, tolerance, F_1, meps);

        //F_1(2*h)
        calcul_rich(solution, DF, num_prop, 2*tolerance, F_1h, meps);

        //F_2(h)
        F_2[0]=F_1[0]+(F_1[0]-F_1h[0])/3;
        F_2[1]=F_1[1]+(F_1[1]-F_1h[1])/3;

        //F_2(2h)
        calcul_rich(solution, DF, num_prop, 2*tolerance, F_1, meps);
        calcul_rich(solution, DF, num_prop, 2*tolerance, F_1h, meps);
        F_2h[0]=F_1[0]+(F_1[0]-F_1h[0])/3;
        F_2h[1]=F_1[1]+(F_1[1]-F_1h[1])/3;

        //Calcul F_3(h)
        vec_tan[0]=F_2[0]+(F_2[0]-F_2h[0])/15;
        vec_tan[1]=F_2[1]+(F_2[1]-F_2h[1])/15;

        mpreal norm = sqrt(vec_tan[0] * vec_tan[0] + vec_tan[1] * vec_tan[1]);
        vec_tan[0] /= norm;
        vec_tan[1] /= norm;
        cout << "Converged to vector: f_vec = [ " << vec_tan[0] << " , " << vec_tan[1]
    << "]\n" <<endl;

}

//funcion para calcular el angulo entre el vector y el eje de las x

mpreal angle_cos(mpreal v[2]){

        mpreal num,den;
        num=v[0];
```

```
        den=sqrt(v[0] * v[0] + v[1] * v[1]) * sqrt(1);
        return num/den;
}

//eje principal del programa

int main() {
        mpreal::set_default_prec(mpfr::digits2bits(DIGITS));
        mpreal DF[2][2], F[2], solution[2], p1[2], h, p2[2],
    vacia[2][2]={{0.0, 0.0}, {0.0, 0.0}}, f_solution[2];
        mpreal vap1, vap2;
        mpreal meps;
            mpreal vep1[2], vep2[2];
            mpreal tan[2], f_tan[2], angle, vec_rich[2];
            int num_prop, k;
            k=mpreal::get_default_prec();
            meps=0.1;
            h=pow(0.5,k/2);
            printf("%d\n",k);
            cout << "Paso para calcular variedades: " << h <<endl;
            printf("Parte del punto (-3.14,0)\n");

            //Calculamos solucion de F(x)-x para encontrar puntos fijos
        //del sistema(orbitas periodicas)
            newton(solution,DF, F, meps);
            cout << "Converged to solution: x = [ " << solution[0] << " , "
        << solution[1] << "]" <<endl;

            //Imprimimos F
        printf("Matriz F(x):\n");
        cout << F[0] <<endl<< F[1] <<endl;

        //Imprimimos matriz DF
        printf("Matriz DF:\n");
        cout << DF[0][0] << "    " << DF[0][1] << "\n" <<DF[1][0]<<"    "<<
        DF[1][1]<<endl;

        //Calculamos vaps
        vaps(DF, &vap1, &vap2);
        printf("Valores propios:\n");
        cout<<"Vap1:"<<vap1<<" \n";
        cout<<"Vap2:"<<vap2<<" \n";
        mpreal modulo1 = calculate_modulus(vap1);
        mpreal modulo2 = calculate_modulus(vap2);
        printf("modulos de los valores propios:\n");
        cout<<"Vap1:"<<modulo1<<"\n";
        cout<<"Vap2:"<<modulo2<<"\n";

        //Decimos si son estables o inestables
```

```
        //Punto estable(atractivo)
        if(modulo1<1 && modulo2<1){printf("Punto estable\n");}

        //Punto inestable
        else if(modulo1>1 || modulo2>1 ){printf("Punto inestable\n");}

        else if (modulo1==1 && modulo2==1){printf("Punto centrico\n");}

        //Imprimimos veps asociados a los vaps
        cout <<"Vector propio asociado al vap( " << vap1 << " ):\n";

        calcula_veps(DF, vap1, vep1);

        cout << "(" << vep1[0] << " , " << vep1[1] << ")" <<endl;

        cout <<"Vector propio asociado al vap( " << vap2 << " ):\n";

        calcula_veps(DF, vap2, vep2);

        cout << "(" << vep2[0] << " , " << vep2[1] << ")" <<endl;

        //Calculamos intervalo fundamental

        //Calculamos p1=p0 + h*v
        p1[0]=solution[0]+h*(-vep1[0]);
        p1[1]=solution[1]+h*(-vep1[1]);

        //Calcumos p2
update_F(p1,vacia,1,meps);

        p2[0]=p1[0];
        p2[1]=p1[1];
        p1[0]=solution[0]+h*(-vep1[0]);
        p1[1]=solution[1]+h*(-vep1[1]);
        tan[0]=p2[0]-p1[0];
        tan[1]=p2[1]-p1[1];

        //Imprimimos p1 y p2, este es el intervalo fundamental con el segmento recto
        printf("\n\nIntervalo fundamental:\n");

        cout <<"Punto P1: ( " << p1[0] << " , " << p1[1] << " )\n";

        cout <<"Punto P2: ( " << p2[0] << " , " << p2[1] << " )\n";

        //Bucle para aplicar el metodo de la biseccion y la propagacion del intervalo
        printf("\n\nEntramos en el metodo de la biseccion:\n");

num_prop=bisection_on_variety(p1, p2, vacia, f_solution, solution, meps);
```

```
        cout <<"Converged to solution: sol = [ " << solution[0] << " , "
        << solution[1] << " ]\n";

        cout <<"Converged to imaged solution: f_sol = [ " << f_solution[0] << " , "
        << f_solution[1] << " ]\n\n";

        cout <<"Vector tangente: vec = [ " << tan[0] << " , " << tan[1] << " ]\n";

        //Calculamos el vector tangente en el eje y mediante extrapolacion de Richardson

        richardson(solution, DF, num_prop,vec_rich, meps);

        //Calculo del angulo
angle=2*acos(angle_cos(vec_rich));

        cout <<"Angle de diferencia entre separatrius segons el metode de derivacio
    Richardson i el cos: " << angle <<"\n";

angle=2*atan2(vec_rich[1],vec_rich[0]);

        cout <<"Angle de diferencia entre separatrius segons el metode de derivacio
    Richardson i la tan: " << angle <<"\n\n";

        //Calculo de la imagen del vector tangente vector tangente en el eje y,
        //iteramos tantas veces como se ha iterado para encontrar el intervalo
        //fundamental
        cout <<"Vector tangente: vec = [ " << tan[0] << " , " << tan[1] << " ]\n";

        vector_prop(solution, DF, tan, f_tan, num_prop, meps);

        //Calculo del angulo angle=2*acos(angle_cos(f_tan));

        cout <<"Angle de diferencia entre separatrius segons el metode de propagacio
    i el cos: " << angle <<"\n";

        angle=2 * atan2(f_tan[1],f_tan[0]);

        cout <<"Angle de diferencia entre separatrius segons el metode de propagacio
    i la tan: " << angle <<"\n\n";

        comprovacio_angle(meps);

        return 0;
    }
```

Compilación

Para la compilación del programa se necesitan varios archivos.

En primer lugar, se requiere un archivo de texto plano que contenga el sistema de EDOs para crear el integrador numérico mediante el paquete de software de **Taylor**. En nuestro caso concreto, lo hemos llamado *tfg.eqs*.

El siguiente paso sería crear el integrador numérico específico para ese sistema de EDOs. Para ello, utilizamos el comando:

```
taylor -name tfg -o tfg.c -jet -step tfg.eqs
```

donde la flag $-name$ indica el nombre de la función que ejercerá como integrador número-rico, en nuestro caso se llamará $taylor\_step\_tfg$.

Por otra parte también tendremos que generar el archivo *taylor.h* que será la biblioteca donde se encontrarán todas las funciones internas del integrador.Para ello se ejecutará el comando:

```
taylor -name tfg -o taylor.h -mpfr -header tfg.eqs
```

Finalmente tras generar estos archivos ya podremos generar el ejecutable mediante la compilación de archivos "linkados". Para ello utilizaremos el siguiente comando:

```
g++ -O2 principal_mpfr.cc tfg.c -o ejecutable -lmpfr -lgmp -lm
```

donde principal_mpfr.cc es el programa principal donde se combina al resto de programas.

Finalmente para su ejecución solo quedaría ejecutar el siguiente comando en la terminal:

```
./ejecutable
```