Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías Industriales

Resolución numérica de ecuaciones elasto-plásticas para fatiga multiaxial

Autor: Julio Sánchez García

Tutor: Alfredo de Jesús Navarro Robles

Dpto. Ingeniería Mecánica y Fabricación
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2021

Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías Industriales

Resolución numérica de ecuaciones elasto-plásticas para fatiga multiaxial

Autor:
Julio Sánchez García

Tutor:
Alfredo de Jesús Navarro Robles
Catedrático de Universidad

Dpto. Ingeniería Mecánica y Fabricación
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2021

Trabajo Fin de Grado: Resolución numérica de ecuaciones elasto-plásticas para fatiga multiaxial

Autor: Julio Sánchez García
Tutor: Alfredo de Jesús Navarro Robles

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

Agradecimientos

A quienes me han apoyado siempre.

Julio Sánchez García
Sevilla, 2021

Resumen

El objeto de este TFG es desarrollar un programa de cálculo numérico que resuelva las ecuaciones elasto-plásticas de un Método de Deformaciones Locales para fatiga multiaxial utilizando el entorno de MATLAB.

El programa diseñado, que se basa en Programación Orientada a Objetos, recibe como entrada el historial de deformaciones de una probeta cilíndrica hueca de pared delgada y devuelve las tensiones como salida. Con este fin, debe ejecutar algoritmos de descarga y memoria análogos a los empleados en el Método de Deformaciones Locales uniaxial. La resolución del Sistema de Ecuaciones Diferenciales Ordinarias se lleva a cabo empleando un método de Runge-Kutta con dos pares encajados y de paso variable. Una vez realizado el cálculo, el programa almacena toda la información del historial de cargas de la probeta y representa los resultados obtenidos de manera gráfica y en tiempo real.

Abstract

The aim of this project is to desing a numerical calculation programme able to resolve elasto-plastic flow equations for a multiaxial Local Strain-Life Method using the MATLAB software.

The programme designed, which makes use of Object-Oriented Programming, recieves the strain record of a thin-walled cylindrical test tube and displays the stresses once it is executed. In order to achieve this goal, it must perform unloading and memory algorithms similar to those used in the Local Strain-Life Method for uniaxial cases. The solution to the system of Ordinary Differential Equations is calculated by using two adaptative stepsize Runge-Kutta interwoven methods. Once the results have been obtained, the programme stores all the loading information and shows the final outcome in a real time graphical representation.

Índice

Notación

E	Módulo de Young
K	Constante de esfuerzos de Ramberg-Osgood
n	Exponente de endurecimiento de Ramberg-Osgood
ν	Coeficiente de Poisson
G	Módulo de cizalladura
ε	Componente normal de la deformación
γ	Componente tangencial de la deformación
$\boldsymbol{\varepsilon}$	Vector de deformaciones
q	Distancia efectiva entre puntos de tensiones
Q	Módulo del vector de tensiones para cargas
σ	Componente normal de la tensión
τ	Componente tangencial de la tensión
$\boldsymbol{\sigma}$	Vector de tensiones
θ	Ángulo entre vectores de carga
$\boldsymbol{\Phi}()$	Módulo de endurecimiento del material
$\phi()$	Módulo de endurecimiento del material tras descargas
$\boldsymbol{\sigma}_k$	Puntos de inversión de carga
$\boldsymbol{\sigma}_{c,k}$	Centros de las sucesivas circunferencias de cargas
$\boldsymbol{\sigma}'$	Tensiones inmediatamente posteriores a $\boldsymbol{\sigma}$
$\boldsymbol{\sigma}''$	Tensiones interpoladas entre $\boldsymbol{\sigma}$ y $\boldsymbol{\sigma}'$
t_{σ_0}	Instante de tiempo en el que se da el valor $\boldsymbol{\sigma}_0$
t_{def}	Vector de instantes de tiempo de entrada
t_{tens}	Vector de instantes de tiempo de salida
n_{def}	Longitud de los vectores de datos de entrada
n_{tens}	Longitud de los vectores de datos de salida
Δt	Incremento temporal
$\|\mathbf{v}\|$	Norma del vector $\mathbf{v}$
O	Orden
PVI	Problema de Valores Iniciales
EDO	Ecuación Diferencial Ordinaria
EDOs	Ecuaciones Diferenciales Ordinarias

1 Introducción

1.1 Motivación y objeto del proyecto

La finalidad de este Trabajo de Fin de Grado es programar en MATLAB un código que permita calcular, a partir del registro de deformaciones nominales de entrada, la evolución temporal de las componentes del vector de tensiones en un punto de una probeta sometida a esfuerzos de fatiga multiaxial a bajo número de ciclos. Las deformaciones locales consideradas, debido a la situación de estudio, tendrán componentes tanto elásticas como plásticas.

Para ello, el programa ha de resolver un sistema de ecuaciones diferenciales al mismo tiempo que detecta y almacena los ciclos de carga que se dan en el material según el Método de Deformaciones Locales desarrollado en el artículo *Plasticity theory for the multiaxial Local Strain-Life Method* [6].

Por último, se debe generar una salida gráfica que facilite el análisis por parte del usuario de los resultados obtenidos en tiempo real.

1.2 Concepto de fatiga

La fatiga es un fenómeno que se da en materiales sometidos a esfuerzos variables debido al cual se producen fallos mecánicos a un nivel de carga menor al esperado en un caso estático. Los efectos de la fatiga se ponen de manifiesto siempre que haya variaciones en las tensiones y deformaciones en algún punto del componente durante su puesta en servicio.

Considerando que más del setenta por ciento de los fallos que se producen en elementos de máquinas son debidos a la fatiga, es de vital importancia estudiar sus causas y diseñar consecuentemente. Para ello, es necesario tener una estimación de la vida de la pieza (en ciclos de carga) y utilizar un modelo de cálculo que reproduzca el comportamiento del material.

Uno de los modelos más usados desde que se realizan estudios de fatiga es la Curva S-N, como la de la figura 1.1. Cada uno de los puntos de esta curva representa el número de ciclos de carga que puede soportar un material antes romperse dado un valor de tensión nominal S. Esta función puede obtenerse de manera experimental mediante ensayos mecánicos o bien estimarse mediante la tensión de rotura del material junto con otros factores, como el acabado superficial, el tipo de carga o el concentrador efectivo de tensiones de la pieza. Con este método es posible diseñar elementos mecánicos si se conoce el número de ciclos de carga que estos deberán soportar a lo largo de su puesta en servicio.

Sin embargo, la curva S-N tiene ciertas limitaciones que hacen que este procedimiento de diseño sea poco fiable en determinadas circunstancias. Según las condiciones de servicio, algunos ensayos experimentales demuestran que existe dispersión entre el pronóstico de vida arrojado por la curva y los resultados reales.

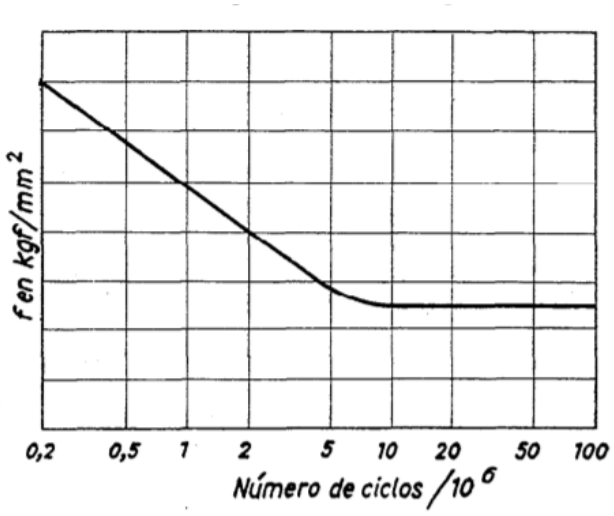

Figura 1.1 Ejemplo de curva S-N [2] .

Una de sus principales limitaciones se produce en los casos en los que las tensiones alcanzan valores elevados y el fallo mecánico se produce a bajo número de ciclos. En estas situaciones de fatiga se usan métodos basados en la curva $\varepsilon - N$ (como la de la figura 1.2) en lugar de la curva S-N. En este tipo de formulaciones se establece una separación entre los periodos de nucleación y propagación, diferenciándose así de la curva S-N, que considera la fatiga como un fenómeno de una sola etapa [10].

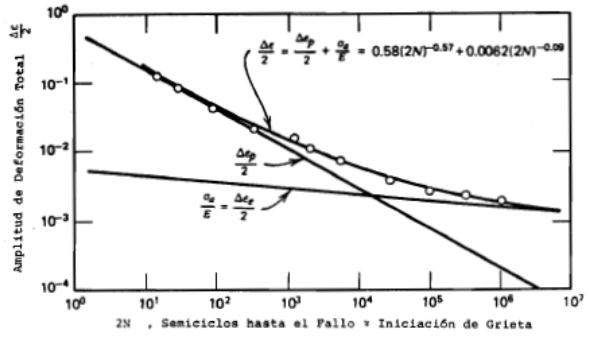

Figura 1.2 Curva $\varepsilon - n$ para el Método de las Deformaciones Locales [10].

Adicionalmente, el estudio de la fatiga se vuelve todavía más complicado ante la existencia de cargas multiaxiales, existiendo una combinación de tensiones y deformaciones normales y tangenciales, todas variables, en un mismo punto del material.

Actualmente, el campo de la fatiga está en continua evolución y trata de desarrollar nuevos modelos teóricos que proporcionen unos criterios de diseño cada vez más precisos a los ingenieros que trabajan con componentes mecánicos sometidos a cargas variables.

1.2.1 El Método de las Deformaciones Locales

El Método de las Deformaciones Locales es uno de los procedimientos más usados para realizar cálculos de fatiga uniaxial a bajo número de ciclos. Este método se basa en la curva $\varepsilon - N$ y, por

tanto, es necesario conocer las deformaciones producidas en el punto crítico del material. Estas deformaciones máximas en la pieza se producirán en entallas o en otro tipo de concentradores de tensiones [10].

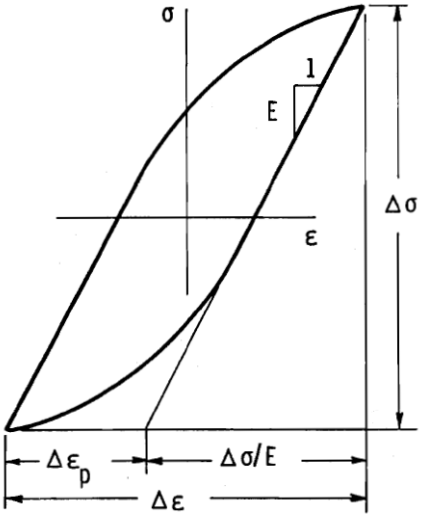

Figura 1.3 Curva $\varepsilon - \sigma$ para el Método de las Deformaciones Locales para fatiga uniaxial [10].

Para poder llegar a conocer las amplitudes de las deformaciones en la pieza, se realiza un análisis de los lazos de carga en la entalla, conociendo previamente los esfuerzos sobre la pieza, el concentrador de tensiones y su geometría.

Con el fin de obtener las deformaciones en el concentrador a partir de las tensiones, se suele utilizar la Regla de Neuber conjuntamente con un modelo que represente el comportamiento del material. Así pues, considerar al material como elasto-plástico perfecto o utilizar la ecuación de Ramberg-Osgood serían dos posibles ejemplos de modelos de comportamiento.

Siguiendo estos criterios, se modela un diagrama $\sigma - \varepsilon$ como el de la figura 1.3, donde se muestran los lazos de histéresis debidos al ciclo de cargas en la pieza. Finalmente, una vez conocidas las amplitudes de los ciclos de deformaciones, se estima la vida del material usando algunos de los modelos $\varepsilon - N$, como puede ser el Método SWT.

1.3 Método de las Deformaciones Locales para fatiga multiaxial

A pesar de que el Método de Deformaciones Locales está enormemente extendido para el estudio de casos de fatiga a bajo número de ciclos, aún no se ha conseguido ampliar al caso multiaxial con el grado de simplicidad deseado.

Para conseguir este objetivo sería necesario, en primer lugar, desarrollar una serie de reglas que reproduzcan la manera en la que se trabaja con ciclos de histéresis y el efecto de memoria como en el caso uniaxial. El segundo paso consistiría en elaborar una fórmula semejante a la de Neuber para tratar con deformaciones inelásticas y entallas. Por último, habría que proponer un método de conteo de ciclos y un criterio de cálculo de vida.

En el artículo *Plasticity theory for the multiaxial Local Strain-Life Method* [6] se desarrolla una teoría cuyo objetivo es proporcionar a los diseñadores un método de cálculo para fatiga multiaxial lo más parecido posible al usado para el caso uniaxial. Esta teoría se centra en la resolución del primero de los dilemas mencionados en el párrafo anterior.

El artículo expone un desarrollo similar a la regla de memoria empleada cuando se cierran ciclos de histéresis en el caso uniaxial. A partir de lo cual, se podría definir un método de conteo para amplitudes de carga variable multiaxial.

Para ello, se utiliza como base la idea de distancia entre puntos de tensiones, calculada a partir de la expresión del criterio de plastificación. Al contrario de lo que ocurre en otro tipo de teorías cíclicas de plasticidad, no se utilizan superficies de plastificación o de cargas que se mueven en el espacio de tensiones [6].

1.4 Ensayo de fatiga

Antes de abordar el funcionamiento del código, es necesario conocer el contexto real que sirve como base para el desarrollo del programa, así como la información que se va a procesar. Para ello, en este apartado se describe el ensayo de fatiga que proporciona los datos e hipótesis que constituyen la base del diseño.

Como ocurre con otro tipo de ensayos mecánicos, los ensayos a fatiga se realizan en laboratorios con maquinaria especializada y siguiendo las Normas correspondientes. Las máquinas de ensayo a fatiga, como las que existen en el laboratorio de Ingeniería Mecánica de ETSI de Sevilla (figura 1.4), cuentan con la capacidad de generar varios tipos de situaciones de carga y medir los resultados finales. En cada uno de estos ensayos, se pueden modificar parámetros como la amplitud o la simetría de tensiones para obtener distintos resultados y adaptarlos a circunstancias reales. La norma UNE 7118:1958 [2] clasifica los ensayos de fatiga en cuatro grupos:

- **Ensayo de flexión rotativa**: Se aplica un flector a una probeta cilíndrica que gira sobre su eje.
- **Ensayo de flexión plana**: Aplicación de un flector alternativo a una probeta de sección rectangular.
- **Ensayo de tracción compresión**: Un axil variable se ejerce sobre una probeta de sección indefinida.
- **Ensayo de torsión**: Se aplica un momento torsor variable a la probeta cilíndrica.
- **Ensayos especiales**: Aquí se engloban un conjunto heterogéneo de ensayos que tratan de determinar otro tipo de características del material. A este grupo pertenecen los ensayos multiaxiales como el que se va a analizar en este proyecto.

En el caso de estudio de este proyecto, el programa debe recibir como únicas entradas las deformaciones normales y tangenciales en un punto de una probeta cilíndrica hueca de espesor despreciable así como el intervalo temporal en el que se producen dichas deformaciones. Estas vienen dadas por el control de deformaciones de la máquina de ensayos de fatiga multiaxial en la que se colocaría la probeta bajo estudio.

Es necesario comentar que no ha sido objeto de este TFG realizar el ensayo de fatiga correspondiente ni modificar o diseñar sus parámetros. Aún así, se ha querido describir este experimento para dar un contexto y poner de manifiesto la situación real sobre la que se aplicarían los resultados obtenidos.

1.4.1 Probeta

En el ensayo a fatiga multiaxial considerado, se colocaría una probeta cilíndrica con un orificio interno a lo largo de todo su eje axial cuyas dimensiones se especifican en la figura 1.5. La geometría externa de la probeta es similar a la utilizada en otro tipo de ensayos de materiales, con los extremos

Figura 1.4 Máquina de ensayo biaxial [14].

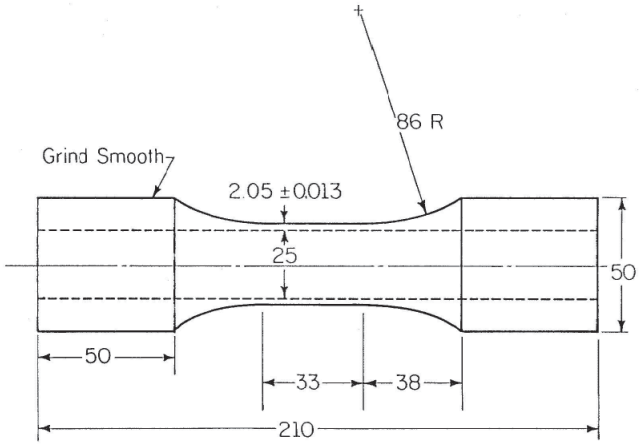

Figura 1.5 Probeta Inconel 718 para ensayos de fatiga multiaxial [1].

de mayor diámetro que la sección central y un radio de acuerdo lo suficientemente grande como para que no se produzcan concentradores de tensiones.

La probeta contaría con una serie de galgas extensiométricas (o se montaría un extensómetro en su defecto) que miden en tiempo real las deformaciones que se producen en cada uno de los puntos donde están colocadas. Un hecho importante que hay que tener en cuenta es que, en este caso, el espesor es lo suficientemente pequeño como para poder considerar las tensiones tangenciales constantes a lo largo del mismo. Por esta razón, el programa solo va a calcular el resultado para un único punto geométrico.

1.4.2 Material

El programa se va a diseñar de manera que el usuario final sea capaz de modificar los valores de las propiedades físicas del material estudiado. Los datos necesarios para el correcto funcionamiento

del código son las constantes que aparecen en la ecuación de Ramberg-Osgood (1.1).

$$\varepsilon = \frac{\sigma}{E} + K\left(\frac{\sigma}{E}\right)^{\frac{1}{n}} \tag{1.1}$$

- **Módulo elástico o módulo de Young** E: Relaciona las tensiones normales en un punto del material con su deformación longitudinal y se determina a partir del ensayo de tracción.
- **Módulo de cizalladura** G: Relaciona las tensiones tangenciales en un punto del material con su deformación transversal. Se puede obtener de forma indirecta mediante el ensayo de tracción usando el módulo de Young y el coeficiente de Poisson. $G = E/(1+2\nu)$.
- **Coeficiente de esfuerzos de Ramberg-Osgood** K: Coeficiente propio de cada material, que se usa en la ecuación de Ramberg-Osgood.
- **Exponente de endurecimiento de Ramberg-Osgood** n: Exponente que depende del grado de endurecimiento de un material.

A priori, se podrían introducir los datos de cualquier material que cumpliese con el modelo de Ramberg-Osgood y al que fuesen aplicables las ecuaciones de comportamiento utilizadas. En el apartado 3.2.1 se explica detalladamente cómo se declaran estas propiedades en el programa y cómo se tratan estos valores en las distintas partes del código.

Para la fase de diseño y en el posterior análisis de resultados se va a emplear la aleación Inconel 718. Los valores de las propiedades físicas de este material son las mostradas a continuación [1].

$$E = 208500\ MPa \quad G = 77800\ MPa \quad K = 1910\ MPa \quad n = 0.08$$

El Inconel 718 es una aleación níquel-cromo que está formada principalmente de Níquel (50-55 %), Cromo (17-21 %), Hierro (balance), Molibdeno (2.8-3.3 %) y Niobio (4.75-5.5 %). Es una aleación que tiene un gran comportamiento ante cargas en entornos corrosivos y a altas temperaturas. Sus principales usos son la fabricación de ejes, diversas partes de motores jet para aviones y sistemas de almacenamiento criogénico [13].

1.4.3 Trayectorias de ensayos

En la figura 1.6 se representan las trayectorias en deformaciones para los ensayos de fatiga multiaxial que se han estudiado en este TFG. Se aprecia que al variar las amplitudes, los valores medios y el desfase entre las componentes del vector de deformaciones, se obtiene una gran variedad de datos.

Como ya se ha comentado, no se han realizado ensayos de fatiga en este proyecto, pero se simularán las deformaciones de entrada generando una serie de archivos de texto con los datos que proporcionaría el control de deformaciones de la máquina de ensayos.

En un principio, se va a estudiar solo un ciclo de cada una de las situaciones, pero es posible que, tras considerar los resultados, sea necesario ampliar el análisis a otro tipo de datos de entrada.

1.5 Programación en MATLAB

MATLAB es un software comercial para cálculo numérico distribuido por la empresa MathWorks, con sede en Natick, Massachusetts (EE.UU.), que es utilizado ampliamente en diversos ámbitos de la Ingeniería y las Ciencias Aplicadas [7].

MATLAB es un lenguaje de alto nivel en el que se pueden realizar gran variedad de operaciones con matrices, vectores y estructuras de datos. Principalmente, se usa para proyectos de cálculo numérico debido a que cuenta con una serie de funciones predefinidas que realizan tareas como construcción de matrices, cuadratura, interpolación de funciones o resolución de PVI. También

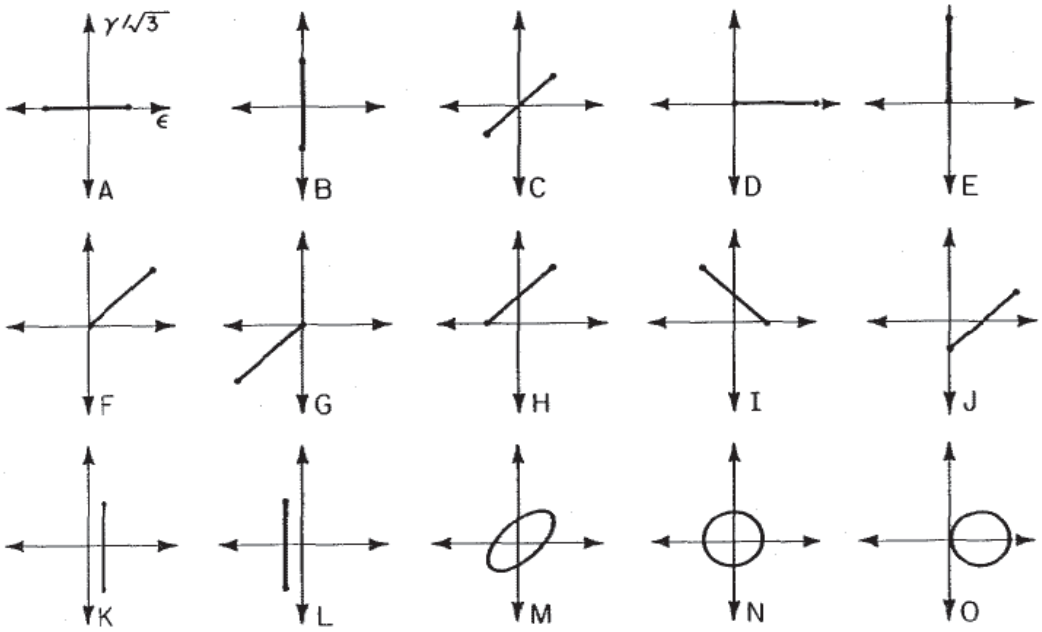

Figura 1.6 Trayectorias en deformaciones para ensayos de fatiga multiaxial [1].

cuenta con herramientas de cálculo simbólico y con otro tipo de módulos como Simulink, que se utiliza para simulación de sistemas. En este trabajo no serán necesarias este tipo de utilidades y solo se hará uso del potencial que tiene el software para cálculos numéricos.

Se ha elegido MATLAB para realizar este proyecto debido a que cuenta con una gran potencia de cómputo y porque permite una enorme versatilidad a la hora de programar y desarrollar funciones específicas. Adicionalmente, es un lenguaje de programación enormemente extendido en diversos ámbitos de la Ingeniería y usado por un gran número de profesionales en todo el mundo. Una muestra de ello es que gran parte de las funcionalidades de este programa son utilizadas y enseñadas en asignaturas de diversa índole en la Escuela Técnica Superior de Ingeniería de Sevilla.

1.5.1 Programación Orientada a Objetos en MATLAB

La Programación Orientada a Objetos (conocida abreviadamente como POO) es un estilo de programación de alto nivel que difiere de la tradicional Programación Estructurada. Surge como una alternativa de programación que ayuda a organizar códigos complejos y a estructurar las funcionalidades y la información que maneja el programa. En los últimos años se ha expandido a numerosos campos, entre los que destacan principalmente el diseño software para ordenadores, las aplicaciones móviles y la programación de páginas web. Varios ejemplos de lenguajes que soportan POO y que en la actualidad son ampliamente populares en los campos mencionados son: JavaScript, Python o PHP.

La POO se basa principalmente en unas estructuras conocidas como clases, a partir de las cuales se crean o definen los objetos. Simplificando el funcionamiento de las clases, estas cuentan con una serie de variables (o propiedades) que guardan la información de cada objeto y de una serie de métodos (o funciones) específicos para cada clase. Al declarar un objeto, se produce una instanciación de una clase, es decir, el objeto cuenta con los campos de propiedades definidos en la clase y es posible realizar modificaciones sobre ellos con los métodos previamente definidos.

En MATLAB es posible utilizar este tipo de programación mediante los archivos tipo "class", en los que se definen las variables como "properties" y los métodos como funciones "methods". También es posible definir atributos específicos a las propiedades y a los métodos, para regular o restringir su uso y funcionalidad en otras partes del programa. Una vez que se tienen definidas las

clases, simplemente se pueden instanciar objetos en cualquier fichero o función para modificar sus propiedades [9].

Durante la realización de este trabajo, se decidió usar este tipo de programación para evitar el uso de variables globales y hacer un código más ordenado y visualmente sencillo. El uso de clases en este proyecto es relativamente simple y no ha sido necesario emplear técnicas avanzadas o atributos inusuales. Esta implementación de la POO y cómo funciona la clase diseñada para el proyecto se explica con detalle en la sección 3.4.

1.6 Estructura del documento

Una vez que se ha presentado el tema a tratar y se han introducido los conceptos básicos (Capítulo 1), se considera oportuno detallar cómo está estructurado el resto del documento.

- **Desarrollos matemáticos** (Capítulo 2): Se explican las formulaciones y algoritmos matemáticos que posteriormente se han incluido en alguna parte del código.
- **Explicación del programa de cálculo numérico** (Capítulo 3): En cada uno de los subapartados de este capítulo se expone paso a paso cómo funcionan todas las partes que conforman el código. Asimismo, se explica cómo se incluyen los desarrollos matemáticos abordados previamente y se muestran varios diagramas de flujo para organizar la información. Por último, se desarrolla el proceso de diseño que se ha llevado a cabo para maximizar la eficiencia y disminuir la aparición de errores.
- **Análisis de resultados** (Capítulo 4): Una vez que se ha tratado el funcionamiento del programa, se introducen los diferentes estados de carga y se procede a presentar los resultados obtenidos.
- **Conclusiones y líneas de ampliación** (Capítulos 5 y 6): Reflexiones realizadas tras desarrollar el trabajo y análisis de posibles mejoras futuras.
- **Manual de usuario** (Apéndice A): En este anexo se muestra a un posible usuario cómo debe introducir la información requerida por el programa y cómo ejecutarlo. También se detalla cómo deben interpretarse las salidas generadas con el fin de que no se produzcan errores o confusiones.

2 Desarrollos matemáticos

En este capítulo se explican las formulaciones y operaciones matemáticas que se han utilizado en algún lugar del programa de cálculo numérico. Es conveniente comentar estos conceptos matemáticos para poder entender el diseño de cada una de las funciones que incluyen estas fórmulas, que se explican en el apartado 3.

2.1 Cálculos del Método de Deformaciones Locales para fatiga multiaxial

Esta sección se centra tanto en el algoritmo detrás del método que se ha implementado en el programa como en el cálculo de las circunferencias de carga y las distancias efectivas de tensiones. Toda la información que se presenta en este apartado, incluidas las figuras, ha sido extraída de la publicación *Plasticity theory for the multiaxial Local Strain-Life Method* [6].

En todo momento, las ecuaciones para flujo plástico en el caso de cargas biaxiales tensión-torsión vienen dadas por la expresión (2.1) [11]. Según se trate de cargas o descargas, la manera de calcular la matriz $A(\sigma,\tau)$ y la distancia efectiva de tensiones será diferente.

$$\begin{bmatrix} \frac{d\sigma}{dt} \\ \frac{d\tau}{dt} \end{bmatrix} = \frac{d\sigma}{dt} = A(\sigma)\frac{d\varepsilon}{dt} = A(\sigma,\tau)\begin{bmatrix} \frac{d\varepsilon}{dt} \\ \frac{d\gamma}{dt} \end{bmatrix} \tag{2.1}$$

2.1.1 Ecuaciones elasto-plásticas referidas al origen

En este apartado se hace referencia a las ecuaciones elasto-plásticas usadas para la primera aplicación de las cargas y para un posterior caso en el que, tras la aplicación de descargas, el punto supere a la última circunferencia y vuelva a tener como referencia el origen [11].

$$A(\sigma,\tau) = \frac{1}{\frac{1}{E}\frac{1}{G} + \Phi(Q)\left[\frac{4n_2^2}{E} + \frac{n_1^2}{G}\right]} \begin{bmatrix} \frac{1}{G} + 4n_2^2\Phi(Q) & -2n_1n_2\Phi(Q) \\ -2n_1n_2\Phi(Q) & \frac{1}{E} + n_1^2\Phi(Q) \end{bmatrix} \tag{2.2}$$

$$n_1 = \frac{2}{3}\sigma\frac{1}{Q} \tag{2.3}$$

$$n_2 = \tau\frac{1}{Q} \tag{2.4}$$

Para el caso de cargas, el valor $Q = q/2$ viene dado por el módulo del vector de tensiones en el punto, y el módulo de endurecimiento del material por la ecuación (2.6). Así, se van calculando

valores sucesivos de Q a medida que evolucionan las tensiones hasta el punto en el que $dQ/dt < 0$, momento en el que se detecta descarga y hay que utilizar las ecuaciones del apartado siguiente [6].

$$Q = |\sigma| = \sqrt{2\tau^2 + \frac{2}{3}\sigma^2} \tag{2.5}$$

$$\Phi(Q) = \left(\sqrt{\frac{3}{2}}\right)^{(\frac{1}{n}+1)} \frac{1}{n}\left(\frac{Q}{K}\right)^{\frac{1}{n}} \frac{1}{Q} \tag{2.6}$$

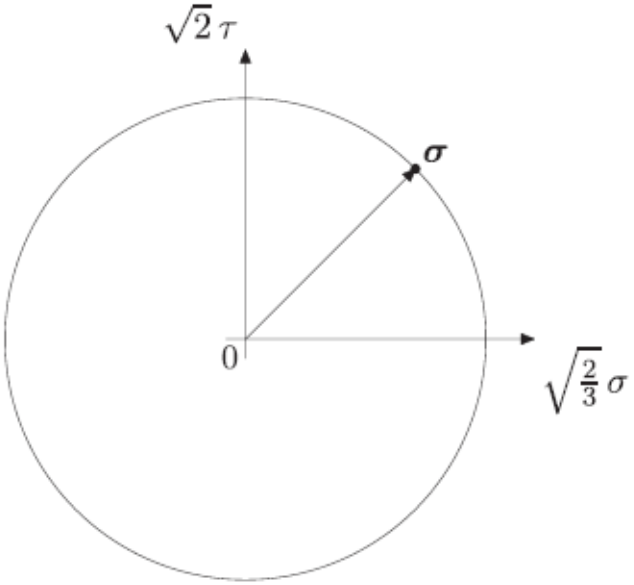

Figura 2.1 Proceso inicial de cargas [6].

2.1.2 Ecuaciones elasto-plásticas para descargas

Las ecuaciones mostradas en esta sección son válidas una vez que se hayan detectado descargas y siempre que no se vuelva a sobrepasar el primer punto de inversión. Justo en el momento en el que se produce una descarga, se genera una nueva circunferencia y los cálculos se realizan usando el último valor de σ_k de referencia [6]. A partir de ahora, la matriz $A(\sigma,\tau)$ viene dada por la expresión (2.7) [12].

$$A(\sigma,\tau) = \frac{1}{\frac{1}{E}\frac{1}{G} + \frac{\phi(q)}{cos\theta}\left[\frac{4n_2^2}{E} + \frac{n_1^2}{G}\right]} \begin{bmatrix} \frac{1}{G} + 4n_2^2\frac{\phi(q)}{cos\theta} & -2n_1n_2\frac{\phi(q)}{cos\theta} \\ -2n_1n_2\frac{\phi(q)}{cos\theta} & \frac{1}{E} + n_1^2\frac{\phi(q)}{cos\theta} \end{bmatrix} \tag{2.7}$$

$$n_1 = \frac{2}{3}(\sigma - \sigma_k)\frac{2}{q} - \frac{2}{3}(\sigma_{c,k} - \sigma_k)\frac{2}{q_k} \tag{2.8}$$

$$n_2 = (\tau - \tau_k)\frac{2}{q} - (\tau_{c,k} - \tau_k)\frac{2}{q_k} \tag{2.9}$$

Desde este instante, las expresiones para calcular q y el módulo de endurecimiento son diferentes. En la figura 2.2 se representa el camino que recorren las tensiones tras producirse dos descargas

sucesivas, la primera en σ_0 y la segunda en σ_1.

$$q = \frac{|\sigma - \sigma_k|}{cos\theta} = \frac{\frac{2}{3}(\sigma-\sigma_k)^2 + 2(\tau-\tau_k)^2}{\frac{2}{3}(\sigma-\sigma_k)(\sigma_{c,k}-\sigma_k) + 2(\tau-\tau_k)(\tau_{c,k}-\tau_k)} \frac{q_k}{2} \tag{2.10}$$

$$cos\theta = \frac{\frac{2}{3}(\sigma-\sigma_k)(\sigma_{c,k}-\sigma_k) + 2(\tau-\tau_k)(\tau c,k-\tau_k)}{\sqrt{\frac{2}{3}(\sigma-\sigma_k)^2 + 2(\tau-\tau_k)^2}\frac{q_k}{2}} \tag{2.11}$$

$$\phi(q) = 2^{1-\frac{1}{n}} \left(\sqrt{\frac{3}{2}}\right)^{\frac{1}{n}+1} \frac{1}{n} \left(\frac{q}{K}\right)^{\frac{1}{n}} \frac{1}{q} \tag{2.12}$$

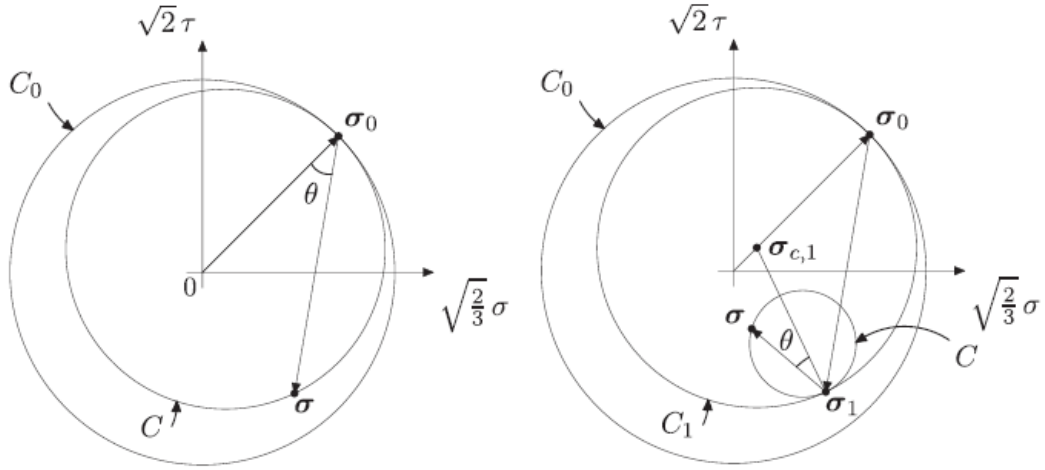

Figura 2.2 Primera y segunda descargas [6].

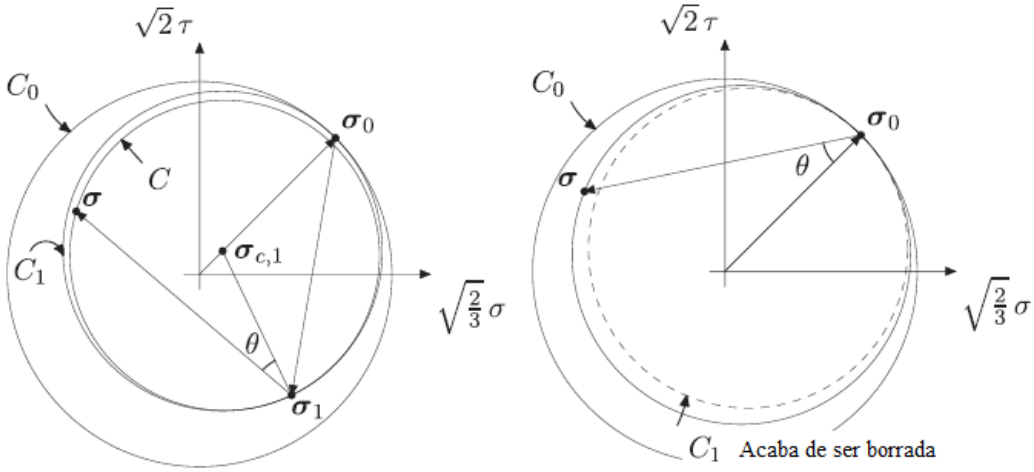

Figura 2.3 El punto de tensiones sobrepasa el valor de C_1 y el efecto de memoria se pone de manifiesto [6].

Para finalizar la explicación de la teoría, resta detallar cómo se produce el efecto de memoria en el material. Este fenómeno ocurre en el momento en el que el valor de q actual sea mayor que el último valor q_k calculado. Esto se ve gráficamente en la figura 2.3, donde σ sobrepasa a la circunferencia C_1, que es "borrada" del historial. A partir de este instante, las referencias usadas para calcular los

valores sucesivos de q son q_{k-1}, σ_{k-1} y $\sigma_{c,k-1}$. Si se diera el caso en el que σ sobrepasa el valor de C_0, es decir, la primera circunferencia, habría que volver a utilizar las ecuaciones de carga [6].

2.2 Cálculo vectorial del centro de las circunferencias de cargas

Cada vez que se detecta una descarga cuando $dq/dt < 0$, es necesario calcular una nueva circunferencia y, por tanto, un nuevo centro $\boldsymbol{\sigma}_{\mathbf{c,k}}$. Estos centros se calculan mediante la suma vectorial representada en la figura 2.4, que se traduce matemáticamente en la expresión (2.13). Para hallar el centro, se suma al punto σ_0 un vector que tenga la misma dirección y sentido que el vector $\sigma_{c,k} - \boldsymbol{\sigma}_{\mathbf{k}}$, pero cuyo módulo sea igual al último valor de $q/2$ antes de detectar la descarga.

$$\sigma_{c,k+1} = \frac{\sigma_{c,k} - \boldsymbol{\sigma}_{\mathbf{k}}}{||\sigma_{c,k} - \boldsymbol{\sigma}_{\mathbf{k}}||} \frac{q_{k+1}}{2} \tag{2.13}$$

$$||\sigma_{c,k} - \boldsymbol{\sigma}_{\mathbf{k}}|| = \sqrt{\frac{2}{3}(\sigma_{c,k} - \sigma_k)^2 + 2(\tau_{c,k} - \tau_k)^2} \tag{2.14}$$

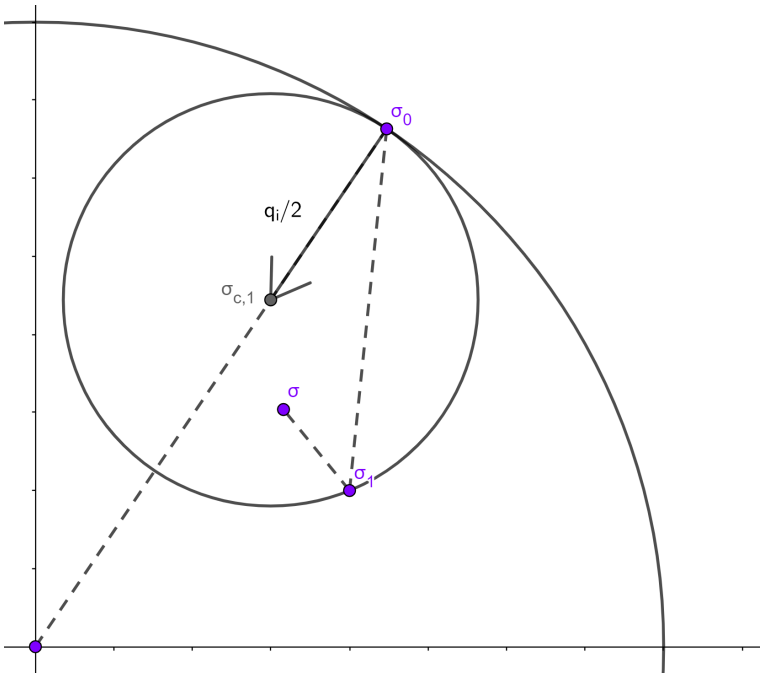

Figura 2.4 Suma vectorial para el cálculo de centros de circunferencias de carga.

Estas expresiones se introducen dentro de la sección de detección de descargas, en el bucle de integración del programa para realizar el cálculo en cada inversión. Para que se realice de manera correcta, hay que guardar todos los datos que se necesitan en la expresión (2.2) en algún lugar de la memoria de cargas.

2.3 Método de la bisectriz para interpolación del vector de tensiones

Como se describió en la explicación de la teoría para fatiga multiaxial, cuando el valor actual de q crece en cada paso de integración hasta llegar a ser lo suficientemente grande como para superar al último valor guardado de q_k, se aplica memoria de cargas. A partir de ese momento, es necesario "olvidar" los datos de la última circunferencia con el objetivo de tener un nuevo punto de referencia para los cálculos.

Cuando esto ocurre, se requiere implementar un algoritmo de retracción para calcular, interpolando la trayectoria de evolución de las cargas, el valor del vector de tensiones en el que se obtendría ese mismo valor de q_k guardado en memoria.

La figura 2.5 ilustra el fenómeno que se acaba de describir. Las componentes del vector tensión han realizado una evolución por el camino $O-\sigma_0-\sigma_1-\sigma$. En el siguiente instante de integración, se pasa directamente del punto σ al punto σ' y, debido a que existe un intervalo de integración finito, sería casi imposible llegar exactamente a σ''. Durante el paso $\sigma-\sigma'$, el algoritmo de memoria detectaría que se ha obtenido un q mayor al último valor de q_k y se realizaría una llamada a la función que interpole el valor σ''.

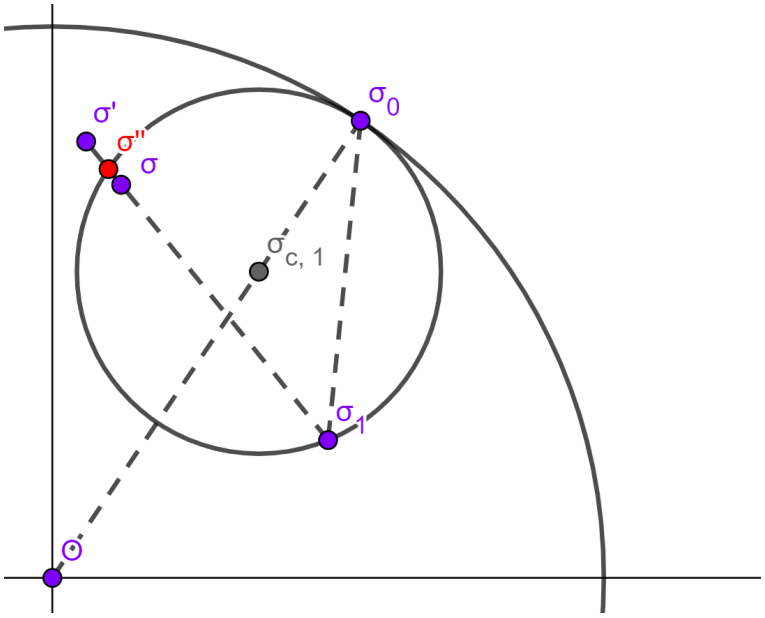

Figura 2.5 Detección de memoria y retracción.

Para realizar esta tarea, se va a suponer que el camino entre los puntos sucesivos (σ y σ' en el ejemplo) es rectilíneo debido a que se considera que el paso de integración es lo suficientemente pequeño. Así, se obtendrá una función para los valores de la distancia efectiva de tensiones que solo va a depender de una variable en lugar de dos, ya que se ha impuesto la condición de contorno (ecuación (2.15)) por la que las tensiones deben pertenecer a la línea que une los puntos σ y σ'. Esto permite reducir el problema a una variable, teniendo el valor de τ como función de σ. Por último, hay que tener en cuenta que en los casos de tensión tangencial pura ($\sigma=0$) se interpolará solo con τ debido a que todas las componentes ($\sigma=0$, $\sigma'=0$ y $\sigma''=0$) son cero.

$$\tau''(\sigma'') = \tau + \frac{(\tau'-\tau)}{\sigma'-\sigma}(\sigma''-\sigma) \tag{2.15}$$

$$q_{buscado} - q(\sigma'', \tau''(\sigma'')) = 0 \tag{2.16}$$

En primer lugar, se ha intentado usar funciones de MATLAB como "fzero" o "fsolve" para hallar la solución al problema no lineal de la expresión (2.16). Estas funciones no pueden ser empleadas debido a que q puede tener valores negativos o positivos cuando se sobrepasa el valor de la última circunferencia en memoria. Las funciones preprogramadas de MATLAB dejan de funcionar, ya que requieren que el intervalo tenga extremos de diferente signo y simplemente devuelven errores de compilación (caso de "fzero") o no convergen al punto deseado el 100 % de las veces (caso de

"fsolve"). En consecuencia, es necesario diseñar manualmente un algoritmo que sea posteriormente implementado en una función que realice esta tarea.

Para esta función se ha decidido utilizar una versión del método de la bisectriz. Como se comentó anteriormente, el problema no cumple el Teorema de la Convergencia Global porque los extremos del intervalo no tienen signos diferentes, por lo que no sería posible utilizar un Método de Newton-Raphson. También hay que tener sumo cuidado a la hora de programar, debido a que es muy posible encontrar problemas de cancelación por los estrechos márgenes de tolerancia que se pudieran exigir. En el apartado 3.2.7 se explica cómo se ha incluido este desarrollo en una función de MATLAB .

En resumen, el método se basa en calcular el valor de q en el punto medio del intervalo (σ_m), dado inicialmente por σ y σ'. Si este valor de q es menor al buscado, entonces $\sigma \leftarrow \sigma_m$, y en el caso en el que sea mayor que el buscado o menor que cero, $\sigma' \leftarrow \sigma_m$. De esta manera, se reduce el intervalo hasta obtener la tolerancia deseada.

Finalmente, una vez conocidas las componentes del vector tensión, se interpola el instante de tiempo para el que se tendría el valor σ'' con la expresión (2.17) o (2.18), según haya o no tensiones normales respectivamente.

$$t_{\sigma_0''}(\sigma'') = t_{\sigma_0} + \Delta t \frac{\sigma'' - \sigma}{\sigma' - \sigma} \tag{2.17}$$

$$t_{\sigma_0''}(\tau'') = t_{\sigma_0} + \Delta t \frac{\tau'' - \tau}{\tau' - \tau} \tag{2.18}$$

2.4 Cálculo de diferencias finitas

En el sistema de ecuaciones diferenciales es necesario poder disponer de la derivada del vector de deformaciones para generar una solución paso a paso. Para ello, hay que realizar una diferenciación numérica del vector de deformaciones que se recibe como argumento de entrada al programa. No es posible obtener esta derivada de manera analítica debido a que el historial de deformaciones está compuesto por una serie de valores discretos en el tiempo.

Lamentablemente, MATLAB no cuenta con ningún tipo de comando para calcular diferencias finitas más que la función "diff", que devuelve la diferencia entre cada uno de los valores de un vector y su inmediatamente anterior. Dividiendo este resultado por el incremento de tiempo entre ambos valores consecutivos Δt, se podría obtener una derivada numérica análoga a la fórmula de diferencias progresivas de la ecuación (2.19). Esta fórmula solo tiene orden de precisión unidad, por lo que es posible que no genere buenos resultados para casos en los que en Δt no es lo suficientemente pequeño. En la figura 2.6 se compara el resultado de derivar un intervalo discreto usando el comando "diff" (diferencias progresivas) con la derivada analítica de la función, y se pone de manifiesto que el error que tiene esta fórmula no es aceptable para $\Delta t = 0.1s$.

$$\begin{aligned} Diferencia\ progresiva: \quad & f'(x_0) = \frac{f(x_0 + \Delta t) - f(x_0)}{\Delta t} \\ Diferencia\ regresiva: \quad & f'(x_0) = \frac{f(x_0) - f(x_0 - \Delta t)}{\Delta t} \end{aligned} \tag{2.19}$$

$$error = \frac{f''(\xi)\Delta t}{2} \rightarrow \mathrm{O} = 1 \tag{2.20}$$

Como en un principio se desconoce cuál va a ser la frecuencia de muestreo de los datos introducidos por el usuario, se incorporará un proceso de cálculo de derivadas con más precisión. Por esa razón, es necesario utilizar algún algoritmo de diferencias finitas de mayor orden y programarlo manualmente en lugar de utilizar una función predefinida por MATLAB. Así, se ha decidido estudiar

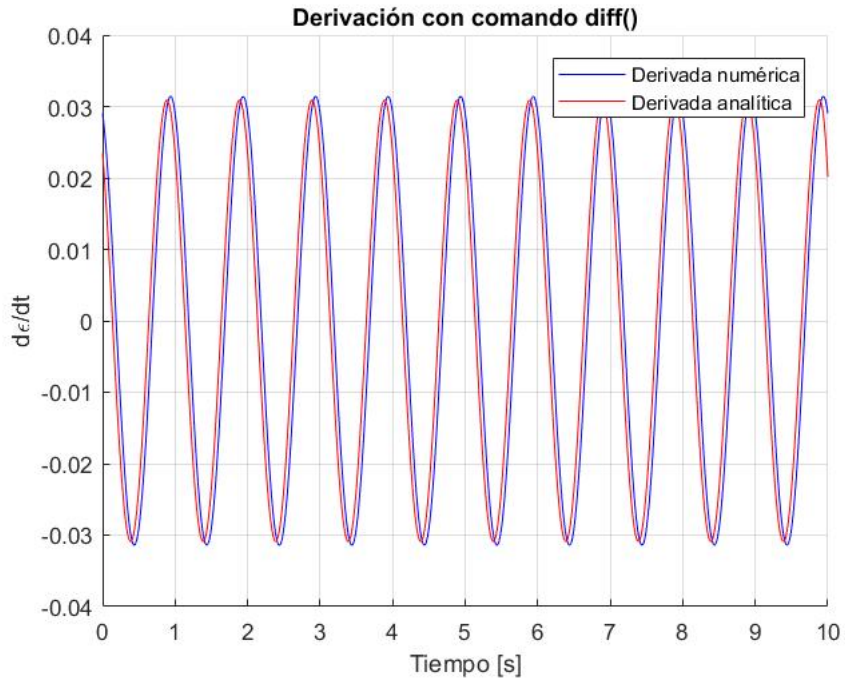

Figura 2.6 Comparativa de derivación con comando "diff".

el método de diferencias centrales (expresión (2.21)) y el de diferencias de 5 puntos (expresión (2.23)), que tienen órdenes de precisión $O = 2$ y $O = 4$ respectivamente [3].

Para analizar la exactitud de cada uno estos dos métodos, se ha vuelto a comparar el valor de las diferencias finitas arrojadas por cada uno de ellos con la derivada de la misma señal senoidal anterior calculada de forma analítica. En ambos casos se ha usado el mismo paso de integración $\Delta t = 0.1\ s$ que en la figura 2.6.

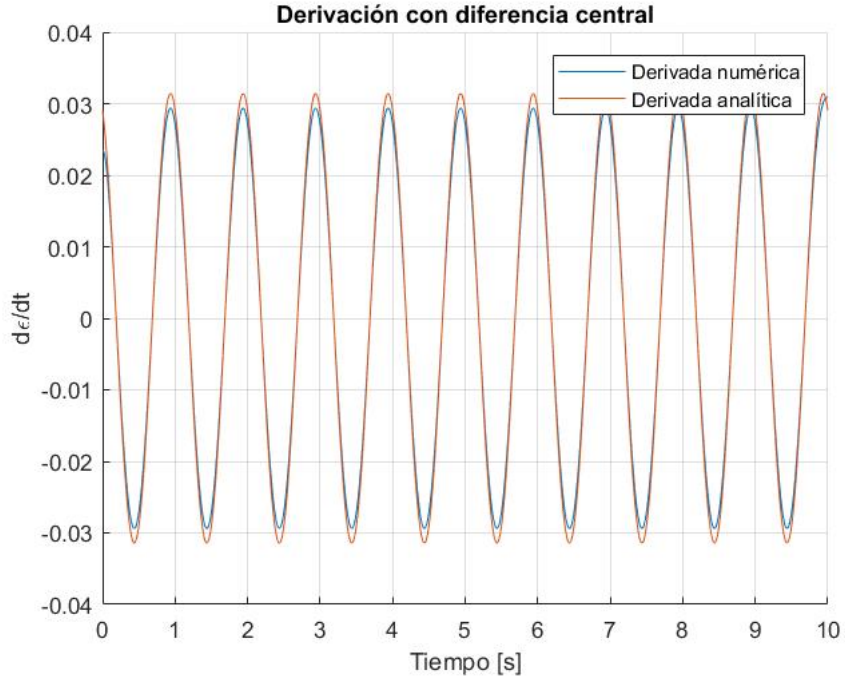

Figura 2.7 Comparativa de derivación con fórmula de diferenciación central.

$$f'(x_0) = \frac{f(x_0+\Delta t) - f(x_0-\Delta t)}{2\Delta t} \tag{2.21}$$

$$error = \frac{f'''(\xi)\Delta t^2}{3} \rightarrow \mathrm{O} = 2 \tag{2.22}$$

En la figura 2.7 se observa que el método de diferencias centrales ha mejorado con respecto a la gráfica anterior, pero aún no es lo suficientemente preciso. Es por eso que finalmente se va a optar por implementar en el programa la fórmula de diferenciación de 5 puntos, cuyo resultado se ve en la figura 2.8 y donde se confirma el buen desempeño de este método en la mayor parte del registro.

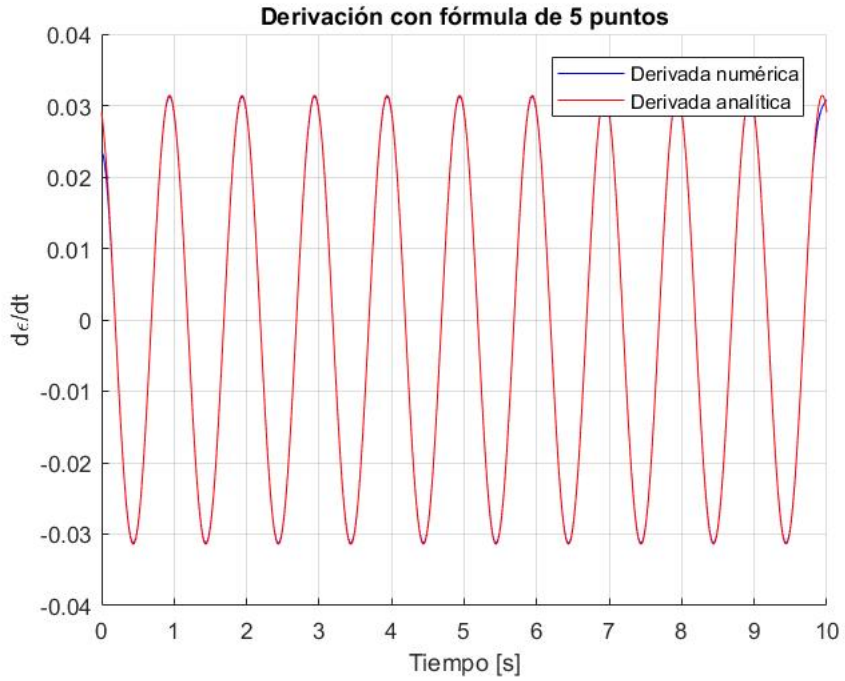

Figura 2.8 Comparativa de derivación con fórmula de 5 puntos.

$$f'(x_0) = \frac{f(x_0-2\Delta t) + 8f(x_0-\Delta t) - 8f(x_0+\Delta t) - f(x_0+2\Delta t)}{12\Delta t} \tag{2.23}$$

$$error = \frac{f^{v}(\xi)\Delta t^4}{30} \rightarrow \mathrm{O} = 4 \tag{2.24}$$

Debido a que son necesarios dos valores a la izquierda y a la derecha del punto estudiado para usar la derivada de 5 puntos, los dos primeros y los dos últimos puntos del vector de deformaciones no admiten la expresión (2.23). Para el segundo y penúltimo punto, hay que emplear la diferenciación central de la fórmula (2.21), con una precisión de orden 2 y usando un solo punto a derecha e izquierda. Por último, en el primer y último punto se ha usado la diferenciación progresiva y regresiva respectivamente (expresiones de (2.19)), que cuentan con precisión de orden 1.

Este hecho puede producir ciertas inestabilidades, como muestra la gráfica 2.8, donde se aprecia que los extremos no se ajustan adecuadamente a la curva analítica. En cualquier caso, esto no supone un gran problema debido a que el primer punto del historial se considerará elástico (la condición inicial de integración) y, por tanto, la derivada no va a influir en ese instante. En el resto del intervalo el resultado es sumamente parecido al de la derivada analítica.

Otra opción para evitar estas inestabilidades podría ser disminuir Δt, ya que, como se puede ver en la expresión (2.20), el error es directamente proporcional a Δt. Para obtener la mayor precisión posible, al margen del incremento de tiempo de las deformaciones de entrada de cada usuario, se ha optado por usar el método de 5 puntos. Además, el tiempo de cómputo total no se ve afectado excesivamente por la fórmula utilizada debido a que solo es necesario calcular las derivadas una vez a lo largo de todo el programa.

Como ya se comentó anteriormente, para poder implementar este método de derivadas pseudo-adaptativas con la fórmula de 5 puntos, es necesario programar manualmente una función de MATLAB usando las ecuaciones (2.23), (2.21) y (2.19). Esto se codificará en la función "Calcula-Derivada5Puntos", que se explica en el apartado 3.2.3.

Por último, hay que realizar una mención al comando de MATLAB "fnder", que calcula la derivada analítica de un spline. En la figura 2.9 se compara su precisión para la misma función analizada en los casos anteriores. Se observa que mejora la aproximación en los extremos del intervalo, con lo que es posible concluir que en ciertas ocasiones será incluso más fiable que la fórmula de 5 puntos. De cualquier forma, se van a incluir sendas funciones para poder calcular derivadas con los dos últimos método descritos, ya que es posible que la diferenciación numérica supere a "fnder" en aquellos casos en los que los splines no se ajusten correctamente a los puntos de precisión.

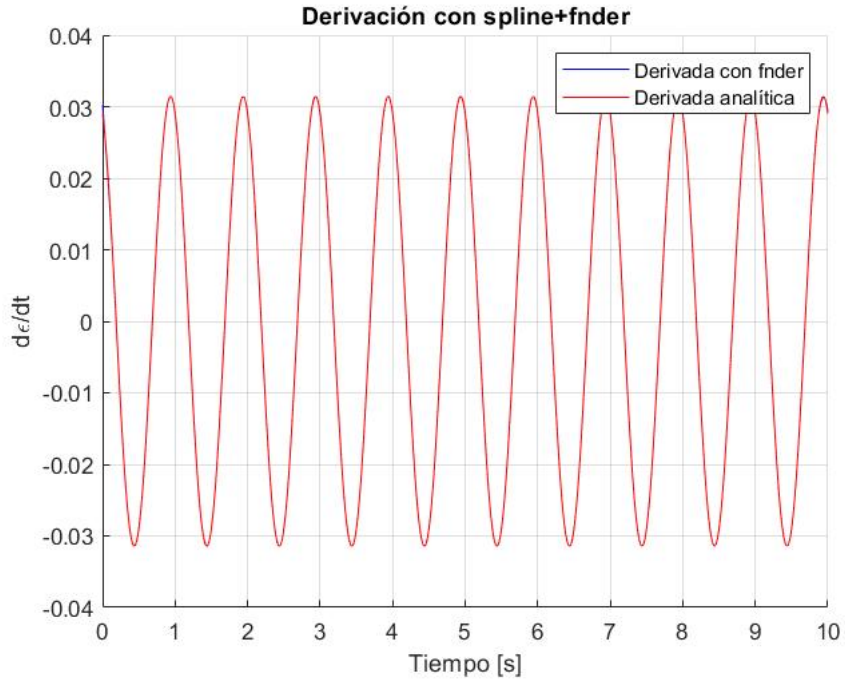

Figura 2.9 Comparativa de derivación con comando "fnder".

2.5 Integración paso a paso de sistemas de ecuaciones diferenciales

La expresión (2.25) es un sistema de dos ecuaciones diferenciales ordinarias (EDO) con dos incógnitas $\sigma(t)$ y $\tau(t)$. Este sistema cuenta con infinitas soluciones, pero en conjunto con unas condiciones iniciales se convierte en un Problema de Valores Iniciales (PVI) con una única solución.

$$\frac{d\sigma}{dt} = A(\sigma,\tau)\frac{d\varepsilon}{dt} \tag{2.25}$$

Debido a que la mayoría de los sistemas EDO no cuentan con soluciones analíticas, existen métodos numéricos para obtener una solución más o menos aproximada dentro de un determinado intervalo $[t_0, t_f]$. Estos métodos se basan en el uso de fórmulas de cuadratura para realizar aproximaciones en cada paso de integración. Los algoritmos de integración más conocidos son los métodos tipo Runge-Kutta.

MATLAB cuenta en su directorio de funciones con varios comandos que ejecutan un proceso de integración numérica para PVI (funciones ODE). Estos comandos son ampliamente conocidos y usados por profesionales de diversas áreas para la resolución de sistemas de ecuaciones diferenciales.

Las funciones ODE se basan en pares de Runge-Kutta encajados para realizar un proceso de integración adaptativo con paso variable. Por ejemplo, el comando "ODE45" utiliza un par encajado de Dormand-Prince (DOPRI5(4)) que cuenta con órdenes $p = 5$ y $\hat{p} = 4$. De manera resumida, este algoritmo compara los resultados de ambos métodos en cada iteración y disminuye así el paso de tiempos para alcanzar las tolerancias deseadas.

Desde un primer momento, se intentó utilizar uno de estos comandos ODE (ODE45, ODE113, ODE23 ...) para integrar el sistema de dos ecuaciones diferenciales. Los comandos ODE, al ser adaptativos, realizan más llamadas a la función a integrar que el número de pasos del vector de salida. Esto produce inestabilidades en las variables que guardan la memoria del material, produciendo una serie de datos basura que interfieren con la solución real. En resumen, el funcionamiento interno de ODE producía interferencias con los bucles de memoria y detección de descargas y modificaba las variables como q_k, σ_k o $\sigma_{c,k}$ cuando no tenía que hacerlo.

Entonces, se intentó solucionar este problema utilizando el comando $'Event'$ dentro de la función "odeset" para detener el bucle de integración en el punto en el que se detectara memoria. El comando "odeset" interrumpe la resolución del PVI cuando se produce un evento detectado como un cero en su argumento de entrada. La idea inicial era aplicar el algoritmo de descargas y memoria fuera de la función ODE45 una vez se detuviese la integración. Lamentablemente, el funcionamiento interno de este comando no satisface las necesidades del bucle de integración y eso produce nuevas inestabilidades sin corregir el problema anterior.

El mal funcionamiento de "odeset" se origina porque la función estudiada no cumple con el Teorema de la Convergencia Global, al igual que ocurría en el apartado 2.3. En los casos en los que se tenía $q < 0$, ODE45 no llegaba a detenerse o simplemente MATLAB devolvía errores internos. Por otro lado, se intentó establecer una bandera con valor nulo que avisara a "odeset" del instante en el que se producía una descarga o se sobrepasaba una circunferencia de memoria, pero seguía sin arrojar resultados positivos. En este caso, la función ODE45 permanecía durante un tiempo indefinido intentado buscar valores a lo largo de un intervalo cercano al punto σ, sin conseguir llegar finalmente a él. A pesar de que para algunos registros de cargas sí que llegó a funcionar este método, no se pudo asegurar una efectividad en el 100 % de los casos.

Por tanto, la única solución viable era la de programar de manera manual un bucle de integración en el que se pudiesen controlar en todo momento las ecuaciones y el bucle de memoria. Esto evita que se generen datos basura que interfieran con los reales y que produzcan salidas erróneas. A pesar de que se resuelven estas dos cuestiones, surge un nuevo conflicto: conseguir un nivel de precisión lo suficientemente bueno y equiparable al obtenido por los comandos ODE de MATLAB.

Por este motivo, se ha tomado la decisión de programar un bucle de integración que funciona con el mismo par encajado DOPRI5(4) que usa MATLAB en su función ODE45 y cuyo Tablero de Butcher se presenta en (2.26) [5]. Es un método de 7 etapas que devuelve el siguiente paso de la integración como una combinación lineal de estas etapas (según las expresiones (2.27) y (2.28)), al

igual que todas las fórmulas tipo Runge-Kutta.

$$
\begin{array}{c|c}
c & \mathbf{A} \\
\hline
 & b^T \\
\hline
 & \hat{b^T}
\end{array}
\rightarrow
\begin{array}{c|ccccccc}
0 & 0 & & & & & & \\
\frac{1}{5} & \frac{1}{5} & & & & & & \\
\frac{3}{10} & \frac{3}{40} & \frac{9}{40} & & & & & \\
\frac{4}{5} & \frac{44}{45} & -\frac{56}{15} & \frac{32}{9} & & & & \\
\frac{8}{9} & \frac{19372}{6561} & -\frac{25360}{2187} & \frac{64448}{6561} & -\frac{212}{729} & & & \\
1 & \frac{9017}{3168} & -\frac{355}{33} & \frac{46732}{5247} & \frac{49}{176} & -\frac{5103}{18656} & & \\
1 & \frac{35}{384} & 0 & \frac{500}{1113} & \frac{125}{192} & -\frac{2187}{6784} & \frac{11}{84} & \\
\hline
b_i & \frac{35}{384} & 0 & \frac{500}{1113} & \frac{125}{192} & -\frac{2187}{6784} & \frac{11}{84} & 0 \\
\hat{b_i} & \frac{5179}{57600} & 0 & \frac{7571}{16695} & \frac{393}{640} & -\frac{92097}{339200} & \frac{187}{2100} & \frac{1}{40}
\end{array}
\tag{2.26}
$$

$$y_{n+1} = y_n + h_n(b_1k_1 + ... + b_sk_s) \tag{2.27}$$

$$k_1 = f(t_n, y_n); \quad ... \quad k_7 = f(t_n + c_7h_n, y_n + h_n(a_{71}k_1 + ... + a_{7,6}k_6)) \tag{2.28}$$

Debido a que la fórmula DOPRI5(4) es un método de varias etapas y en cada una de ellas hay que llamar al sistema de ecuaciones a integrar, ha sido necesario separar el cálculo de las etapas en dos funciones diferentes. Para la primera etapa, el cálculo de k_1, se va a usar una función que cuenta con el bucle de memorias y en la que siempre se usará como argumento t_n y y_n. En cambio, para el resto de etapas k_i se van a emplear las ecuaciones sin bucle de memorias, para evitar generar los valores no deseados que sí aparecían con ODE45. La programación de este sistema se explica en detalle en la sección 3.3.

2.6 Integración de sistemas de ecuaciones diferenciales con paso adaptativo

Siguiendo con la estrategia de conseguir una precisión lo más parecida posible a la del comando ODE45, se ha querido emplear el mismo método de Dormand-Prince que se ha explicado en el apartado 2.5, pero con paso variable. Como ya se dijo, esta fórmula es un par encajado de Runge-Kutta de órdenes $\hat{p} = 4$ y $p = 5$ y con el mismo Tablero de Butcher de la expresión (2.26).

En primer lugar, se calculan los valores de las etapas y se hallan las salidas y_{n+1} y $\hat{y}_{n+1}$. Tras esto, es posible estimar el error en cada uno de los pasos comparando los valores obtenidos por medio de los pares de mayor y menor orden ($p = 5$ y $\hat{p} = 4$ respectivamente) de acuerdo con (2.29). Para aplicar el método adaptativo, esta estimación se compara posteriormente con la tolerancia deseada por el usuario y se desecha el valor calculado en el caso en el que no sea lo suficientemente preciso [4].

Por último, se prueba de nuevo con un incremento de tiempo menor si fuese necesario y se vuelven a realizar todos los cálculos hasta llegar a este mismo punto. Normalmente, los algoritmos también cuentan con un contador límite de iteraciones para no mantener un bucle infinito en los casos de tolerancias muy exigentes. El nuevo incremento de tiempo se puede estimar con la fórmula (2.30) para hacer el algoritmo más eficiente y que tarde menos en converger.

$$EST_n \approx ||\hat{y}_{n+1} - y_{n+1}|| \tag{2.29}$$

$$h'_n \approx h_n \sqrt[\hat{p}+1]{\frac{TOL}{EST_n}} = h_n \sqrt[5]{\frac{TOL}{EST_n}} \tag{2.30}$$

El bucle de integración dentro de los cuales se programará este algoritmo con paso adaptativo se encuentra en el apartado 3.5.

3 Programa de cálculo numérico en MATLAB

Este apartado desgrana las partes en las que se divide el programa y explica el funcionamiento de cada una de ellas. También se detallan las decisiones que se han tomado a la hora de elegir cómo diseñar la programación de cada uno de los códigos y cómo se han implementado los desarrollos matemáticos del apartado 2.

En primer lugar, se explicará la solución con POO e integración del PVI con paso fijo, para avanzar posteriormente hacia la mejora que supone la implementación de un paso variable. A continuación, se desarrollará una alternativa que realiza los mismos cálculos pero utilizando variables globales, dirigido a aquellas personas que no están familiarizadas con el funcionamiento de la Programación Orientada a Objetos. Por último, se enseñará cómo se ha creado un fichero que trabaja directamente con tensiones en lugar de deformaciones.

Igualmente, es necesario mencionar dos criterios de diseño que han sido la base para cada uno de los fragmentos de código que se ilustran a lo largo de esta sección: optimizar el programa con la finalidad de disminuir el tiempo de procesado y evitar al máximo los posibles errores de cancelación numérica.

Para aquellas funciones o archivos de MATLAB de mayor complejidad, se van a adjuntar una serie de diagramas de flujo simplificados y con información sintetizada. En estos diagramas de flujo no se van a representar todas las acciones que se realicen en el código, sino que constituirán un esquema de las operaciones más importantes que se llevan a cabo en una parte concreta del programa.

3.1 Estructura del programa de cálculo

En líneas generales, el programa se puede resumir mediante el diagrama de flujo de la figura 3.1. En primer lugar, es necesario leer los datos de entrada y acondicionar las variables para la integración numérica; seguidamente, se produce la resolución del problema de valores iniciales; y por último, hay que generar una salida para mostrar el resultado al usuario. La parte más crítica del programa es la integración del sistema de ecuaciones diferenciales, donde hay que diseñar un algoritmo específico para poder detectar las descargas y aplicar la memoria de carga en cada instante de tiempo.

Para poder obtener un programa que alcance el resultado deseado, ha sido necesario usar archivos de MATLAB de diferente naturaleza, combinándolos para que desempeñen las tareas descritas anteriormente. Estos archivos se pueden separar en tres bloques principales cuyo contenido se resume a continuación: el código principal (archivo tipo script.m), las funciones auxiliares (function.m) y la clase de MATLAB (class.m) "punto".

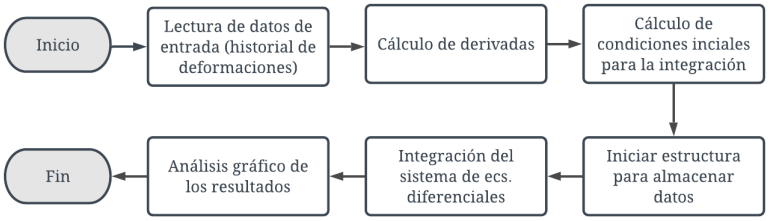

Figura 3.1 Diagrama de flujo resumen del funcionamiento del programa.

- **Archivo principal o "main"**: Es el fichero tipo "script" que debe ejecutarse y en el que el usuario debe introducir el nombre del archivo en el que están localizados los datos de entrada. Este "script" se encarga de ordenar la lectura y la derivación finita de las deformaciones, así como de iniciar el objeto donde se guardará todo el registro de datos. Adicionalmente, en el "main" se encuentra el bucle de resolución paso a paso de las ecuaciones diferenciales.
- **Clase "punto"**: Es la clase de la que va a derivar el objeto en el que se guarden todos los datos. También contiene los métodos que permiten calcular cada uno de los pasos en el bucle de integración. Cuenta con un método principal, en el que se encuentra el algoritmo de descargas y memoria, y uno auxiliar, que solo aplica las ecuaciones de salida en un punto. Ambos se combinan para generar las diferentes etapas del método paso a paso dentro del archivo principal.
- **Funciones auxiliares**: Realizan tareas específicas dentro del programa y facilitan y ordenan la programación. También pueden ser llamadas por medio del *CommandWindow* para realizar cálculos puntuales si el usuario lo necesitara. Estas funciones no son métodos dentro de la clase "punto", sino que son archivos independientes que realizan cálculos específicos.
- **Clase "datmat" para material**: Es una clase de MATLAB en la que se guardan las constantes físicas de las propiedades del material. A las propiedades de esta clase se les ha dado el atributo "Constant", no cuentan con métodos y no pueden ser modificadas fuera de este archivo.

En cada uno de los apartados siguientes se va a explicar en detalle cómo funcionan todos los archivos de MATLAB que componen cada una de estas partes, por qué se han diseñado de esta manera y cómo se relacionan entre ellos para lograr el resultado final.

Como se puede ver, se ha hecho uso de las clases de MATLAB mencionadas en el apartado 1.5.1 para implementar un sistema de Programación Orientada a Objetos. En un principio, se consideró la posibilidad de utilizar variables globales, pero finalmente se desechó la idea porque pueden llegar a ser fuente de errores difíciles de detectar. Muchos profesores y programadores experimentados desaconsejan el uso de variables globales a la hora de programar debido a que pueden interferir con variables locales, e incluso la propia web de ayuda online de MATLAB intenta disuadir de su uso a no ser que sea estrictamente necesario [8].

El uso de POO también permite un código más limpio y claro, que es más fácil de modificar y de entender por una persona externa al diseño del programa. Por último, con esta implementación también se mejora la integración de toda la información (historial de tensiones, deformaciones, valores de q ...) en un único objeto para el punto estudiado de la probeta.

3.2 Funciones auxiliares

En un programa que llega a alcanzar un cierto nivel de complejidad y en el que hay que realizar una gran variedad de operaciones, separar algunas tareas en funciones auxiliares simplifica enormemente

los ficheros principales. En este apartado se van a exponer qué tareas han sido delegadas a este tipo de funciones y cómo las llevan a cabo.

Asimismo, se cuenta con la ventaja de poder utilizar estas funciones desde el *CommandWindow* de MATLAB para comprobar valores puntuales o incluir alguna de ellas en otro archivo .m si se necesitase. De esa manera, estas funciones también pueden llegar a ser usadas en códigos futuros con un propósito totalmente diferente y, por tanto, ahorrar tiempo al programador con métodos ya definidos. Es por eso que, a la hora de diseñar estas funciones, siempre se ha tenido en cuenta el criterio de dotarlas de una cierta independencia del código principal, incrementando así la versatilidad de todo el programa.

Estas funciones auxiliares se emplean, sin necesidad de ser modificadas, tanto en la solución con Programación Orientada a Objetos como en el posterior ejemplo con variables globales. Este hecho refuerza la idea del párrafo anterior, pudiendo distinguir una serie de cálculos y algoritmos auxiliares que se podrían considerar pseudo-independientes.

3.2.1 Datos del material

Varias de las funciones usadas a lo largo del código necesitan los datos de las propiedades físicas del material del que está hecha la probeta. Estos datos serán constantes en todo momento y deben poder ser modificados por el usuario en el caso en el que se quiera realizar la integración numérica de una probeta fabricada con otro tipo de material.

Para guardar estos datos, se ha decidido usar un archivo clase de MATLAB con propiedades constantes. El atributo "Constant" que se puede ver en la línea 5 del Código 3.1 impide que estos datos sean modificados por el usuario de manera externa y es posible acceder a ellos desde cualquier función como si se tratase de una estructura global.

Si se quisieran modificar las propiedades del material estudiado, solo habría que reescribirlas en el archivo de tipo clase "datmat.m" y, desde ese momento, todas las funciones emplearían para los cálculos esos nuevos valores.

Código 3.1 Clase de MATLAB datmat.

```
1  classdef datmat
2   % Propiedades del Inconel utilizado declaradas como constantes
3   % Se puede acceder a ellas desde cualquier función y no se pueden
4   % modificar en ningún otro lugar que no sea este mismo script
5      properties (Constant)
6          Young=208.5e3; % MPa
7          K=1910; % MPa
8          n=0.08;
9          G=77.8e3; % MPa
10     end
11     methods
12     end
13 end
```

El motivo por el que se decidió usar una clase con propiedades constantes en lugar de una estructura fue evitar tener que pasar como argumento en cada una de las funciones los datos del material. En realidad, se podría haber optado por una variable global, al igual que se hace con otro tipo de datos en el programa en variables globales, pero ya se han comentado los posible problemas que esto puede conllevar. Además, otra razón por la que se ha tomado esta decisión es para hacer énfasis en que las propiedades físicas deben ser invariables a lo largo del programa y no es posible que sean modificadas por ninguna función. La clase con propiedades constantes actúa

como mecanismo de seguridad ante posibles fallos del usuario en caso de que este modifique el código incorrectamente.

Se aconseja introducir los valores de E, G y K en MPa en lugar de Pa para que el orden de magnitud de los números que maneja MATLAB sea menor y, por tanto, se eviten problemas de precisión. En principio, no habría problema en utilizar Pascales si se desea, pero todas las pruebas realizadas han sido con datos en megapascales. Hay que tener en cuenta que la tensiones de salida tendrán unidades de MPa o Pa, según las unidades en las que se introduzcan estos datos.

3.2.2 Función Lectura

Se encarga de leer los datos de entrada, es decir, los vectores de deformaciones normales y tangenciales y el vector de tiempos en el punto estudiado. Estos valores se encuentran en un archivo tipo .txt que debe guardarse en la misma carpeta del programa.

La función recibe como argumento el nombre del archivo en el que se encuentran dichos datos como un vector de caracteres y devuelve dos salidas diferentes: una matriz de dos columnas con los valores de deformaciones y un vector columna con los instantes de tiempo en los que se producen dichas deformaciones. Ambas salidas tienen el mismo número de filas n_{def}, que se corresponde con el número total de registros, es decir, los puntos de precisión que devuelve el control de deformaciones de la máquina de ensayos de fatiga.

La función "Lectura" está diseñada para leer archivos de texto con una configuración de tres columnas y un número indefinido de filas. El orden de dichas columnas debe ser el siguiente (de izquierda a derecha): vector de tiempos, deformaciones normales y deformaciones tangenciales.

Código 3.2 Función Lectura.

```
1  function [def_entrada, tiempo]=Lectura(nombre) % [ndefx2, ndefx1]
2
3  % Lee a partir de un fichero de texto (el nombre se da como argumento)
4  % los valores de las deformaciones y el vector de instantes de
5  % tiempo y los convierte en vectores columnas como salidas
6
7  %%% ABRIR FCHERO Y LEER DATOS %%%
8
9  fileID = fopen(nombre,'r');
10 formatSpec = '%f';
11 tam=[3 Inf];
12 datos_entrada=fscanf(fileID, formatSpec, tam);
13 fclose(fileID);
14 datos_entrada=datos_entrada';
15
16 %%% ORDENAR DATOS DE SALIDA %%%
17
18 tiempo=datos_entrada(:, 1);
19 def_entrada=datos_entrada(:, 2:3);
20
21 end
```

Se recomienda revisar la disposición de los archivos de entrada antes de ejecutar el programa para asegurarse de que los datos se encuentran en el orden descrito anteriormente. Si se tuviese un archivo de texto con una ordenación de datos diferente a la descrita, habría que modificar esta función conforme a lo requerido en cada situación.

3.2.3 Función CalculaDerivada5Puntos y CalculaDerivadaSpline

Estas funciones calculan los vectores de derivadas de las deformaciones de entrada, que devuelven en forma de estructuras spline. Para ello, deben recibir dichas deformaciones así como el vector de tiempos para cada uno de los datos. La diferencia entre ambas funciones es que "CalculaDerivada5Puntos" emplea el método numérico de la fórmula de 5 puntos mientras que "CalculaDerivadaSpline" utiliza el comando "fnder".

Adaptar la fórmula de los 5 puntos no es excesivamente complicado y basta con utilizar un comando "switch" dentro del bucle "for" que recorra cada uno de los puntos del vector de deformaciones. Es importante destacar que dentro del bucle "for" siempre se ejecutará alguna de las instrucciones del "switch", pero nunca más de una en cada pasada. El resultado final es el algoritmo del Código 3.3, que devuelve dos vectores columna de dimensiones $n_{def}x1$ que contienen los valores de $d\varepsilon/dt$ y $d\gamma/dt$. Esta función sirve para cualquier longitud de registros, ya que calcula sus dimensiones de manera interna (siempre que se pasen argumentos como vectores columna).

Código 3.3 Función CalculaDerivada5Puntos.

```
function [ppdepsilon, ppdgamma]=CalculaDerivada5Puntos(def, tdef)

deltaT=tdef(2)-tdef(1); % Siempre funciona con paso fijo

% Esta función recibe un vector con las deformaciones de entrada y
% calcula dos splines de salida con las derivadas de las mismas

[fil, ~]=size(def);
nval=fil; % Número de valores del bucle

epsilon=def(:, 1);gamma=def(:, 2);

depsilon=zeros(nval, 1); % Inicializar vectores para mayor velocidad
dgamma=zeros(nval, 1); % nvalx1

% Bucle de derivación: según el punto utiliza una u otra forma para
% obtener la mayor precisión posible)

for i=1:nval

    switch i

        case 1 %dif progresiva
            depsilon(i)=(epsilon(i+1)-epsilon(i))/deltaT;
            dgamma(i)=(gamma(i+1)-gamma(i))/deltaT;

        case 2 %dif central
            depsilon(i)=(epsilon(i+1)-epsilon(i-1))/(2*deltaT);
            dgamma(i)=(gamma(i+1)-gamma(i-1))/(2*deltaT);

        case nval %dif regresiva
            depsilon(i)=(epsilon(i)-epsilon(i-1))/deltaT;
            dgamma(i)=(gamma(i)-gamma(i-1))/deltaT;
```

```
34
35          case (nval-1) %dif central
36              depsilon(i)=(epsilon(i+1)-epsilon(i-1))/(2*deltaT);
37              dgamma(i)=(gamma(i+1)-gamma(i-1))/(2*deltaT);
38
39          otherwise %dif 5 puntos
40              depsilon(i)=(epsilon(i-2)-8*epsilon(i-1)+8*epsilon(i+1)-
                    epsilon(i+2))/(12*deltaT);
41              dgamma(i)=(gamma(i-2)-8*gamma(i-1)+8*gamma(i+1)-gamma(i+2))
                    /(12*deltaT);
42
43      end
44  end
45
46  %Crear splines para poder ser evaluados en cualquier t
47  ppdepsilon=spline(tdef, depsilon);
48  ppdgamma=spline(tdef, dgamma);
49
50  end
```

La limitación de la función anterior es que el incremento de tiempo debe ser constante en todo el intervalo, hecho difícil de garantizar en la mayoría de registros experimentales. Para estos casos debe utilizarse la función "CalculaDerivadaSpline", que no depende de un paso constante porque interpola el registro de deformaciones antes de derivar. Si ambas funciones fuesen adecuadas para tratar los datos introducidos por el usuario, se recomienda comparar resultados, ya que es posible que el desempeño de una sobre la otra difiera según qué situaciones.

Código 3.4 Función CalculaDerivadaSpline.

```
1  function [ppdepsilon, ppdgamma]=CalculaDerivadaSpline(def_entrada, tdef)
2  % Esta función genera splines de derivadas utilizando los comandos
3  % spline y fnder de MATLAB
4
5      epsilon=def_entrada(:, 1);
6      gamma=def_entrada(:, 2);
7
8  % Crear splines con las deformaciones
9      ppepsilon=spline(tdef, epsilon);
10     ppgamma=spline(tdef, gamma);
11
12 % Derivar los splines
13     ppdepsilon=fnder(ppepsilon, 1);
14     ppdgamma=fnder(ppgamma, 1);
15
16 end
```

3.2.4 Funciones CalculaqCarga, CalculaqDescos y CalculaqDesqk

Uno de los aspectos clave del programa es el cálculo de la distancia efectiva entre puntos de descargas sucesivas, es decir q. Hay que distinguir entre estado de cargas o descargas, ya que en cada uno de

estos dos casos es necesario utilizar una ecuación diferente.

En primer lugar y para el caso de cargas, la ecuación a emplear es la 3.1 [11]. Como se puede ver en el Código 3.5, su implementación es sencilla y la función devuelve el diámetro de la circunferencia dados como argumentos las componentes del vector de tensiones.

$$Q = \sqrt{\frac{2}{3}\sigma^2 + 2\tau^2} \tag{3.1}$$

Código 3.5 Función CalculaqCarga.

```
1 function q=CalculaqCarga(sigma, tau)
2 % Cálculo de q mediante la ecuación para cargas (q=2*Q)
3
4 q=2*sqrt((2/3)*sigma^2+2*tau^2);
5
6 end
```

Para el caso más general de descargas, es necesario emplear la ecuación 3.2 [12]. Como se puede ver, es posible hallar el valor de q calculando la norma del vector $\sigma - \sigma_k$ y el coseno del ángulo que forman los vectores de cargas, o bien utilizando el desarrollo del miembro más a la derecha de la expresión. Se han desarrollado funciones independientes que calculan el valor de q de estas dos maneras para comparar la estabilidad de ambos métodos y evitar posibles errores de cancelación numérica. Asimismo, si algún usuario quisiera comparar valores de q podría optar por usar cualquiera de las dos ecuaciones, según la situación en la que se encuentre.

$$q = \frac{|\sigma - \sigma_k|}{cos\theta} = \frac{\frac{2}{3}(\sigma - \sigma_k)^2 + 2(\tau - \tau_k)^2}{(\sigma_{c,k} - \sigma_k)(\sigma - \sigma_k) + (\tau_{c,k} - \tau_k)(\tau - \tau_k)} \frac{q_k}{2} \tag{3.2}$$

$$cos\theta = \frac{(\sigma - \sigma_\mathbf{k}) \cdot (\sigma_{c,k} - \sigma_\mathbf{k})}{|\sigma - \sigma_\mathbf{k}||\sigma_{c,k} - \sigma_\mathbf{k}|} \tag{3.3}$$

El Código 3.6 plasma en MATLAB el primero de estos métodos (se usa el miembro central de la ecuación 3.2 de la manera que se ve en 3.3). Para ello necesita los últimos valores guardados en memoria de σ_k y $\sigma_{c,k}$ además de las componentes del vector de tensiones. En primera instancia, los cálculos de las normas y del producto escalar (líneas 13-17) no deberían generar ningún tipo de error de cancelación.

Código 3.6 Función CalculaqDescos.

```
1  function q=CalculaqDescos(sigma, tau, tensk, ck)
2
3  % Cálculo de q sin desarrollar (usando el coseno)
4
5  sigmak=tensk(1);tauk=tensk(2);
6
7  % Vectores sobre los que se realiza el cálculo
8  vect1=ck-tensk;
9  vect2=[sigma, tau]-tensk;
10
11 %%% CALCULAR q CON EL COSENO %%%
12
```

```
13 % Cálculo del módulo de los vectores
14 normvect1=sqrt((2/3)*vect1(1)*vect1(1)+2*vect1(2)*vect1(2));
15 normvect2=sqrt((2/3)*vect2(1)*vect2(1)+2*vect2(2)*vect2(2));
16 % Producto escalar de los vectores
17 pescalar=(2/3)*vect1(1)*vect2(1)+2*vect1(2)*vect2(2);
18
19 costheta=(pescalar/(normvect1*normvect2));
20 q=(sqrt((2/3)*(sigma-sigmak)^2+2*(tau-tauk)^2))/costheta; % Salida
21
22 end
```

El Código 3.7 muestra la segunda alternativa para hallar q. En este caso es posible que existan errores de cancelación en las líneas 11 y 12 si se llegasen a realizar restas de números muy próximos entre sí. En esta función, y a diferencia del Código 3.6, también es necesario añadir como argumento el valor de referencia de q_k, lo que incluso puede suponer un problema de propagación de errores sumado a la cancelación.

Código 3.7 Función CalculaqDesqk.

```
1  function q=CalculaqDesqk(sigma, tau, tensk, ck, qk)
2
3  % Cálculo de q mediante la ecuación desarrollada
4
5  sigmack=ck(1);tauck=ck(2);
6
7  sigmak=tensk(1);tauk=tensk(2);
8
9  %%% CALCULAR q CON LA FÓRMULA DESARROLLADA %%%
10
11 num=(2/3)*(sigma-sigmak)^2+2*(tau-tauk)^2;
12 den=(2/3)*(sigmack-sigmak)*(sigma-sigmak)+2*(tauck-tauk)*(tau-tauk);
13
14 q=(qk/2)*num/den; % Salida
15
16 end
```

Se ha realizado una comprobación para varios registros de tensiones en los que ambas funciones generan los mismos valores de q. Para el caso del Código 3.7 existe algún pequeño error que se da en ocasiones muy puntuales debido a la posible propagación de errores numéricos mediante los valores guardados de q_k y/o a algún tipo de cancelación. Igualmente, se ha obtenido alguna inestabilidad puntual para el caso del Código 3.6 en otra serie de registros. Es por eso por lo que se aconseja al posible usuario que realice comprobaciones con ambos procedimientos si no tiene gran seguridad sobre los resultados generados. Por defecto, los archivos principales y las funciones usadas en el programa final tendrán implementado el cálculo con el coseno y los productos escalares del Código 3.6. Si se quisiera cambiarlo, solo habría que sustituir "CalculaqDescos" por "CalculaqDesqk" en el código. Se ha tomado esta decisión debido a que es menos probable obtener cancelación numérica por este método. De todas formas, ambos procedimientos arrojan datos sumamente parecidos y con diferencias despreciables en la gran mayoría de los casos.

Como se comentó al principio del apartado 3.2, se han querido mantener cálculos importantes en funciones independientes para hacer de todo el conjunto un código más polivalente. Es necesario hacer especial hincapié en el hecho de tener estas tres funciones de forma independiente al resto del

código, ya que el cómputo del parámetro q es la base de la teoría en la que se basa el programa y es un cálculo que podría ser muy utilizado por un usuario futuro. Este hecho se ha puesto de manifiesto durante la propia fase de pruebas y de diseño, donde se han utilizado estas funciones en numerosas ocasiones para verificar el progreso realizado.

3.2.5 Funciones GeneraAQ y GeneraADes

Generar la matriz $A(\sigma,\tau)$ de la expresión (3.4) es una parte fundamental de este programa, ya que conforma el cuerpo principal del sistema de ecuaciones diferenciales. Al igual que ocurre con el cálculo de q, esta matriz tiene una morfología diferente para el caso de cargas y el de descargas [12]. Para programar estas funciones se ha hecho uso de las herramientas de construcción de matrices de MATLAB.

$$\frac{d\sigma}{dt} = A(\sigma,\tau)\frac{d\varepsilon}{dt} \tag{3.4}$$

Se ha decidido realizar estos cálculos de forma separada en dos nuevas funciones: "GeneraAQ" (Código 3.8), que se encarga de calcular $\mathbf{A}(\sigma,\tau)$ para el caso de cargas, y "GeneraADes" (Código 3.9), que se usa para el caso general de descargas. Estas llaman a su vez a otras funciones auxiliares, "HModQ" y "HModDes" (que se describen en el apartado siguiente) respectivamente, para que realicen el cálculo del módulo de endurecimiento en cada caso. Estas funciones son una parte realmente importante del bucle de integración debido a que es necesario calcular un nuevo valor de la matriz $\mathbf{A}(\sigma,\tau)$ para cada instante de tiempo en la iteración.

Para el caso de cargas, la matriz viene dada por la expresión (3.5) y hay que resaltar que la función "GeneraAQ" debe recibir como argumento, además de las componentes del vector de tensiones, el valor de $Q = q/2$ en lugar de q.

$$A(\sigma,\tau) = \frac{1}{\frac{1}{E}\frac{1}{G} + \Phi(Q)\left[\frac{4n_2^2}{E} + \frac{n_1^2}{G}\right]} \begin{bmatrix} \frac{1}{G} + 4n_2^2\Phi(Q) & -2n_1n_2\Phi(Q) \\ -2n_1n_2\Phi(Q) & \frac{1}{E} + n_1^2\Phi(Q) \end{bmatrix} \tag{3.5}$$

Código 3.8 Función GeneraAQ.

```
1  function A=GeneraAQ(sigma, tau, Q) % Q=q/2
2  % Genera la matriz A para el caso de CARGAS con Q=q/2
3
4  %%% DATOS DEL MATERIAL %%%
5
6  G=datmat.G;
7  Young=datmat.Young;
8
9  %%% CÁLCULO DE n1 y n2 %%%
10
11 n1=2/3*sigma/Q;
12 n2=tau/Q;
13
14 %%% CONSTRUIR A CON Q DADO EXTERNO %%%
15 % Esta función llama a HmodQ(Q) con Q=q/2
16
17 aux=1/(1/(Young*G)+HModQ(Q)*((4*n2^2/Young)+(n1^2/G)));
18
19 B11=1/G+4*n2^2*HModQ(Q);
```

```
B12=-2*n1*n2*HModQ(Q);
B21=B12;
B22=1/Young+n1^2*HModQ(Q);
B=[B11, B12; B21, B22];

A=aux*B; % Salida

end
```

Para el caso de descargas, la matriz $\mathbf{A}(\sigma, \tau)$ se calcula a partir de (3.6) [12]. En este caso, el valor que se debe pasar como argumento es q en lugar de $q/2$. Adicionalmente, esta función debe recibir las variables σ_k, $\sigma_{c,k}$ y q_k, guardadas en la memoria del bucle de integración. Como se puede ver en el Código 3.9, hay que realizar las mismas operaciones que en el Código 3.6 para hallar el coseno del ángulo de los vectores de tensiones.

$$A(\sigma,\tau) = \frac{1}{\frac{1}{E}\frac{1}{G} + \frac{\phi(q)}{cos\theta}\left[\frac{4n_2^2}{E} + \frac{n_1^2}{G}\right]} \begin{bmatrix} \frac{1}{G} + 4n_2^2\frac{\phi(q)}{cos\theta} & -2n_1n_2\frac{\phi(q)}{cos\theta} \\ -2n_1n_2\frac{\phi(q)}{cos\theta} & \frac{1}{E} + n_1^2\frac{\phi(q)}{cos\theta} \end{bmatrix} \tag{3.6}$$

Código 3.9 Función GeneraADes.

```
function A=GeneraADes(sigma, tau, q, tensk, ck, qk) % q=diámetro
% Genera la matriz A para el caso de DESCARGAS q=diámetro

sigmak=tensk(1);tauk=tensk(2);
sigmack=ck(1);tauck=ck(2);

%%% DATOS DEL MATERIAL %%%

G=datmat.G;
Young=datmat.Young;

%%% CÁLCULO DE n y cos %%%

n1=(2/3)*(sigma-sigmak)*2/q-(2/3)*(sigmack-sigmak)*2/qk;
n2=(tau-tauk)*2/q-(tauck-tauk)*2/qk;

% Mismo cálculo del coseno que en CalculaqDescos

vect1=ck-tensk;
vect2=[sigma, tau]-tensk;

normvect1=sqrt((2/3)*vect1(1)*vect1(1)+2*vect1(2)*vect1(2));
normvect2=sqrt((2/3)*vect2(1)*vect2(1)+2*vect2(2)*vect2(2));
pescalar=(2/3)*vect1(1)*vect2(1)+2*vect1(2)*vect2(2);
costheta=(pescalar/(normvect1*normvect2));

%%% CONSTRUIR A CON q %%% (en este caso se usa HModDes)

aux=1/(1/(Young*G)+(HModDes(q)/(costheta^2))*((4*n2^2/Young)+(n1^2/G)));
```

```
30
31 B11=1/G+4*n2^2*HModDes(q)/(costheta^2);
32 B12=-2*n1*n2*HModDes(q)/(costheta^2);
33 B21=B12;
34 B22=1/Young+n1^2*HModDes(q)/(costheta^2);
35 B=[B11, B12; B21, B22];
36
37 A=aux*B; % Salida
38
39 end
```

3.2.6 Funciones HModQ y HModDes

Para poder generar la matriz $\mathbf{A}(\sigma,\tau)$ con las funciones del apartado anterior, es necesario calcular el módulo de endurecimiento del material. De acuerdo con el modelo en el que se basa este TFG, el módulo de endurecimiento se obtiene a partir de las ecuaciones (3.7) y (3.8) para el caso de cargas y descargas respectivamente [12].

$$\Phi(Q) = \left(\sqrt{\frac{3}{2}}\right)^{(\frac{1}{n}+1)} \frac{1}{n}\left(\frac{Q}{K}\right)^{\frac{1}{n}} \frac{1}{Q} \tag{3.7}$$

$$\phi(q) = 2^{(1-\frac{1}{n})}\left(\sqrt{\frac{3}{2}}\right)^{(\frac{1}{n}+1)} \frac{1}{n}\left(\frac{q}{K}\right)^{\frac{1}{n}} \frac{1}{q} \tag{3.8}$$

Para realizar estos cálculos, se han diseñado funciones auxiliares que devuelven el módulo de endurecimiento $\phi(q)$ dado un valor de q. "HModQ" (Código 3.10) es llamada por la función "GeneraAQ" en casos de cargas y "HModDes" (Código 3.11) es llamada por "GeneraADes" para los casos de descarga.

Como se puede apreciar, son funciones bastante sencillas que solo sirven para despejar el código de otras más complejas. Simplemente acceden a las propiedades del material y calculan el módulo con las ecuaciones correspondientes. El único detalle digno de atención es que para el caso de cargas hay que pasar como argumento el valor de $Q = q/2$, y para el caso de descargas, este sería q.

Otro aspecto importante que hay que tener en cuenta es que no es posible calcular el valor de $\phi(q)$ o $\Phi(Q)$ para los casos en los que $\sigma = 0$, $\tau = 0$ y $q_k = 0$, ya que MATLAB devolvería el valor "INF". Este trío de valores solo se podría dar si las condiciones iniciales fueran $\sigma = [0,0]$ y, por tanto, hay que tenerlo en cuenta para que el programa realice la comprobación una vez que el usuario haya introducido las entradas.

Código 3.10 Función HModQ.

```
1  function Sal=HModQ(Q)
2
3  % Esta función calcula el Hardening Modulus para un material de
4  % Ramberg-Osgood dada Q=q/2 para el caso de cargas
5
6  K=datmat.K;
7  n=datmat.n;
8
9  Sal=(sqrt(3/2)^(1/n+1))*(1/n)*((Q/K)^(1/n))*(1/Q);
10 end
```

Código 3.11 Función HModDes.

```
1  function Sal=HModDes(q)
2
3  % Esta función calcula el Hardening Modulus para un material de
4  % Ramberg-Osgood con q=diámetro y en descarga
5
6  K=datmat.K;
7  n=datmat.n;
8
9  Sal=2^(1-1/n)*sqrt(3/2)^(1/n+1)*(1/n)*(q/K)^(1/n)*(1/q);
10 end
```

En caso de que se quisiera estudiar otro tipo de probeta que no cumpliese el modelo de material con el que se ha diseñado este código, solo habría que modificar estas funciones en lugar de tener que hacer cambios en múltiples lugares del programa. Por otro lado, también se podrían declarar nuevas funciones con otro tipo de modelos elasto-plásticos y sustituirlas en las funciones "GeneraAQ" (Código 3.8) y "GeneraADes" (Código 3.9) si fuese requerido.

3.2.7 Funciones BuscatensRet y CalculatRet

"BuscatensRet" es la función que se encarga de ejecutar el método de la bisectriz explicado en el apartado 2.3. Para ello, debe recibir como argumentos las componentes de los puntos σ_0, σ'_0, el valor de q que hay que "buscar" y los datos necesarios para calcular ese valor a lo largo de las sucesivas iteraciones. Las salidas que devuelve son las dos componentes del vector de tensiones calculadas mediante la interpolación y que posteriormente se añadirán al historial de cargas.

Para evitar errores de cancelación en el algoritmo, se ha optado por utilizar una tolerancia absoluta entre los valores de σ en lugar de una tolerancia relativa con el valor de q, que podría ser lo más común. Así se asegura obtener el punto de tensiones que se busca con las mismas cifras decimales que se indiquen en el valor de tolerancia. Este valor se ha estipulado por defecto como $TOL = 10^{-8}$ porque es mucho menor que el orden del mínimo paso de integración, que será aproximadamente 0.0001 segundos.

En los algoritmos tipo bisectriz es muy común producir cancelaciones numéricas, y esta es la razón por la que en el Código 3.12 no se realizan restas entre valores próximos más que en las líneas donde es absolutamente necesario (línea 41).

Ha sido obligatorio diferenciar los casos en los que no hay tensiones normales de los que sí las presentan, con objeto de reducir el problema a una variable, τ o σ respectivamente. Una vez que se elige qué variable se va utilizar, se impone la condición de contorno para que el punto buscado se encuentre en la línea que une σ'_0 y σ_0 y se itera hasta obtener la tolerancia deseada. En el algoritmo, simplemente se calcula el valor de q_0 en el punto medio del intervalo y se asigna como extremo inferior (en el caso en el que $q_0 < q_{buscado}$) o como extremo superior (en el caso en el que $q_0 > q_{buscado}$ o $q_0 < 0$). Este desarrollo se presenta gráficamente en el diagrama de flujo de la figura 3.2.

Código 3.12 Función BuscatensRet.

```
1  function [sigmaret, tauret]=BuscatensRet(tensmenos, tensmas, tensk, ck,
       qbuscado)
2  % Esta función se encarga de interpolar cuando se detecta memoria para
3  % retraer las tensiones al punto justo donde se detectó el q>qk
4
```

```
5  Tol=1e-8; % AbsTol en tensiones
6
7  %%% BUCLE DE RESOLUCIÓN %%%
8
9  % Iniciar variables
10
11 sigmamin=tensmenos(1); % Extremo derecho del intervalo
12 sigmamax=tensmas(1); % Extremo izquierdo del intervalo
13 sigmaaux=(sigmamax+sigmamin)/2; % Punto medio del intervalo
14
15 if sigmamin==0 && sigmamax==0 % Caso en el que no hay tensiones normales
16
17     taumin=tensmenos(2); % Se interpola con tau en vez de sigma
18     taumax=tensmas(2);
19     tauaux=(taumax+taumin)/2;
20
21 while(taumin+Tol<taumax) % Así se evita la cancelación
22
23     tauaux=(taumax+taumin)/2;
24     qaux=calculaqDescos(0, tauaux, tensk, ck);
25
26     if ((qaux>0) && (qaux<qbuscado))
27         taumin=tauaux;
28     else
29         taumax=tauaux;
30     end
31 % El bucle va cerrando el intervalo hasta que la diferencia de los
32 % extremos del intervalo de tensiones en lo suficientemente pequeña <Tol
33 end
34
35 sigmaret=0; % Salida
36 tauret=tauaux; % Salida
37
38 else % Caso en el que sí hay tensiones normales
39
40 % Sigma como función de tau para interpolar solo con una variable
41 tau_aux=@(sigma)(tensmenos(2)+((tensmas(2)-tensmenos(2))/(tensmas(1)-
       tensmenos(1)))*(sigma-tensmenos(1)));
42
43 while(sigmamin+Tol<sigmamax) % Así se evita la cancelación
44
45     sigmaaux=(sigmamax+sigmamin)/2;
46     tauaux=tau_aux(sigmaaux);
47     qaux=CalculaqDescos(sigmaaux, tauaux, tensk, ck);
48
49     if ((qaux>0) && (qaux<qbuscado))
50         sigmamin=sigmaaux;
51     else
52         sigmamax=sigmaaux;
53     end
```

```
% El bucle va cerrando el intervalo hasta que la diferencia de los
% extremos del intervalo de tensiones es lo suficientemente pequeña <Tol
end

sigmaret=sigmaaux; % Salida
tauret=tau_aux(sigmaret); % Salida

end
end
```

Una vez que "BuscatensRet" ha calculado el valor de σ que se buscaba, el algoritmo de memoria llamará a "CalculatRet", que se encarga de interpolar el instante de tiempo en el que se alcanzaría σ_0 para poder incluirlo en el historial de cargas del punto. Esta función es muy simple y solo realiza la interpolación lineal que se explicó en el apartado 2.3. Para ello, debe recibir σ_0'', σ_0, Δt y t_0.

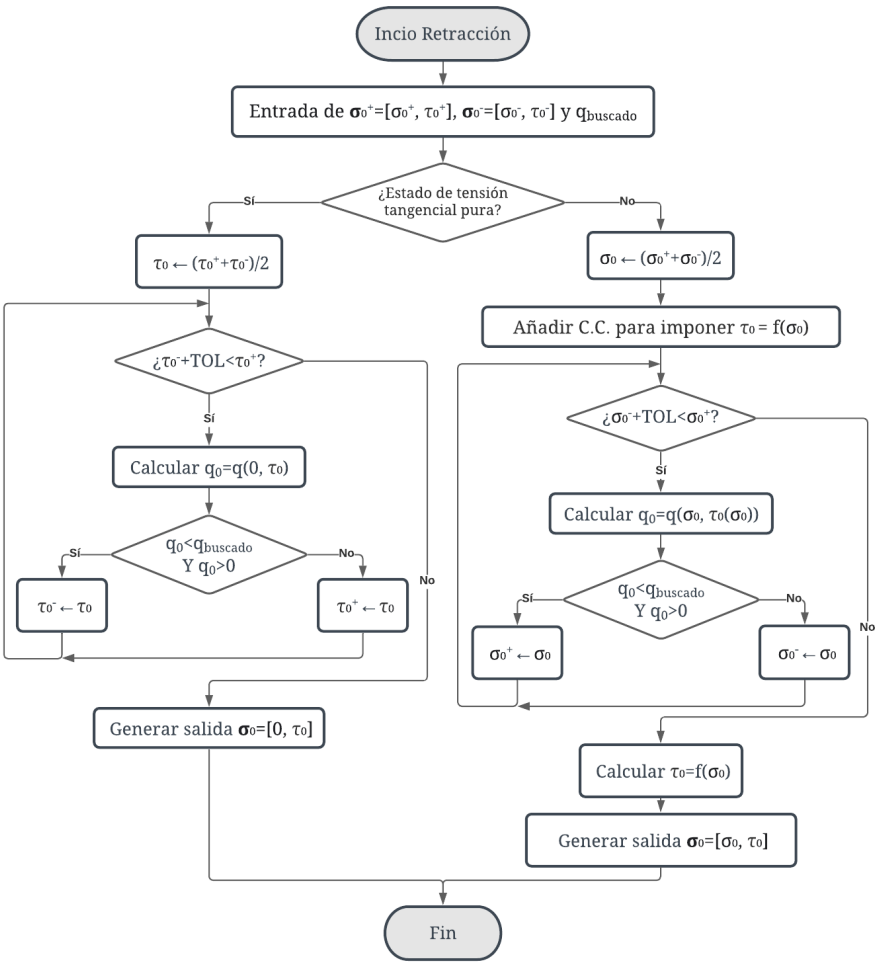

Figura 3.2 Diagrama de flujo del método de la bisectriz para interpolar tensiones.

Código 3.13 Función CalculatRet.

```
function tret=CalculatRet(tensret, tens, tensant, tant, deltaT)
% Esta función calcula el tiempo en el que ocurre la interpolación
```

```
% calculada en BuscatensRet al retraerse a un punto de memoria

sigmaret=tensret(1);tauret=tensret(2);

sigma=tens(1);tau=tens(2);

sigmaant=tensant(1);tauant=tensant(2);

% Se debe distinguir entre los casos en los que sigma(t)=0 y el resto
% al igual que se hizo en BuscatensRet

if sigmaant==0 && sigma==0

    coef=(tauret-tauant)/(tau-tauant);
    tret=tant+deltaT*coef;

else

    coef=(sigmaret-sigmaant)/(sigma-sigmaant);
    tret=tant+deltaT*coef;

end
end
```

Un aspecto muy importante a tener en cuenta es que cada vez que el bucle de integración principal realice una llamada a "BuscatensRet" y posteriormente a "CalculatRet", se está añadiendo un punto nuevo al historial de cargas de manera artificial. Esto no altera el algoritmo de integración, ya que el punto interpolado no se usa para integrar, pero sí que aumenta la longitud del vector de tensiones final. Es por esta razón por la que hay que guardar los resultados que se obtienen en un lugar diferente a las salidas del bucle de integración, al que le faltará la información correspondiente a la interpolación de valores cada vez que se detecta memoria.

3.2.8 Función dibujar

Una vez que se han realizado las operaciones de cálculo numérico, se ha conseguido generar una gran estructura de datos con toda la información acerca del punto geométrico que se está estudiando. Para poder entender fácilmente la información es necesario contar con una salida en forma de representación gráfica.

De esa manera, se ha decidido programar la función "dibujar", que recibe como argumentos los datos de la variable "punto1" y se encarga de realizar una animación en tiempo real. La animación generada cuenta con dos subgráficas que evolucionan de manera simultánea intentando ofrecer la mayor cantidad de datos posibles.

Con el fin de conseguir este efecto, se han utilizado los comandos "animatedline" para generar las gráficas y "addpoints" para ir introduciendo los puntos que se quieren dibujar en tiempo real. Se puede personalizar la visualización de la salida con el regulador de velocidad de la línea 14 del Código 3.14. Esto permite calcular cuál debe ser la pausa del programa (en la línea 95) para que el tiempo que dura la representación de la gráfica sea el elegido. Hay que remarcar que ese tiempo será aproximado y que depende de cada ordenador en el que se ejecute el programa.

Código 3.14 Función dibujar.

```
function dibujar(sigma, tau, tenst, ct, qt, tiempo, reg, mem, nombre)

%%% DIBUJO EN TIEMPO REAL DE LAS SALIDAS DEL PROGRAMA %%%

% Variables auxiliares

nval=length(tiempo);
j=1; % Contador del número de circunferencias que llevo hechas

% Para regular la velocidad se puede modificar el tiempo que se quiere
% que tarde la representación en segundos. Es un orden de magnitud ya
% que depende del ordenador en el que se ejecute.

duracion=10;

regulador=0.7*duracion/nval;

% Valores para los límites de la gráfica

xmax=sqrt(2/3)*max(sigma);if xmax==0; xmax=100; end
xmin=sqrt(2/3)*min(sigma);if xmin==0; xmin=-100; end
ymax=sqrt(2)*max(tau);if ymax==0; ymax=100; end
ymin=sqrt(2)*min(tau);if ymin==0; ymin=-100; end

% Para poner los ejes cuadrados para que no salgan como elipses (se
% puede poner como comentarios si da igual que salgan como elipses

if xmax>ymax; ymax=xmax; else; xmax=ymax; end
if xmin>ymin; xmin=ymin; else; ymin=xmin; end

%%% REALIZAR GRÁFICAS DE SALIDA %%%

% Gráficas de tensiones

figure(2)
hold on
subplot(1, 2, 1) % \sigma
xlim([0, tiempo(end)])
ylim([xmin, xmax])
axis square
title('Evolución de \sigma')
plot(tiempo, sigma);
xlabel('Tiempo [s]')
ylabel('\sigma')
grid on

subplot(1, 2, 2) % \tau
xlim([0, tiempo(end)])
ylim([ymin, ymax])
axis square
```

```
51 title('Evolución de \tau')
52 plot(tiempo, tau);
53 xlabel('Tiempo [s]')
54 ylabel('\tau')
55 grid on
56 hold off
57
58 % Gráficas en tiempo real %
59
60 % figure(1, 'WindowState', 'fullscreen') % Según la versión de MATLAB
61 f1=figure(1);
62 f1.Units='normalized';
63 f1.OuterPosition=[0 0 1 1];
64
65 subplot(1, 2, 2) % Evolución de q en tiempo real
66 xlim([0, tiempo(end)])
67 ylim([0, 1.2*real(max(qt))])
68 axis square
69 title('Evolución de q')
70 evolucion_q=animatedline('Color', 'b'); % Evolución de las tensiones
71 xlabel('Tiempo [s]')
72 ylabel('q')
73 grid on
74
75 subplot(1, 2, 1)
76 hold on
77 title(['Evolución de las tensiones // ', nombre])
78 tensiones=animatedline('Color','k', 'LineWidth', 1); % Evolución de q
79 centros=animatedline('Color', 'w', 'MarkerEdgeColor', 'r', 'Marker', 'x'
       );
80 tensiones_ref=animatedline('Color', 'w', 'MarkerFaceColor', 'r', 'Marker
       ', 'o');
81 xlim([1.2*real(xmin), 1.2*real(xmax)]) % Establece los límites de dibujo
82 ylim([1.2*real(ymin), 1.2*real(ymax)]) % Se pone real() por si hay
       inestabilidad en el cálculo
83 axis square
84 grid on
85 xlabel('\sigma\surd(2/3)')
86 ylabel('\tau\surd2')
87
88 for i=1:nval
89
90 % Evolución de las tensiones en la gráfica izquierda
91 addpoints(tensiones, sqrt(2/3)*real(sigma(i)), sqrt(2)*real(tau(i)));
92 drawnow limitrate
93
94 % Regulador de velocidad
95 pause(regulador)
96
97 % Evolución de q respecto a t en la gráfica derecha
```

```
98
99  addpoints(evolucion_q, tiempo(i), real(qt(i)));
100 drawnow limitrate
101
102 % Dibujo de los centros y los puntos tensk
103
104 addpoints(centros, sqrt(2/3)*real(ct(i, 1)), sqrt(2)*real(ct(i, 2)));
105 drawnow limitrate
106
107 addpoints(tensiones_ref, sqrt(2/3)*real(tenst(i, 1)), sqrt(2)*tenst(i,
        2));
108 drawnow limitrate
109
110 % Dibujo de las circunferencias
111
112 if reg(i)==1 % Si hay regresión dibujo el círculo que se acaba de formar
113
114 % Pongo las rojas debajo para que cuando borre las azules se queden
115 % como si fuesen de memoria en linea discontinua
116
117   if i==nval % Por cómo se guarda, la última circunferencia está aquí
118     Caux(j)=viscircles([sqrt(2/3)*ct(i, 1), sqrt(2)*ct(i, 2)], qt(i)/2,
          'Color', 'r', 'LineWidth', 0.5, 'LineStyle', '--');
119     C(j)=viscircles([sqrt(2/3)*ct(i, 1), sqrt(2)*ct(i, 2)], qt(i)/2, '
          Color', 'b','LineWidth', 0.5);
120     drawnow limitrate
121     else
122     Caux(j)=viscircles([sqrt(2/3)*ct(i, 1), sqrt(2)*ct(i, 2)], qt(i-1)/2,
           'Color', 'r','LineWidth', 0.5, 'LineStyle', '--');
123     C(j)=viscircles([sqrt(2/3)*ct(i, 1), sqrt(2)*ct(i, 2)], qt(i-1)/2, '
          Color', 'b', 'LineWidth', 0.5);
124   drawnow limitrate
125     end
126
127     j=j+1;
128
129 end
130 % Si hay memoria se cierra la circunferencia y elimina las anteriores
131 if mem(i)==2
132    delete(C(j-1));
133    j=j-1;
134 end
135
136 end
137
138 hold off
139
140 end
```

En la primera de las gráficas se muestra el plano de tensiones, donde, a medida que evolucionan

los valores del vector σ, se van representando las circunferencias calculadas en el bucle de memoria con sus respectivos centros. También se indican los valores de las tensiones de referencia que se han ido calculando en cada una de las iteraciones. En el momento en el que una circunferencia se "olvida" en el algoritmo debido a la detección de memoria, se puede apreciar que pasa a tener un contorno discontinuo. En la segunda gráfica se muestra la evolución de la distancia efectiva de tensiones a lo largo del tiempo. En todo momento, el valor instantáneo de q se corresponde con el de las tensiones de la subgráfica anterior.

También se genera una segunda figura en la que simplemente se muestra la evolución temporal de los valores de $\sigma(t)$ y $\tau(t)$ a lo largo del ciclo. La combinación de estas dos representaciones, además de ayudar a comprender el algoritmo, suponen una gran ayuda para cualquiera que necesite analizar los datos obtenidos.

3.3 Archivo "mainPOO.m"

Este fichero es el armazón del programa, en él están secuenciadas las tareas principales que hay que realizar y efectúa llamadas a la mayoría de las funciones auxiliares descritas en el apartado 3.2.

En las primeras líneas del Código 3.15 (1-47), el usuario ha de introducir el archivo donde se encuentra el historial de tensiones y se representan gráficamente las deformaciones para que se puedan verificar los datos. En la línea 43 se genera el objeto "punto1" instanciando la clase "punto". Este se ha denominado así porque es el único punto que se va a estudiar, como se comentó anteriormente. En caso de querer analizar varios puntos en la probeta se podría declarar un vector o una matriz de puntos. Para saber como instanciar el punto de manera correcta se recomienda leer la sección 3.4.2. Por último, en la línea 47 se puede introducir el valor de la banda de tolerancia del algoritmo de descargas si se desea.

Entre las líneas 53 y 58, el programa realiza una comprobación de los datos de entrada para que cuando $\varepsilon = 0$ y $\gamma = 0$ se pase a las siguientes componentes hasta encontrar una pareja de valores no nulos que no produzcan errores al calcular la matriz $\mathbf{A}(\sigma, \tau)$. En el índice i se guarda aquel primer valor del vector de deformaciones con ambas componentes no nulas, que será necesario para indicar en qué momento debe comenzar el bucle de resolución del PVI.

Antes de resolver el sistema se designan los valores iniciales tanto para la integración (condiciones iniciales) como para los algoritmos de memoria. Estos valores se guardan junto con los registros de deformaciones en el objeto "punto1". Para calcular las condiciones iniciales, se supone que el primer punto del registro (siempre que sea no nulo) es completamente elástico y, por tanto, se puede calcular σ usando la Ley de Hooke. Seguidamente, el primer valor de q se obtiene con la función de cálculo para cargas "CalculaqCarga". Las primeras tensiones de referencia y el primer centro se consideran $\sigma_k = [0, 0]$ y $\sigma_{c,k} = [0, 0]$ porque no hay circunferencias en el historial. Por último, se indica que es necesario continuar con el estado de cargas inicial declarando $prim = 1$.

Entre 82 y 103, las funciones que calculan derivadas devuelven estructuras interpoladas tipo spline en lugar de datos discretos para poder tener el valor de $d\varepsilon/dt$ en cualquier instante de tiempo que se requiera. Esto es necesario porque se desconoce cuáles van a ser los números que compondrán el vector de tiempo de resolución del PVI, ya que dependerá del paso de integración escogido por el usuario. Es posible elegir entre derivar con "CalculaDerivada5Puntos" o "CalculaDerivdaSpline", comentando las líneas según sea el caso. Al igual que con los datos anteriores, estas estructuras se cargan dentro del objeto "punto1".

A partir de la línea 105 comienza la integración del PVI. Entre las líneas 107 y 112 se definen los parámetros de la integración, que pueden ser modificados si se desea. Por defecto, el incremento temporal se va a fijar en 0.1 veces Δt_{def}. A continuación, se declara el Tablero de Butcher del par encajado (114-126) y, por último, se encuentra el bucle de integración (128-140). En este bucle se calculan las etapas según las expresiones (2.28) para cada instante de tiempos. Esto se hace llamando a los métodos que se han declarado en la clase "punto" y que se explican en el apartado 3.4. Como

se puede ver, en la primera etapa se usa "punto1.funcioncos" y en el resto "punto1.funcioncosaux" para que se apliquen correctamente los algoritmos.

Así, para cada paso de integración, siempre se comprobará si se han producido descargas o si es necesario aplicar memoria en el cálculo de la primera etapa k_1. Esto se debe a que en los métodos tipo Euler siempre se calcula la primera etapa de la forma $k_1 = f(t, y_n)$. Una vez que "funcioncos" detecte inversiones o memoria, si los hay, "funcioncosaux" genera el resto de etapas con las mismas ecuaciones de salida. Cuando se calcule el siguiente paso de integración y_{n+1} como una combinación lineal de las etapas, se le aplica el algoritmo en la siguiente iteración del bucle "while" en "funcioncos". Esta manera de actuar evita que el programa calcule las etapas dento de un mismo paso con ecuaciones o referencias de cargas diferentes para cada una de ellas. El cálculo de las etapas 2-7 debe ser siempre consecuente con el de la primera, utilizando exactamente las mismas ecuaciones.

Código 3.15 Archivo mainPOO.

```
clc, clear, close all

%%% INTRODUCIR NOMBRE DEL ARCHIVO DE DATOS %%%

nombre='def_cero'; % Sin extensión .txt
nombretxt=[nombre, '.txt'];

%%% DATOS MATERIAL %%

Young=datmat.Young;
K=datmat.K;
n=datmat.n;
G=datmat.G;

%%% LECTURA DE DATOS DE ENTRADA (deformaciones) %%%

[def_entrada, tdef]=Lectura(nombretxt); % [ndefx2, ndefx1]

% Se dibujan las deformaciones de entrada

figure(3)
subplot(1, 2, 1)
grid on
hold on
title('Def. entrada')
plot(def_entrada(:, 1), def_entrada(:, 2)/sqrt(3), 'b')
xlabel('\epsilon')
ylabel('\gamma\surd3')
hold off
subplot(1, 2, 2)
grid on
hold on
title('Def. entrada')
plot(tdef, def_entrada(:, 1), 'k')
plot(tdef, def_entrada(:, 2), 'g')
```

```
36 legend('\sigma', '\tau')
37 xlabel('Tiempo [s]')
38 ylabel('\epsilon, \gamma')
39 hold off
40
41 %%% INICIAR ESTRUCTURA PUNTO %%%
42
43 punto1=punto(tdef, def_entrada); % Se pasa el tiempo y defs. y
44                                  % se crea el objeto punto 1. Podría
45                                  % crearse un vector de puntos si se
46                                  % quieren analizar más puntos
47 punto1.TOL=0; % Tol. de detección (recomendado 0 o núm. muy pequeño)
48
49 %%% CONDICIONES INICIALES %%%
50
51 % Primer punto se considera elástico (sigma=[Young, 0; 0, G]*def)
52
53 i=1;
54 while(sum(def_entrada(i, :))==0) % Si amabs componentes son nulas
55     i=i+1; % En i se guarda el valor en el que
56 end        % empiezan las tensiones no nulas
57
58 tens0=[Young, 0; 0, G]*def_entrada(i, :)'; % 2x1
59
60 % Iniciar valores para el cálculo de q sucesivos
61
62 q0=CalculaqCarga(tens0(1), tens0(2));
63 ck0=[0, 0]; % El primer centro siempre va a ser [0, 0]
64 tensk0=[0, 0]; % Consideramos que es [0, 0]
65 prim=1;% Empieza con ec de cargas
66
67 punto1.tensk=tensk0;
68 punto1.ck=ck0;
69 punto1.qk=[]; % No hay memoria de qk por ahora
70
71 % Condiciones iniciales para el bucle de integración
72 % (Al final del bucle hay que eliminar el primer componente de cada uno
73 % de estos debido a que por como está programado se duplica). Es
74 % simplemente un formalismo que simplifica el algoritmo.
75
76 punto1.q=q0;
77 punto1.ttens=tdef(i);
78 punto1.sigma=tens0(1);
79 punto1.tau=tens0(2);
80 punto1.prim=prim;
81
82 %%% CALCULA DERIVADAS %%%
83 % (Utilizar solo uno de los métodos y comentar el otro)
84
85 % Pedir splines de derivadas con fnder
```

```
86  % (se pasan los valores no nulos para que no haya interferencias
87  % con las pendientes del spline)
88
89  [ppdepsilon, ppdgamma]=CalculaDerivadaSpline(def_entrada(i:end, :), tdef
        (i:end));
90
91  % Pedir splines de derivadas según fórmula de 5 puntos
92
93  % [ppdepsilon, ppdgamma]=CalculaDerivada5Puntos(def_entrada, tdef);
94
95  % Pasar datos al objeto (se pasan como salidas del spline para ser
96  % evaluadas en cualquier momento. Así se deja y'(t) como función de t
97  % en cualquier instante donde poder evaluar la derivada
98
99  % En el caso en el que se quiera represtnar la derivada simplemente se
100 % evalúa el spline: def derivada=ppval(punto1.ppdepsilon, punto1.tdef)
101
102 punto1.ppdepsilon=ppdepsilon;
103 punto1.ppdgamma=ppdgamma;
104
105 %%% INTEGRACIÓN %%%
106
107 % Parámetros de integración
108
109 y(:,1)=tens0;
110 h=(tdef(2)-tdef(1))/10; % Paso de integración
111 tintegracion=tdef(i):h:tdef(end);
112 np=length(tintegracion);
113
114 % Tablero de Butcher
115
116 c=[1/5, 3/10, 4/5, 8/9, 1, 1];
117
118 A=[1/5      0          0         0         0          0
119    3/40     9/40       0         0         0          0
120    44/45    -56/15     32/9      0         0          0
121    19372/6561 -25360/2187 64448/6561 -212/729 0         0
122    9017/3168 -355/33   46732/5247 49/176   -5103/18656 0
123    35/384   0          500/1113 125/192  -2187/6784 11/84
124    ];
125
126 b=[35/384 0 500/1113 125/192 -2187/6784 11/84 0];
127
128 % Bucle de integración
129
130 for k=1:np % En la primera etapa se aplica memoria
131      k1=punto1.funcioncos(tintegracion(k), y(:,k));
132 % Aquí no hay que aplicar memoria porque guarda valores basura
133      k2=punto1.funcioncosaux(tintegracion(k)+h*c(1), y(:,k)+h*k1*A(1,1))
         ;
```

```
134     k3=punto1.funcioncosaux(tintegracion(k)+h*c(2), y(:,k)+h*([k1 k2]*A
            (2, 1:2)'));
135     k4=punto1.funcioncosaux(tintegracion(k)+h*c(3), y(:,k)+h*([k1 k2 k3
            ]*A(3, 1:3)'));
136     k5=punto1.funcioncosaux(tintegracion(k)+h*c(4), y(:,k)+h*([k1 k2 k3
             k4]*A(4, 1:4)'));
137     k6=punto1.funcioncosaux(tintegracion(k)+h*c(5), y(:,k)+h*([k1 k2 k3
             k4 k5]*A(5, 1:5)'));
138     k7=punto1.funcioncosaux(tintegracion(k)+h*c(6), y(:,k)+h*([k1 k2 k3
             k4 k5 k6]*A(6, 1:6)'));
139     y(:,k+1)=(y(:,k)+h*[k1 k2 k3 k4 k5 k6 k7]*b'); % Método R-K p=5
140 end
141
142 % Por como está programado, hay que eliminar estos valores para que los
143 % vectores tengan la longitud real y no haya contenido duplicado
144 % (se corresponden con los primeros valores del bucle del PVI)
145
146 punto1.ttens(1)=[];
147 punto1.sigma(1)=[];
148 punto1.tau(1)=[];
149 punto1.q(1)=[];
150 punto1.prim(1)=[];
151
152 % Se van a añadir los puntos de tensión en el caso en el que i no sea
153 % cero y por tanto haya un instante de tiempo en el que todo es nulo
154
155 if i>1
156
157 punto1.ttens=[tdef(1:i-1); punto1.ttens];
158
159 ceros1=zeros(i-1, 1);
160 ceros2=zeros(i-1, 2);
161
162 punto1.q=[ceros1; punto1.q];
163 punto1.sigma=[ceros1; punto1.sigma];
164 punto1.tau=[ceros1; punto1.tau];
165 punto1.ck_hist=[ceros2; punto1.ck_hist];
166 punto1.tensk_hist=[ceros2; punto1.tensk_hist];
167 punto1.mem=[ceros1; punto1.mem];
168 punto1.reg=[ceros1; punto1.reg];
169 punto1.prim=[ceros1; punto1.prim];
170
171 end
172
173 % Se va a calcular la última circunferencia fuera del bucle (solo hay
174 % que calcular el último centro y poner reg=1 para indicar que se ha
175 % hecho la última de las circunferencias)
176
177 punto1.reg(end)=1;
178
```

```
179 if punto1.prim(end)==0
180     normaaux=sqrt((2/3)*(punto1.ck_hist(end-1, 1)-punto1.tensk_hist(end
            -1, 1))^2+2*(punto1.ck_hist(end-1, 2)-punto1.tensk_hist(end-1, 2)
            )^2);
181     punto1.ck_hist(end,:)=punto1.tensk_hist(end,:)+(punto1.ck_hist(end
            -1,:)-punto1.tensk_hist(end-1,:))/normaaux*(punto1.q(end)/2);
182 end
183 if punto1.prim(end)==1
184     punto1.ck_hist(end, :)=[0, 0];
185 end
186
187 %%% GRÁFICAS %%%
188
189 dibujar(punto1.sigma, punto1.tau, punto1.tensk_hist, punto1.ck_hist,
        punto1.q, punto1.ttens, punto1.reg, punto1.mem, nombre)
190
191 % Guardar en formato .jpeg (comentar si no se quieren guardar)
192 %
193 % nombresalida1=['Resultado (', nombre, ').jpg'];
194 % saveas(1, nombresalida1)
195 %
196 % nombresalida2=['Tensiones (', nombre, ').jpg'];
197 % saveas(2, nombresalida2)
198
199 %%% CONSTRUCCIÓN DE MATRIZ DE ANÁLISIS %%%
200 muestra=[punto1.ttens, punto1.sigma, punto1.tau, punto1.q, punto1.
        tensk_hist, punto1.ck_hist, punto1.reg, punto1.mem, punto1.prim];
201
202 % filename = 'Datos_punto.xlsx';
203 % writematrix(muestra,filename,'Sheet',1,'Range','A2')
```

Tras la integración se calcula el centro para el último valor del historial de cargas (ya que no ocurre dentro del bucle) y se eliminan los valores iniciales del historial debido a que están repetidos. Esto ocurre por cómo funciona internamente el algoritmo, simplemente hay que tenerlo en cuenta y borrarlos una vez que han cumplido su función (líneas 142-150).

Para aquellas situaciones en las que se hayan detectado deformaciones nulas al inicio del historial, se van a agregar ceros en los vectores del objeto "punto1" delante de los valores calculados. Esta tarea se realiza para que los intervalos de tiempo de deformaciones y tensiones tengan los mismos puntos de inicio y fin, además de para poder representarlos de manera gráfica adecuadamente. Estas líneas realizan un simple formalismo numérico para paliar el hecho de que el programa no puede recibir deformaciones nulas.

Al final, se genera la salida gráfica haciendo una llamada a la función "dibujar" y se ordenan las variables en una matriz tabla llamada "muestra" para poder analizar los datos numéricos si se quiere. Las filas de "muestra" representan el valor de cada una de las variables históricas para un instante de tiempo determinado en el orden mostrado en la expresión (3.9), lo que se corresponde con unas dimensiones $n_{tens}x11$.

$$muestra = [t_{tens} \;\; \sigma \;\; \tau \;\; q \;\; \sigma_k \;\; \tau_k \;\; \sigma_{c,k} \;\; \tau_{c,k} \;\; reg \;\; mem \;\; prim] \tag{3.9}$$

3.4 Clase tipo punto de MATLAB

Para guardar y modificar la evolución de las variables que calcula el programa se ha creado un fichero tipo "class", a partir del cual van a derivar los objetos que representan los datos del historial de cargas en un punto. Esta clase contiene una serie de propiedades que se corresponden con los registros de la información de un punto geométrico del material, además de contar con los métodos que aplican las ecuaciones elasto-plásticas y que realizan el algoritmo de memoria e inversiones.

3.4.1 Propiedades de la clase

Las propiedades de la clase serán vectores que guarden la información en cada instante de tiempo que se evalúe en la resolución. Con el afán de proporcionar una información completa al usuario, se han querido incluir todos los registros posibles dentro del objeto, ya sean datos de entrada, variables auxiliares o la propia solución del PVI. Así, una vez ejecutado el fichero "main", se podrá acceder a los datos que se necesite a través del objeto instanciado a partir de esta clase.

Código 3.16 Propiedades de la clase punto.

```
    properties
        tdef % vector de tiempos de entrada (tiempos de defs)
        def % deformaciones de entrada
        ppdepsilon % estructuras tipo splines para derivadas
        ppdgamma
        ttens % vector de tiempos de tens (t_sal de la integración)
        sigma % tensiones de salida de la integración
        tau
        q % vector de q calculado para cada una de las tensiones
        qk % vector auxiliar con los valores de q en memoria
        ck % centros correspondientes a los q anteriores
        tensk % centros correspondientes a los q anteriores
        ck_hist % valores de centros para cada valor de ttens
        tensk_hist % valores de tensk para cada valor de ttens
        mem % bandera de memoria en cada ttens
% (0=no memoria, 1=memoria, 2=tens calculada mediante retracción)
        reg % bandera de regresión para cada ttens
% (1=se ha detectado regresión en ese punto)
        bandera % bandera auxiliar para funcioncosaux
        prim % (1=cargas, 0=descargas)

        % Variables para tolerancia de detección

        qregaux; % q de referencia para banda de tolerancia
        regaux; % bandera auxiliar del intervalo de regresión
        TOL; % amplitud de la banda de tolerancia;

    end
```

Todas las propiedades mostradas en el Código 3.16 son vectores que representan la evolución a lo largo del tiempo de alguna variable, ya sea una propiedad física o una bandera auxiliar. La única propiedad que no será un vector al final del bucle es la llamada "bandera". La explicación de cada

una de ellas y qué tipo de datos guardan, además de la dimensión que van a tener una vez se termine de ejecutar el programa, se describe a continuación.

- **"tdef"** (n_{def}x1): Es el vector de instantes de tiempo leído junto con las deformaciones de entrada, viene dado por los datos del historial de carga y no se modifica durante el programa. Se guarda por si es necesario representar las deformaciones.
- **"def"** (n_{def}x2): Deformaciones de entrada $\varepsilon(t) = [\varepsilon(t), \gamma(t)]$. Al igual que en el caso anterior, este historial no se modifica a lo largo del programa.
- **"ppdepsilon"** (estructura tipo spline): Estructura que puede evaluarse mediante el comando "ppval" para cualquier valor de t y que devuelve el valor de la derivada de la primera componente del vector de deformaciones en dicho instante de tiempo. Se puede entender analíticamente como una función interpolada $d\varepsilon(t)/dt$.
- **"ppdgamma"** (estructura tipo spline): Igual que "ppdepsilon", pero para la derivada de la segunda componente del vector de deformaciones ($d\gamma(t)/dt$).
- **"ttens"** (n_{tens}x1): Es el vector de instantes de tiempo de cada uno de los pasos de integración más los instantes interpolados por "BuscatensRet" y "CalculatRet". Sus extremos son los mismos que los de $tdef$, pero generalmente será más largo debido a que los puntos de precisión suelen estar más próximos para el caso de tensiones.
- **"sigma"** (n_{tens}x1): Valores de la primera componente del vector de tensiones σ calculados por el programa, incluidos los valores interpolados.
- **"tau"** (n_{tens}x1): Igual que "sigma", pero para la segunda componente del vector de tensiones τ.
- **"q"** (n_{tens}x1): Guarda los cálculos de la distancia efectiva de tensiones para cada uno de los valores de σ y τ anteriores.
- **"ck_hist"** (n_{tens}x2): Almacena los centros de las circunferencias de carga ($\sigma_{c,k}$) que se han utilizado en cada punto del historial para calcular q. Este valor será el mismo a lo largo de una serie de datos consecutivos hasta que se produzca una inversión o detección de memoria, cuando cambiará.
- **"tensk_hist"** (n_{tens}x2): Funciona igual que "ck_hist", pero esta vez hace acopio de los datos de σ_k.
- **"qk"** ((b-1)x1): Es una de las tres variables auxiliares de memoria. En ella se guardan los valores de los diámetros de las circunferencias que están almacenadas en el historial de cargas. En caso de producirse una inversión, se añade un nuevo valor a este vector y, cuando se tenga que "olvidar" una circunferencia, se borra el último dato de este vector. Es por eso que su longitud no está determinada y depende del historial de cargas.
- **"ck"** (bx2): Registro análogo a "qk", pero para los valores de los centros de las circunferencias $\sigma_{c,k}$ en lugar de para los diámetros.
- **"tensk"** (bx2): De manera parecida a las dos propiedades anteriores, guarda los valores de las tensiones de referencia σ_k. Para que el programa funcione bien, "tensk" y "ck" tienen un punto extra más que "qk", ya que la primera tensión de referencia y centro siempre serán el punto [0, 0].
- **"reg"** (n_{tens}x1): Variable booleana tipo bandera que indica en qué instante de tiempo se ha detectado una regresión, es decir, $dq/dt < 0$. Tiene el valor 1 en caso afirmativo y 0 en caso de que no haya detectado nada.

- **"mem"** (n_{tens}x1): Es parecida a "reg", pero esta bandera detecta cuándo se aplica el efecto de memoria, es decir, cuándo se borra una de las circunferencias de memoria debido a que se ha producido $q > q_k(end)$ o $q < 0$. Tiene el valor 0 si no se ha localizado memoria, 1 si se ha detectado y 2 en caso de que ese valor sea interpolado al detectar memoria, es decir, calculado por "BuscatensRet".
- **"bandera"** (1x1): Es una bandera auxiliar que se modifica por el método "funcioncos" y que posteriormente se usa por "funcioncosaux" para saber qué tipo de ecuaciones de salida debe usar: elásticas, elasto-plásticas para cargas o elasto-plásticas para descargas.
- **"prim"** (n_{tens}x1): Funciona igual que "reg", pero en este caso indica si en ese punto se han utilizado las ecuaciones para cargas o para descargas. Su valor es 1 para los instantes de tiempo en los que se usen las funciones de carga y 0 para descargas.

3.4.2 Método para instanciación

Es necesario elaborar un método que se encargue de declarar en el archivo principal un objeto que derive de esta clase. Ya que se conoce el historial de deformaciones, este se va a usar como datos de entrada para la instanciación. Asimismo, este método inicia las propiedades de la banda de tolerancia.

Código 3.17 Método para declarar un objeto tipo punto.

```
1        function obj=punto(tdef, def) % Genera el punto cargando defs
2            obj.tdef=tdef;
3            obj.def=def;
4            obj.regaux=0; % Iniciar en 0
5            obj.qregaux=-1; % Inciar en -1
6        end
```

El nombre de este método, a diferencia de los siguientes, debe ser el mismo que el nombre de la clase, ya que se usa para instanciar el objeto. En el Código 3.17 se hace referencia a la variable "obj", que es la predefinida por MATLAB para modificar el objeto que se ha generado con la función.

3.4.3 Método "funcioncos"

Este método constituye el corazón del programa, el lugar en el que se encuentra el algoritmo que realiza el bucle de inversiones y memoria, donde se modifican las variables que guardan información y en el que se generan las salidas de las ecuaciones diferenciales. Es importante remarcar que esta es la única función que modifica las propiedades explicadas anteriormente en todo el programa, ya sea para guardar nuevos datos en el historial o para alterar los datos de las circunferencias guardadas en memoria. El resto de códigos solo realizan labores de lectura de datos o cálculos secundarios y nunca modifican la memoria. El hecho de asegurar que "funcioncos" tiene capacidad exclusiva para hacer modificaciones es crucial para evitar generar datos basura en cada iteración.

Como se aprecia en la primera línea del Código 3.18, solo se necesita como argumento el instante t y el vector $\boldsymbol{\sigma}$ correspondientes al paso de integración actual. Esto se ha diseñado así para que sea análogo a un sistema de ecuaciones del tipo $y' = f(t, y(t))$ y que se pueda resolver mediante un método de resolución paso a paso, igual que los explicados en el apartado 2.5. Como es de esperar, la salida que genera es un vector de dos componentes que se corresponde con el valor de k_1, es decir, la primera etapa de la fórmula de DOPRI5(4). Lo verdaderamente importante de este método no es solo la salida en sí misma, sino todas las modificaciones que realiza en el algoritmo de inversiones y memoria para poder guardar la información del historial de cargas del material.

Esta función consta de cuatro partes principales que ejecutan tareas diferenciadas entre sí y que se explican seguidamente. Para poder ver esquemáticamente estos algoritmos, se adjunta el diagrama de flujo de la figura 3.3, que resume gráficamente las líneas de código.

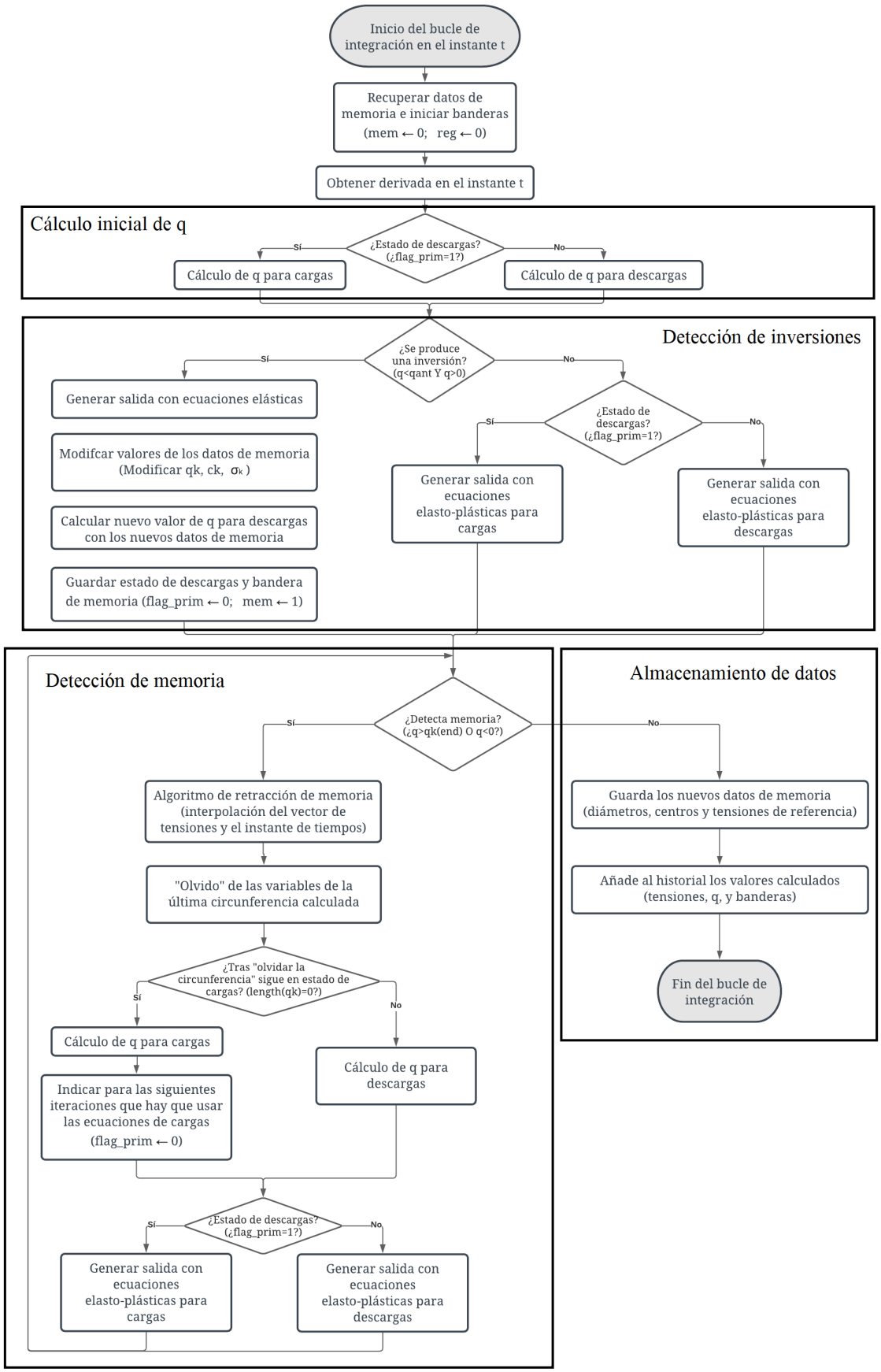

Figura 3.3 Diagrama de flujo del bucle de inversiones y memoria.

- **Obtención de datos y cálculo inicial de q** (líneas 1-42): Se recuperan las variables que guardan los datos de memoria de las circunferencias del objeto y se guardan en variables locales. Se podrían haber modificado directamente sobre el objeto, pero se ha decidido realizar los cambios de forma separada para detectar y arreglar más rápidamente posibles errores. También se calcula la derivada en el instante t_0 mediante los splines y "ppval" y se recuperan las propiedades del material que serán necesarias para generar el resultado. En las líneas 34-42 se realiza el primer cálculo de q en ese instante de tiempo para luego comparar y utilizarlo en la detección de memoria e inversiones.
- **Algoritmo de detección de inversiones** (líneas 44-116): Esta parte detecta los casos en los que $dq/dt < 0$ y se produce una inversión. Como se puede ver, se obliga a $q > 0$ porque, aunque inicialmente suponga $dq/dt < 0$, realmente se trata de una memoria (una circunferencia ha llegado a ser más grande que la anterior) en lugar de una inversión. Llegado este caso, hay que generar una nueva circunferencia, calculando y añadiendo a la memoria nuevos valores de q_k, σ_k y $\sigma_{c,k}$, y hallar el valor q actual para las nuevas referencias de cargas (líneas 79-98). Adicionalmente, se tiene que dar una salida con las ecuaciones elásticas (líneas 74-77) por ser el primer punto para el que se detecta inversión. Si no hubiese regresiones, simplemente se genera una salida con las ecuaciones elasto-plásticas que correspondan (líneas 100-114).
- **Bucle de detección de memoria** (líneas 118-198): Este bucle funciona siempre que $q > q_k$ o $q < 0$, es decir, cuando la evolución de las tensiones conduce a que se sobrepase una circunferencia y haya que borrarla del historial aplicando memoria de cargas. Se ha utilizado un bucle "while" en lugar de un "if" porque es posible que el incremento de tiempo sea lo suficientemente grande como para saltar dos circunferencias o más en lugar de solo una. Esto suele ocurrir en evoluciones uniaxiales o para saltos de Δt amplios, casos en los que, para que el programa funcione como es debido, es necesario utilizar un comando "while". La figura 3.4 ejemplifica el suceso descrito, donde el incremento de tiempo entre σ y σ' es lo suficientemente extenso como para sobrepasar la memoria de dos circunferencias y se volvería al estado de cargas inicial. El primer cálculo que se realiza en este bucle es la interpolación del punto para el que se alcanza justamente q_k siguiendo la evolución de las cargas (131-151). Posteriormente, se "borran" del registro de memoria los valores de la circunferencia que se ha superado (153-158) y, por último, se calcula el nuevo q y se devuelve la salida. Es necesario diferenciar los casos para los que se detecta memoria y se sigue en un estado de descargas (173-180), y aquellos en los que se rebasan todas las circunferencias almacenadas y se vuelve a la ecuación de cargas (160-171). La diferencia entre ambos casos es que la forma de calcular q y la de establecer las banderas para iteraciones posteriores debe ser diferente.
- **Generar salida y acopio de datos** (líneas 200-223): Al final de esta función, se guardan en el objeto los registros de memoria de las circunferencias para usarlos en la siguiente iteración. Adicionalmente, se recopilan los valores ampliando el historial de cargas (también dentro del objeto) para su posterior análisis.

Código 3.18 Método funcioncos.

```
1 function sal=funcioncos(obj, t, y)
2
3 % Se van a poner las variables locales con subíndices _i para
4 % que no haya confusión con las propiedades de la clase
5
6       sigma_i=y(1);tau_i=y(2);
7
```

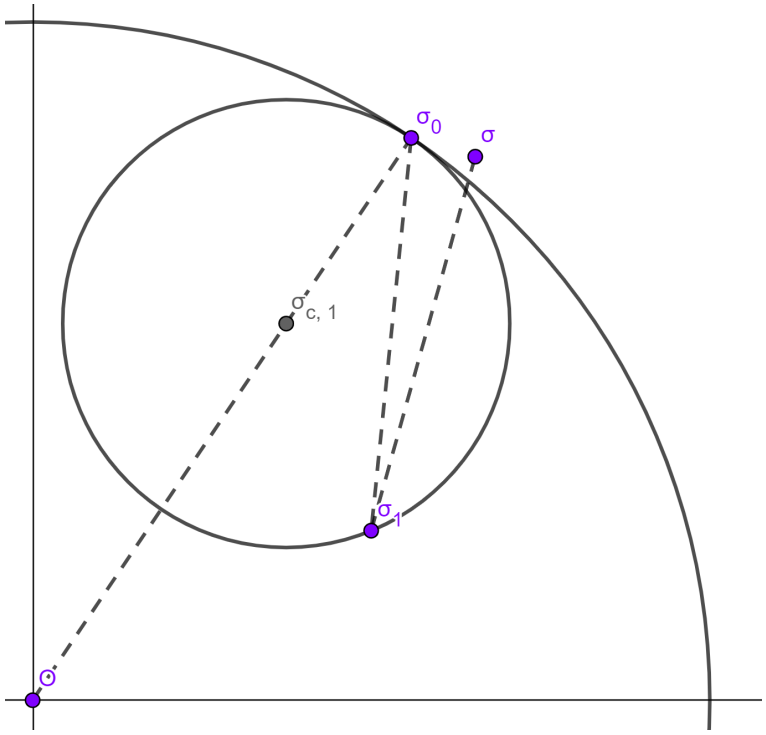

Figura 3.4 Detección de memoria de dos circunferencias en un mismo paso de integración.

```
8  % Poner banderas a cero
9
10         mem_i=0;
11         reg_i=0;
12
13 % Datos del material necesarios
14
15         Young=datmat.Young;
16         G=datmat.G;
17
18 % Datos de iteraciones anteriores. Se podrían modificar directamente
19 % pero se va a trabajar con variables locales _i por ser más cómodas)
20
21         qant=obj.q(end);
22         tensant(1)=obj.sigma(end); % 1x2
23         tensant(2)=obj.tau(end); % 1x2
24         ck_i=obj.ck; % mx2
25         tensk_i=obj.tensk; % mx2
26         qk_i=obj.qk; % mx1
27         prim_i=obj.prim(end);
28
29 % Obtener derivada de la deformación en el instante de tiempo t
30
31         DerDef(1, 1)=ppval(obj.ppdepsilon, t);
32         DerDef(2, 1)=ppval(obj.ppdgamma, t);% 2x1
33
34         %%% CÁLCULO DE q %%%
```

```
35
36          if prim_i==1 % Caso cargas
37              q_i=CalculaqCarga(sigma_i, tau_i);
38          end
39
40          if prim_i==0 % Caso descargas
41              q_i=CalculaqDescos(sigma_i, tau_i, tensk_i(end, :), ck_i(end,
                    :));
42          end
43
44          %%% DETECCIÓN DE REGRESIONES %%%
45
46            inversion=0
47  % Tolerancia para inversiones
48
49           if q_i<qant && q_i>0 && obj.regaux==0
50               obj.regaux=1;
51               obj.qregaux=qant;
52           end
53           if obj.regaux==1 && q_i>0
54           if q_i>obj.qregaux
55              obj.regaux=0;
56           end
57           if q_i+obj.TOL<obj.qregaux
58              inversion=1;
59              obj.regaux=0;
60           end
61           end
62
63  % Si no se quiere utilizar la TOL (ni siquiera 0) se descomenta
64  % este apartado y se comenta el anterior
65
66  %         if q_i>0 && q_i<qant
67  %            inversion=1;
68  %         end
69
70          if inversion==1 % q menor que el último punto
71                          % guardado y además debe ser mayor que cero
72          reg_i=1; % Bandera de regresión
73
74  % Al detectar regresión se usan las ecs elásticas para la salida
75
76          sal=[Young, 0; 0, G]*DerDef; % 2x1
77          bandera_i=0; % Para usar la ec elástica en funcioncosaux
78
79  % Modificar los datos de referencia para cálculos de q posteriores
80
81  % Si son descargas sucesivas se calcula el centro con suma de vectores
82          if prim_i==0
```

```
83            normaaux=sqrt((2/3)*(ck_i(end, 1)-tensk_i(end, 1))^2+2*(ck_i(
                  end, 2)-tensk_i(end, 2))^2);
84            ck_i(end+1,:)=tensk_i(end,:)+(ck_i(end,:)-tensk_i(end,:))/
                  normaaux*(qant/2);
85        end
86
87 % Si ha sido la primera descarga se mantendría en ck=[0, 0]
88        if prim_i==1
89            prim_i=0; % Bandera a cero para siguientes iteraciones
90            ck_i(end+1, :)=[0, 0];
91        end
92
93 % Se añade una fila más a los valores de tensk en memoria
94        tensk_i(end+1, :)=tensant;
95 % Cálculo del nuevo q con los nuevos valores tras la regresión
96        q_i=CalculaqDescos(sigma_i, tau_i, tensant, ck_i(end, :));
97 % Se añade una fila más a los valores de qk en memoria
98        qk_i(end+1)=qant;
99
100       else % Caso de no detectar regresión
101
102 % Se genera la salida con las ecuaciones elasto-plásticas
103 % Según sea el caso de carga o descarga hay diferente matriz A
104       if prim_i==1 % usa matriz A cargas
105           A=GeneraAQ(sigma_i, tau_i, q_i/2); % Q=q/2
106           sal=A*DerDef; % 2x1
107           bandera_i=1;
108       end
109
110       if prim_i==0 % usa matriz A para descargas
111           A=GeneraADes(sigma_i, tau_i, q_i, tensk_i(end, :), ck_i(end,
                  :), qk_i(end));
112           sal=A*DerDef; % 2x1
113           bandera_i=2;
114       end
115
116       end
117
118       %%% EFECTO DE MEMORIA %%%
119
120 % Si length(qk)==1 no puede haber memoria ya que es la primera
121 % carga de todo el histórico
122
123       while (~isempty(qk_i)) && ((q_i>obj.TOL+qk_i(end)) || (q_i+obj.
              TOL<0))
124
125           mem_i=1; % bandera de memoria
126
127 % Si el q actual es mas grande que el último q guardado se entra en el
128 % bucle de memoria, aquí se han cambiado los if por while en el caso
```

```
% en el que se salte varias esferas

% Hay que retraerse al punto donde q=qk (y guardo los valores)
% también se interpola el instante de tiempo en el que ocurre

% Interpolación de las tensiones
          [sigmaret, tauret]=BuscatensRet(tensant, [sigma_i,tau_i],
             tensk_i(end, :), ck_i(end, :), qk_i(end));
% Interpolación del instante de tiempo
          tant=obj.ttens(end);
          deltaT=t-tant;
          tret=CalculatRet([sigmaret, tauret], [sigma_i, tau_i],
             tensant, tant, deltaT);

% Guardo los datos en las variables del objeto

          obj.q(end+1)=qk_i(end);
          obj.sigma(end+1)=sigmaret;
          obj.tau(end+1)=tauret;
          obj.tensk_hist(end+1, :)=tensk_i(end, :);
          obj.ck_hist(end+1, :)=ck_i(end, :);
          obj.mem(end+1)=2; % indica algoritmo de retracción
          obj.reg(end+1)=reg_i;
          obj.ttens(end+1)=tret;
          obj.prim(end+1)=prim_i;

% Actualizar valores para el nuevo cálculo de q ("olvida" los valores de
% la circunferencia anterior porque aplica memoria)

          tensk_i(end, :)=[]; % "Olvida" las tensiones de referencia
          ck_i(end, :)=[]; % "Olvida" el centro de la circunferencia
          qk_i(end)=[];% "Olvida" el diámetro de la circunferencia

          if (length(qk_i))==0 % Usar ec de cargas porque se ha
% superado el número de circunferencias como para tener que volver a
% usar estas ecuaciones (se han quedado sin valores en qk)

% Hay que volver a usar la ec de cargas porque se ha superado la primera
% de todas las circunferencias del estado de carga al aplicar memoria

          q_i=CalculaqCarga(sigma_i, tau_i);

          prim_i=1; % volver a usar ec. de cargas

          end

          if (length(qk_i))>0 % Usar ec. para descargas (caso en el que
                              % hay valores de memoria en qk)

% Calcular nuevo q con ec. para descargas y los nuevos datos tensk y ck
```

```
177
178             q_i=CalculaqDescos(sigma_i, tau_i, tensk_i(end, :), ck_i(end,
                    :));
179
180             end
181
182 % Salida con las ecuaciones elasto-plásticas tras aplicar la memoria
183 % (una vez que se han interpolado todos los puntos intermedios se
184 % calcula la salida con las ecuaciones que correspondan).
185
186             if prim_i==1 % Usa matriz A cargas
187                 A=GeneraAQ(sigma_i, tau_i, q_i/2);
188                 sal=A*DerDef; % 2x1
189                 bandera_i=1;
190             end
191
192             if prim_i==0 % Usa matriz A descargas
193                 A=GeneraADes(sigma_i, tau_i, q_i, tensk_i(end, :), ck_i(
                        end, :), qk_i(end));
194                 sal=A*DerDef; % 2x1
195                 bandera_i=2;
196             end
197
198         end
199
200 %%% GUARDADO DE DATOS PARA ITERACIONES POSTERIORES %%%
201
202         obj.bandera=bandera_i; % Bandera para funciones auxiliares
203
204 % Datos históricos del punto (guardo en el objeto)
205
206         obj.q(end+1, :)=q_i;
207         obj.sigma(end+1, :)=sigma_i;
208         obj.tau(end+1, :)=tau_i;
209         obj.tensk_hist(end+1, :)=tensk_i(end, :);
210         obj.ck_hist(end+1, :)=ck_i(end, :);
211         obj.mem(end+1, :)=mem_i;
212         obj.reg(end+1, :)=reg_i;
213         obj.ttens(end+1, :)=t;
214
215 % Datos para la siguiente iteración (los voy a rescatar luego
216 % porque me sirven de apoyo en el algoritmo)
217
218         obj.qk=qk_i;
219         obj.tensk=tensk_i;
220         obj.ck=ck_i;
221         obj.prim(end+1, :)=prim_i;
222
223         end
```

Sería posible realizar este bucle, pero en lugar de efectuar los cálculos con el producto escalar y el coseno, se harían con la ecuación (3.2.4) desarrollada. Para el programa de variables globales sí que se adjunta "funcionqk", que utiliza "CalculaqDesqk", pero como ya se comentó, se ha optado por mantener "CalculaqDescos" por defecto.

Otro hecho importante que hay que tener en cuenta es que en esta función se ha incluido una tolerancia que abre un margen a la hora de detectar inversiones o memorias (líneas 47-61). Esto está indicado para aquellas situaciones en las que los estados de carga produzcan variaciones inestables en el valor de q y en los que se generen pequeñas circunferencias inexistentes al detectar inversiones. En un principio se va a definir como nula, pero es posible modificarla a criterio del usuario en el método de instanciación. En cualquier caso, no se recomienda utilizar valores excesivamente grandes porque se puede adulterar la solución final. En la figura 3.5 se representa el diagrama de flujo que indica cómo funciona esta tolerancia.

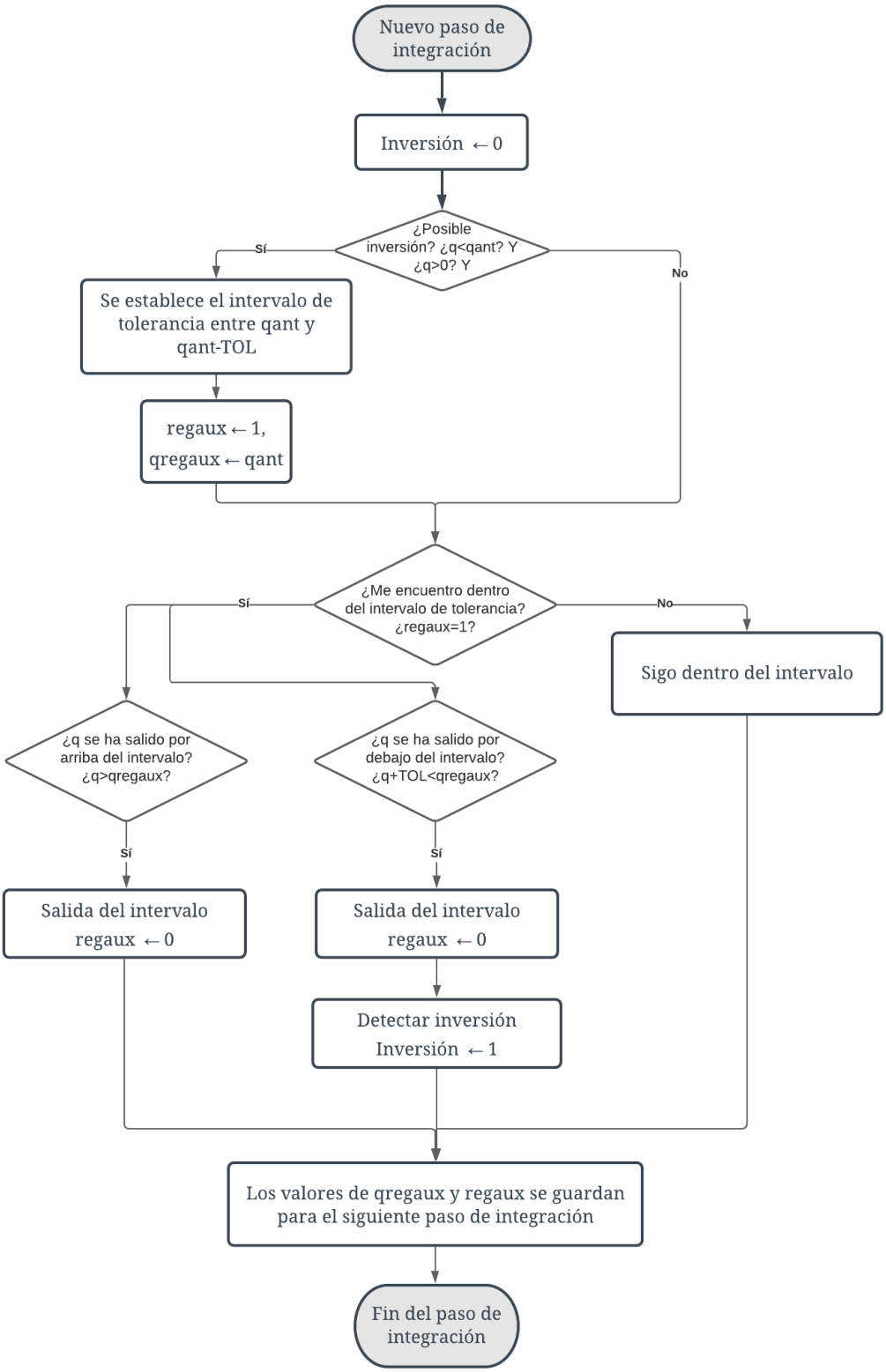

Figura 3.5 Diagrama de flujo del funcionamiento de la banda de tolerancia de inversiones.

3.4.4 Método "funcioncosaux"

Ya se ha comentado que es necesario que el algoritmo de memoria se ejecute una única vez en cada paso de la resolución del sistema de ecuaciones diferenciales. A pesar de esto, el método DOPRI5(4) cuenta con siete etapas y, por tanto, es necesario generar siete salidas diferentes para calcular k_i. Es por eso por lo que se ha utilizado el método extra "funcioncosaux", que solo genera la salida con las mismas ecuaciones usadas por "funcioncos" sin modificar la memoria del objeto "punto1".

Para implementar este sistema, es necesaria una variable que comunique información entre "funcioncos" y "funcioncosaux" para que ambas generen salidas con las mismas ecuaciones: elásticas, elasto-plásticas para cargas o elasto-plásticas para descargas, según el caso. Esto se consigue con la propiedad "bandera", explicada en 3.4.1. El diagrama de flujo de la figura 3.6 representa la evolución que sufre esta variable dentro de "funcioncos". Este diagrama supone realmente una ampliación del visto en 3.3, pero se ha decidido separarlo para aportar más claridad.

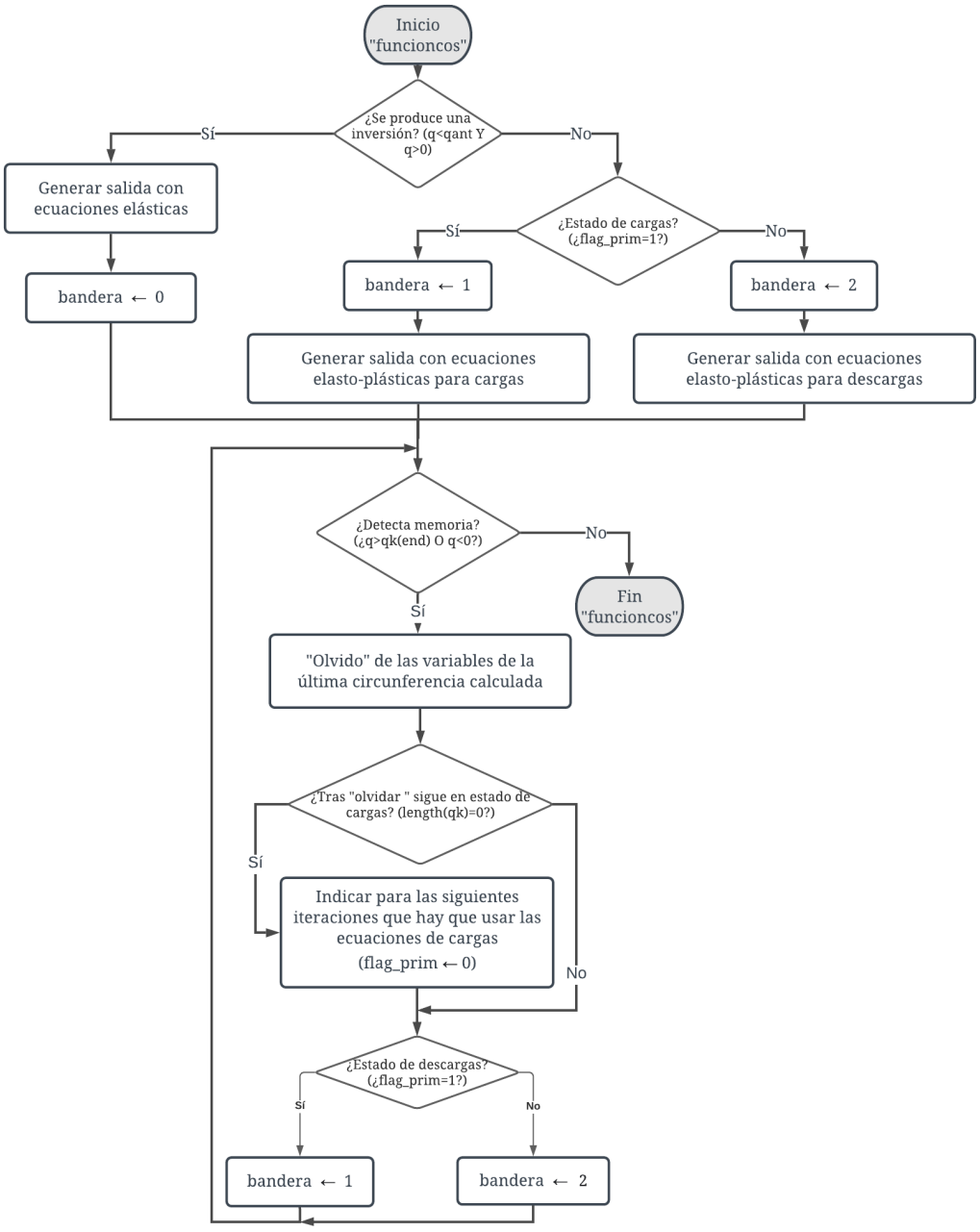

Figura 3.6 Diagrama de flujo de la modificación de la propiedad bandera.

Una vez que en la primera etapa se ha definido el nuevo valor de "bandera" para esta iteración, al ejecutar "funcioncosaux" en el resto de etapas se obtendrán salidas consecuentes a lo esperado. Esto se realiza de manera sencilla con varios comandos "if" consecutivos que dependen del valor de esta variable. Según sea el caso, "funcioncosaux" realizará una llamada a las funciones auxiliares correspondientes que se explicaron en 3.2.

- **"bandera"=0**: Salida con ecuaciones elásticas.
- **"bandera"=1**: Salida con ecuaciones elasto-plásticas para cargas.
- **"bandera"=2**: Salida con ecuaciones elasto-plásticas para descargas.

Código 3.19 Método funcioncosaux.

```
function sal=funcioncosaux(obj, t, y) % función auxiliar para que no se
    produzcan datos basura en la integración

        sigma_i=y(1);tau_i=y(2);

        Young=datmat.Young;
        G=datmat.G;

% Rescatar datos para los cálculos de q %

        ck_i=obj.ck; % mx2
        tensk_i=obj.tensk; % mx2
        qk_i=obj.qk; % mx1

% Calcula derivadas en el instante t

        DerDef(1, 1)=ppval(obj.ppdepsilon, t);
        DerDef(2, 1)=ppval(obj.ppdgamma, t);% 2x1

%%% SEGÚN EL CASO SE UTILIZAN DIFERENTES ECUACIONES PARA LA SALIDA %%%

% Esta bandera se actualiza en funcioncos e indica las ecuaciones
% que se deben usar en la auxiliar
        bandera_i=obj.bandera;

% CASO 1: ECUACIONES ELÁSTICAS
        if bandera_i==0
            sal=[Young, 0; 0, G]*DerDef;
        end

% CASO 2: ECUACIONES ELASTO-PLÁSTICAS PARA CARGAS
        if bandera_i==1
            q_i=CalculaqCarga(sigma_i, tau_i);
            A=GeneraAQ(sigma_i, tau_i, q_i/2);
            sal=A*DerDef; % 2x1
        end

% CASO 3: ECUACIONES ELASTO-PLÁSTICAS PARA DESCARGAS
        if bandera_i==2
            q_i=CalculaqDescos(sigma_i, tau_i, tensk_i(end, :), ck_i(end,
                :));
            A=GeneraADes(sigma_i, tau_i, q_i, tensk_i(end, :), ck_i(end,
                :), qk_i(end));
            sal=A*DerDef; % 2x1
        end
        end
    end
```

3.5 Implementación de un método de integración adaptativo

El hecho de haber incluido la Programación Orientada a Objetos anteriormente, facilita la evolución a un algoritmo de paso variable. Solo es necesario añadir un nuevo método a la clase punto (que para este código cambia su nombre a "puntoadapt") y modificar el bucle de integración del "mainPOO". Este método adicional se encarga de restablecer la configuración de los datos de memoria si se quiere usar un nuevo incremento de tiempo menor.

Para ello, hay que utilizar los conceptos matemáticos explicados en la sección 2.6 y modificar el bucle de integración para calcular el valor EST_n en cada paso. Si este valor es mayor que la tolerancia exigida, simplemente se resetea el objeto y se prueba con un paso menor que es posible estimar (línea 38 del Código 3.20).

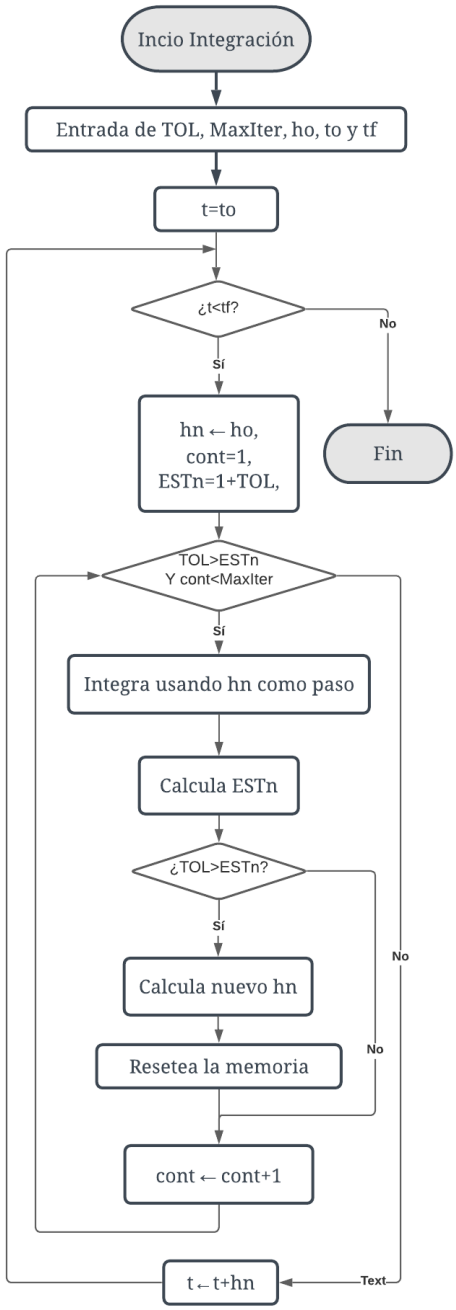

Figura 3.7 Diagrama de flujo de un método de integración con paso variable.

De esta manera, se puede esperar de este nuevo código una precisión parecida a la conseguida con el comando ODE45. Adicionalmente, el usuario del programa ahora tiene acceso al control de la tolerancia deseada y al número máximo de iteraciones que se pueden llevar a cabo en cada paso de integración (líneas 5-12 del Código 3.20). Todas estas características sirven para optimizar el coste de procesamiento del programa en su conjunto y para mejorar el cálculo numérico.

Código 3.20 Cambios en el bucle de resolución adaptativo.

```
1 %%% INTEGRACIÓN %%%
2
```

```
% Parámetros de integración

y(:,1)=tens0; % Condición inicial
yprim(:, 1)=tens0;
h0=(tdef(2)-tdef(1))/2; % Paso de integración inicial
hmin=0.00001; % Mínimo paso de h
t0=tdef(i);
tf=tdef(end);
TOL=1e-6; % Tolerancia
MaxIter=3; % Número máximo de iteraciones en cada paso

...

% Bucle de integración adaptativo

while (t0<tf)

    hn=h0; % Empezar con el paso más grande
    cont=1; % Contador de iteraciones a cero
    ESTn=TOL+1; % Para que entre por primera vez en el bucle

        while ESTn>TOL && cont<=MaxIter

            h=hn; % Paso en esta iteración
            k1=punto1.funcioncos(t0, y(:,end)); % Aplica memoria
% Aquí no hay que aplicar memoria porque guarda valores basura
            k2=punto1.funcioncosaux(t0+h*c(1), y(:,end)+h*k1*A(1,1));
            k3=punto1.funcioncosaux(t0+h*c(2), y(:,end)+h*([k1 k2]*A(2,
                1:2)'));
            k4=punto1.funcioncosaux(t0+h*c(3), y(:,end)+h*([k1 k2 k3]*A
                (3, 1:3)'));
            k5=punto1.funcioncosaux(t0+h*c(4), y(:,end)+h*([k1 k2 k3 k4]*
                A(4, 1:4)'));
            k6=punto1.funcioncosaux(t0+h*c(5), y(:,end)+h*([k1 k2 k3 k4
                k5]*A(5, 1:5)'));
            k7=punto1.funcioncosaux(t0+h*c(5), y(:,end)+h*([k1 k2 k3 k4
                k5 k6]*A(6, 1:6)'));
            yn=y(:,end)+h*[k1 k2 k3 k4 k5 k6 k7]*b'; % R-K p=5
            yprim=y(:,end)+h*[k1 k2 k3 k4 k5 k6 k7]*bprim'; % R-K p=4

            ESTn=norm(yprim-yn); % Error estimado

            if ESTn>TOL && cont<MaxIter % Comparar tolerancia con ESTn
            punto1.reset % Resetear objeto para la nueva iteración
            hn=h*(TOL/ESTn)^(1/5); % Estimar nuevo paso de integración
            if hn<hmin % Evita que haya un h<hmin
                hn=hmin;
                cont=MaxIter; % Salir del bucle si se alcanza hmin
            end
            end
```

```
48
49              cont=cont+1; % Contador de iteraciones
50
51          end
52
53      t0=t0+h; % Nuevo t0 para el siguiente punto de precisón
54      y(:, end+1)=yn;
55
56  end
```

3.5.1 Modificación de la clase Punto a Puntoadapt

Como se adelantaba, es necesario diseñar una evolución de la clase punto para generar objetos que puedan realizar los cálculos paso a paso. Las propiedades y los métodos descritos en el apartado 3.3 no sufren modificaciones y guardan la misma información y se añaden a las que se enumeran a continuación:

- **"qk_aux"** ((b-1)x1): Al principio del bucle de integración en "funcioncos" se crea una copia del valor actual de la propiedad "qk". En caso de tener que resetear el objeto, se volverán a cargar estos valores guardados en "qk_aux", reiniciando así el algoritmo y preparándolo para un paso de integración más pequeño.
- **"tensk_aux"** (bx1): Funciona como "qk_aux", pero con los valores de las tensiones de referencia.
- **"ck_aux"** (bx1): Igual que las dos anteriores, pero con los valores de los centros de las circunferencias.

Código 3.21 Propiedades adicionales de la clase puntoadapt.

```
1     properties
2         .
3         .
4         .
5         % Variables auxiliares para poder tener ODE adaptativo
6
7         qk_aux
8         ck_aux
9         tensk_aux
10    end
```

Código 3.22 Cambios en funcioncos para paso variable.

```
1  function sal=funcioncos(obj, t, y)
2
3  ...
4
5  % Datos de q de iteraciones anteriores. Se puede cambiar directamente
6  % pero se va a trabajar con variables locales _i por ser más cómodas)
7         qant=obj.q(end);
```

```
8         tensant(1)=obj.sigma(end); % 1x2
9         tensant(2)=obj.tau(end); % 1x2
10        ck_i=obj.ck; % mx2
11        tensk_i=obj.tensk; % mx2
12        qk_i=obj.qk; % mx1
13        prim_i=obj.prim(end);
14
15 % Variables auxiliares para resetear los datos en el caso de tener que
16 % diminuir el paso de integración
17        obj.qk_aux=qk_i;
18        obj.tensk_aux=tensk_i;
19        obj.ck_aux=ck_i;
20
21 ...
22
23        end
```

El último de los cambios en la clase es la adición de un método (función "reset") que se encargue de eliminar tanto los datos como la memoria de la última iteración. En primer lugar (líneas 10-21), se borran los datos de interpolación de tensiones solo si los hay. Seguidamente, se borran los últimos registros de cada uno de los historiales y se renuevan los datos de memoria (24-35). Por último, se vuelven a ajustar las banderas de tolerancia (45-52) si se está utilizando un valor de la misma mayor que cero. Tras realizar estas operaciones, se aprecia claramente que la información que se encuentra dentro del objeto "punto1" es exactamente la misma que antes de la llamada a "funcioncos".

Código 3.23 Método Reset.

```
1         function reset(obj)
2 % En el caso en el que se requiera un paso de tiempo más pequeño, esta
3 % función elimina los datos de la iteración anterior y carga los que se
4 % usaron en un principio. En resumen, resetea el objeto punto para
5 % volver a iterar en el mismo punto pero con un h menor
6
7 % Borrar datos de retracción (en el caso en el que se haya llamado a
8 % BuscatensRet hay que borrar también esos parámetros)
9
10            if length(obj.mem)>1
11            while obj.mem(end-1)==2
12                obj.sigma(end-1)=[];
13                obj.tau(end-1)=[];
14                obj.ttens(end-1)=[];
15                obj.q(end-1)=[];
16                obj.ck_hist(end-1, :)=[];
17                obj.tensk_hist(end-1, :)=[];
18                obj.mem(end-1)=[];
19                obj.reg(end-1)=[];
20                obj.prim(end-1)=[];
21            end
22            end
23
```

```
% Borrar el resto de los datos (borra todos los últimos datos
% guardados en las variables de memoria)

            obj.sigma(end)=[];
            obj.tau(end)=[];
            obj.ttens(end)=[];
            obj.q(end)=[];
            obj.ck_hist(end, :)=[];
            obj.tensk_hist(end, :)=[];
            obj.mem(end)=[];
            obj.reg(end)=[];
            obj.prim(end)=[];

% Vuelve a poner los valores de memoria guardados al principio del bucle

            obj.ck=obj.ck_aux;
            obj.tensk=obj.tensk_aux;
            obj.qk=obj.qk_aux;

% Valores para tolerancia de inversiones (comentar si no se está usando)

%             if isempty(obj.q)==0
%             if obj.q(end)<obj.qregaux && obj.q(end)+obj.TOL>obj.qregaux
%                 obj.regaux=1; % Si esta en el intervalo se pone regaux
%             end
%
%             if obj.q(end)==obj.qregaux
%                 obj.qregaux=-1;
%             end
%             end
        end
```

3.6 Programa en variables globales

El primer enfoque que se usó para la realización de este programa fue con variables globales, pero finalmente se descartó y se prefirió trabajar con POO por las ventajas que ello conlleva. En este apartado se describen las características de este código por si algún usuario prefiere emplear variables globales. Las funciones auxiliares explicadas en 3.2 no sufren ninguna modificación y realizan las mismas tareas que ya se han descrito.

En primer lugar, se describen las variables globales usadas en este programa y cuál es la finalidad de los datos que se guardan en cada una de ellas, comparándolas con las propiedades de la clase "punto" explicada en el apartado 3.4.1.

- **"DerDef"**: Estructura de datos con dos splines de las derivadas de las deformaciones. Es equivalente a las estructuras guardadas en "punto.ppdepsilon" y "punto.ppdgamma" y también pueden evaluarse con "ppval" en cualquier instante de tiempo.
- **"dat_q"**: Variable en la que se guardan los datos de memoria del punto para poder comparar en cada iteración los valores obtenidos. Es una estructura en la que se encuentran los vectores de memoria "punto.qk", "punto.tensk" y "punto.ck".

- **"punto"**: Variable en la que se incluye toda la información referente al punto estudiado. Se trata de una estructura de datos que cuenta con los siguientes vectores de valores para cada uno de los puntos de integración: vector de tiempos de integración, evolución de tensiones, evolución de la distancia efectiva q, centros de las circunferencias sucesivas c_k, puntos de tensiones de referencias anteriores σ_k y vectores de banderas.

3.6.1 Script main.m para variables globales y bucle de integración

No hay excesivas diferencias en el archivo principal con respecto a la POO, ya que las tareas que realiza son exactamente las mismas. Difiere en que en este caso es necesario el uso de las funciones auxiliares extra del apartado 3.6.2 para guardar la información. Para no alargar innecesariamente el documento, en el Código 3.24 se han omitido los fragmentos que son iguales a los mostrados en el caso de POO y solo se muestran aquellas líneas en las que hay diferencias sustanciales.

Código 3.24 Modificaciones del archivo principal para el uso de varibales globales.

```
clc, clear, close all

%%% INTRODUCIR NOMBRE DEL ARCHIVO DE DATOS %%%

nombre='def_A';
nombretxt=[nombre, '.txt'];

...

%%% CÁLCULO DE DERIVADAS %%%

% Pedir vectores de derivadas según fórmula de 5 puntos
[depsilon, dgamma]=CalculaDerivada(def_entrada, deltaTdef); %[nvalx1]

% Crear splines para poder ser evaluadas en cualquier t
ppdepsilon=spline(tdef, depsilon);
ppdgamma=spline(tdef, dgamma);

% Establecer y'(t) como función con getDert(t)
setDer(ppdepsilon, ppdgamma);

...

% Iniciar valores para el cálculo de q sucesivos

tant=0; % t0=0
ck0=[0, 0]; % El primer centro siempre va a ser cero
qk0=[]; % Por ahora no hay registros de qk anteriores
qant=CalculaqCarga(tens0(1), tens0(2)); % Ec para cargas
tensant=tens0;
tensk0=[0, 0]; % Se supone [0, 0]
prim=1;% Empieza con ec 1

setDatq(tant, tensant, qant, tensk0, ck0, qk0, prim);

```

```
36 %%% ESTRUCTURA PUNTO %%% (crear la estructura con todos los datos)
37
38 % Inicio de la estructura punto, donde se guardan todos los datos
39 % y se actualizan gracias a que en funcioncos se encuentra la
40 % función sethistorico
41
42 global punto
43 punto=struct('tdef', tdef, 'def', def_entrada, 'ddef', [ppdepsilon,
       ppdgamma], 'ttens', [], 'tens', [], 'tensk', [], 'ck', [], 'q', [], '
       reg', [], 'mem', [], 'prim', []);
44
45 ...
46
47 %%% TRATAMIENTO DE DATOS DE SALIDA %%%
48
49 muestracos=[punto.ttens', punto.tens(:, 1), punto.tens(:, 2), punto.q',
       punto.tensk, punto.ck, punto.reg', punto.mem', punto.prim'];
```

Respecto al bucle de memoria, como ya no hay ninguna clase en la que programar un método, las funciones "funcioncos" y "funcioncosaux" son independientes. Su llamada se realiza de manera análoga a la explicada anteriormente para calcular las etapas del método de DOPRI5(4). Los códigos de "funcioncos.m" y de "funcioncosaux.m" no se añaden para no extender en exceso el documento, ya que son sumamente parecidos a los métodos de los apartados 3.4.3 y 3.4.4.

La propiedad que funcionaba como variable auxiliar llamada "bandera" se ha sustituido por una variable global tipo "int" (1x1) y mantiene la misma función que tenía anteriormente.

El programa en variables globales no cuenta con un sistema de resolución de paso variable porque su programación en POO resulta mucho más sencilla. Esta solución solo se adjunta para ilustrar cómo es posible llegar al mismo resultado sin necesidad de usar objetos.

3.6.2 Funciones auxiliares para variables globales

En este apartado se exponen las funciones auxiliares extra que se emplean para poder actualizar u obtener datos de las estructuras globales. En lugar de declarar las estructuras directamente en el código, la inclusión de estas funciones ayuda a tener una programación más ordenada.

Funciones setDer y getDer

Con "setDer" se guardan las estructuras de salida de la función spline (que se deben dar como argumento) en una variable global con el objetivo de evaluar posteriormente dichos splines en cualquier instante de tiempo.

Código 3.25 Función setDer.

```
1 function setDer(ppdepsilon, ppdgamma)
2 % Guarda las derivadas en las variables globales
3
4 global DerDef
5
6 DerDef.ppdepsilon=ppdepsilon;
7 DerDef.ppdgamma=ppdgamma;
8 end
```

La función "getDer" devuelve un vector con los valores de las derivadas de las deformaciones en el instante de tiempo que se manda como argumento. Para ello, esta función hace uso de los splines anteriores y los evalúa con el comando interno de MATLAB "ppval".

Código 3.26 Función getDer.

```
function [der_t]=getDer(t)
% Esta función se encarga de devolver el valor de las derivadas de las
% deformaciones en un instante de tiempo determinado (se usa en el bucle
% de integración en funcioncos y funcionck)

global DerDef

ppdepsilon=DerDef.ppdepsilon;
ppdgamma=DerDef.ppdgamma;

der_t=[ppval(ppdepsilon, t); ppval(ppdgamma, t)];
end
```

Funciones getDatq y setDatq

La función "setDaqt" carga en la variable global "dat_q" los valores que recibe como argumento, machacando los posibles datos guardados con anterioridad. Normalmente, esta variable global se actualiza en cada paso de tiempo a lo largo del bucle de integración con nueva información de memoria de cargas. Con "getDaq" se recibe como salida los valores guardados en "dat_q".

Código 3.27 Función setDatq.

```
function setDatq(tant, tensant, qant, tensk, ck, qk, prim)
% Guarda los nuevos valores en la variable global dat_q

global dat_q;

dat_q=struct('tant', tant, 'tensant', tensant, 'qant', qant, 'tensk',
    tensk, 'ck', ck, 'qk', qk, 'prim', prim);
end
```

Código 3.28 Función getDatq.

```
function salida=getDatq()
% Devuelve como salida la variable global dat_q

global dat_q

salida=dat_q;
end
```

Función setHistorico

Esta función se encarga de actualizar los valores que se guardan en la variable global "punto" añadiendo un nuevo valor final a cada uno de los vectores que forman los campos de la estructura.

Código 3.29 Función setHistorico.

```
function setHistorico(t, sigma, tau, tensk, ck, q, reg, mem, prim)
% Guarda valores del bucle de integración en la estructura global punto

global punto

punto.ttens(end+1)=t;
punto.tens(end+1, :)=[sigma, tau];
punto.tensk(end+1, :)=tensk;
punto.ck(end+1, :)=ck;
punto.q(end+1)=q;
punto.reg(end+1)=reg;
punto.mem(end+1)=mem;
punto.prim(end+1)=prim;

end
```

A diferencia de lo que ocurre con la función "setDatq" del apartado anterior, en este caso no se sustituye toda la variable por una nueva, si no que se va actualizando la estructura "punto" a medida que se añaden valores a los vectores.

3.7 Fichero de descargas y memoria para tensiones

El programa principal está diseñado para resolver el sistema de ecuaciones diferenciales a partir de una entrada en deformaciones. Aún así, cabe la posibilidad de que un usuario quisiera aplicar el algoritmo de cálculo de circunferencias de cargas a un registro de tensiones. La solución a este problema es mucho más sencilla, ya que solo es necesario utilizar los algoritmos previamente definidos sin resolver el PVI.

Con esa finalidad se ha creado el fichero "AlgoritmoTensiones", que genera la mismas salidas gráficas y numéricas que los "main" anteriores para un registro de tensiones. Para ello, se ha programado el Código 3.30, que trabaja con la misma clase "punto" sin realizar modificaciones en ella.

Código 3.30 AlgoritmoTensiones.

```
clear, clc, close all

%%% ESTE ALGORITMO CALCULA EL VALOR DE q, LOS CENTROS, LAS tensk DE UN
%%% REGISTRO DE TENSIONES (no integra defs, solo ejecuta el algoritmo)

%%% INTRODUCIR NOMBRE DEL ARCHIVO DE DATOS %%%

nombre='TensionesRandom';
nombretxt=[nombre, '.txt'];

%%% LECTURA DE DATOS DE ENTRADA (deformaciones) %%%

[tensiones, ttens]=Lectura(nombretxt); % [mx2, mx1]

```

```
%%% INICIAR ESTRUCTURA PUNTO %%%

% Se pasa [0,0,0] porque n hay información acerca de las deformaciones y
% solo se usa para inicializar el objeto punto1

punto1=punto(0, [0; 0]);
punto1.TOL=0; % Tol. de detección (recomendado 0 o núm. muy pequeño)

%%% CONDICIONES INICIALES %%%

% Primer punto se considera elástico (sigma=[Young, 0; 0, G]*def)

i=1;
while(sum(tensiones(i, :))==0) % Si ambas componentes son nulas
    i=i+1; % En i se guarda el valor en el que
end       % empiezan las tensiones no nulas

tens0=tensiones(i, :); % 2x1

% Iniciar valores para el cálculo de q sucesivos

q0=CalculaqCarga(tens0(1), tens0(2));
ck0=[0, 0]; % El primer centro siempre va a ser [0, 0]
tensk0=[0, 0]; % Consideramos que es [0, 0]
prim=1; % Empieza con ec de cargas

punto1.tensk=tensk0;
punto1.ck=ck0;
punto1.qk=[]; % No hay memoria de qk por ahora

% Condiciones iniciales para el bucle de integración
% (al final del bucle hay que eliminar el primer componente de cada uno
% de estos debido a que por como está programado se duplica). Es
% simplemente un formalismo que simplifica el algoritmo.

punto1.q=q0;
punto1.ttens=ttens(i);
punto1.sigma=tens0(1);
punto1.tau=tens0(2);
punto1.prim=prim;

% Se va a pasar un spline como derivadas pero es irrelevante, ya que
% funcioncos lo necesita o no funciona

aux1=[1,0];
aux2=[0,1];
pp1=spline(aux2, aux1);
pp2=spline(aux2, aux1);
punto1.ppdepsilon=pp1;
punto1.ppdgamma=pp2;
```

```
65
66 % En lugar de integrar simplemente se pasa como argumento las tensiones
67 % para aprovechar funcioncos dentro de la clase punto
68
69 for k=1:length(ttens)
70     nada=punto1.funcioncos(ttens(k), tensiones(k,:)');
71 end
72
73 % Por como está programado, hay que eliminar estos valores para que
74 % los vectores tengan la longitud real y no haya contenido duplicado
75 % (se corresponden con los primeros valores del bucle del PVI)
76
77 punto1.ttens(1)=[];
78 punto1.sigma(1)=[];
79 punto1.tau(1)=[];
80 punto1.q(1)=[];
81 punto1.prim(1)=[];
82
83 % Se van a añadir los puntos de tensión en el caso en el que i no sea
84 % cero y por tanto haya un instante de tiempo en el que todo es nulo
85
86 if i>1
87
88 punto1.ttens=[tdef(1:i-1); punto1.ttens];
89
90 ceros1=zeros(i-1, 1);
91 ceros2=zeros(i-1, 2);
92
93 punto1.q=[ceros1; punto1.q];
94 punto1.sigma=[ceros1; punto1.sigma];
95 punto1.tau=[ceros1; punto1.tau];
96 punto1.ck_hist=[ceros2; punto1.ck_hist];
97 punto1.tensk_hist=[ceros2; punto1.tensk_hist];
98 punto1.mem=[ceros1; punto1.mem];
99 punto1.reg=[ceros1; punto1.reg];
100 punto1.prim=[ceros1; punto1.prim];
101
102 end
103
104 % Se va a calcular la última circunferencia fuera del bucle (solo hay
105 % que calcular el último centro y poner reg=1 para indicar que se ha
106 % hecho la última de las circunferencias)
107
108 punto1.reg(end)=1;
109
110 if punto1.prim(end)==0
111     normaaux=sqrt((2/3)*(punto1.ck_hist(end-1, 1)-punto1.tensk_hist(end
        -1, 1))^2+2*(punto1.ck_hist(end-1, 2)-punto1.tensk_hist(end-1, 2)
        )^2);
```

```
112     punto1.ck_hist(end,:)=punto1.tensk_hist(end,:)+(punto1.ck_hist(end
            -1,:)-punto1.tensk_hist(end-1,:))/normaaux*(punto1.q(end)/2);
113 end
114 if punto1.prim(end)==1
115     punto1.ck_hist(end, :)=[0, 0];
116 end
117
118 %%% GRÁFICAS %%%
119 dibujar(punto1.sigma, punto1.tau, punto1.tensk_hist, punto1.ck_hist,
        punto1.q, punto1.ttens, punto1.reg, punto1.mem, nombre)
120
121 % Guardar en formato .jpeg (comentar si no se quieren guardar)
122 %
123 % nombresalida1=['Resultado (', nombre, ').jpg'];
124 % saveas(1, nombresalida1)
125 %
126 % nombresalida2=['Tensiones (', nombre, ').jpg'];
127 % saveas(2, nombresalida2)
128
129 %%% CONSTRUCCIÓN DE MATRIZ DE ANÁLISIS %%%
130 muestra=[punto1.ttens, punto1.sigma, punto1.tau, punto1.q, punto1.
        tensk_hist, punto1.ck_hist, punto1.reg, punto1.mem, punto1.prim];
131
132 % filename = 'Datos_punto.xlsx';
133 % writematrix(muestra,filename,'Sheet',1,'Range','A2')
```

La explicación del apartado 3.3 facilita la compresión de este fichero. En primer lugar, se escribe el nombre del archivo exactamente igual que con las deformaciones, pero sustituyendo la columna de datos de ε por σ y la de γ por τ. Después de eso, se instancia la clase punto con valores nulos de deformaciones y se pasan splines genéricos para que el método "funcioncos" no produzca errores (20 y 56-64). Estos datos no influyen en el resultado y solamente se guardan para que no ocurran fallos de compilación. En las líneas 69-71 se van calculando y guardando los valores de q y las circunferencias de cargas. Al final del programa se generan las mismas salidas gráficas y la tabla "muestra" como en el caso de deformaciones.

4 Demostración de resultados

Una vez que se ha presentado el programa y se ha desglosado el funcionamiento de cada una de sus partes, se va a proceder a realizar una serie de pruebas. Para ello, se emplean registros de deformaciones generados manualmente que representan las distintas trayectorias expuestas en el apartado 1.4.3. Con el fin de obtener estos registros, se ha programado un fichero de MATLAB que automáticamente genera 15 archivos .txt. Estos datos se muestran en la figura 4.1. También se adjunta la evolución de ε y γ a lo largo del tiempo en la gráfica 4.2.

Todos estos valores son ondas senoidales, con una frecuencia de 1 Hz y que se estudiarán a lo largo de un segundo, es decir, un ciclo. La amplitud máxima de la primera componente del vector de deformaciones será 0.005 y la de la segunda componente, 0.0035. Esto se ha decidido así para que el orden de magnitud de las tensiones obtenidas sea representativo de un caso real. Los desfases entre ambas componentes se han ajustado para que las trayectorias obtenidas sean las mismas que las mostradas en la figura 1.6.

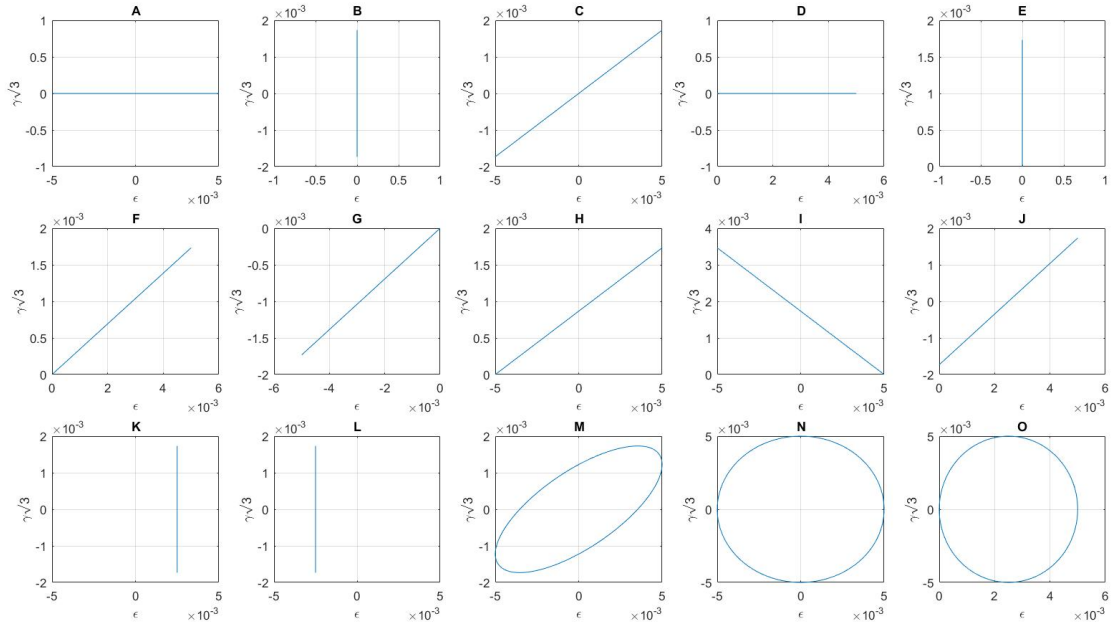

Figura 4.1 Gráficas $\varepsilon - \gamma$ para entradas de deformaciones.

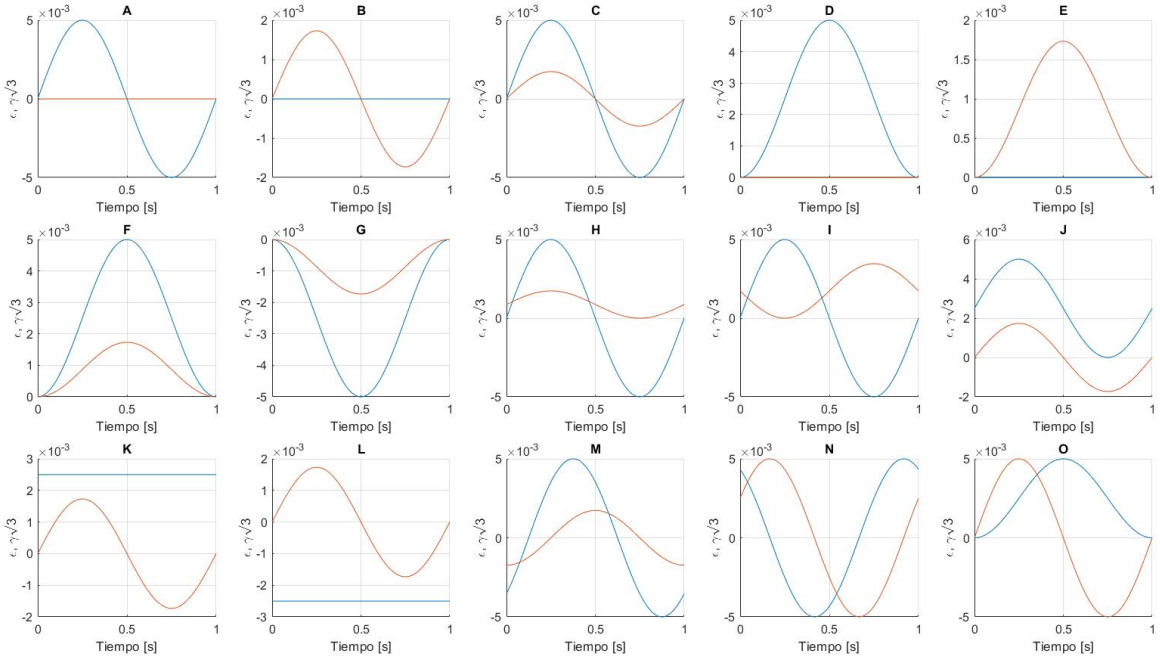

Figura 4.2 Gráficas $\varepsilon - t$ y $\gamma - t$ para entradas de deformaciones.

4.1 Representación gráficas de soluciones obtenidas

Para cada entrada de deformaciones se presentan las dos gráficas que genera la función "dibujar". La primera de ellas con las circunferencias de cargas y la evolución de q a lo largo del ciclo y la segunda con las componentes del vector de tensiones en el intervalo de tiempo.

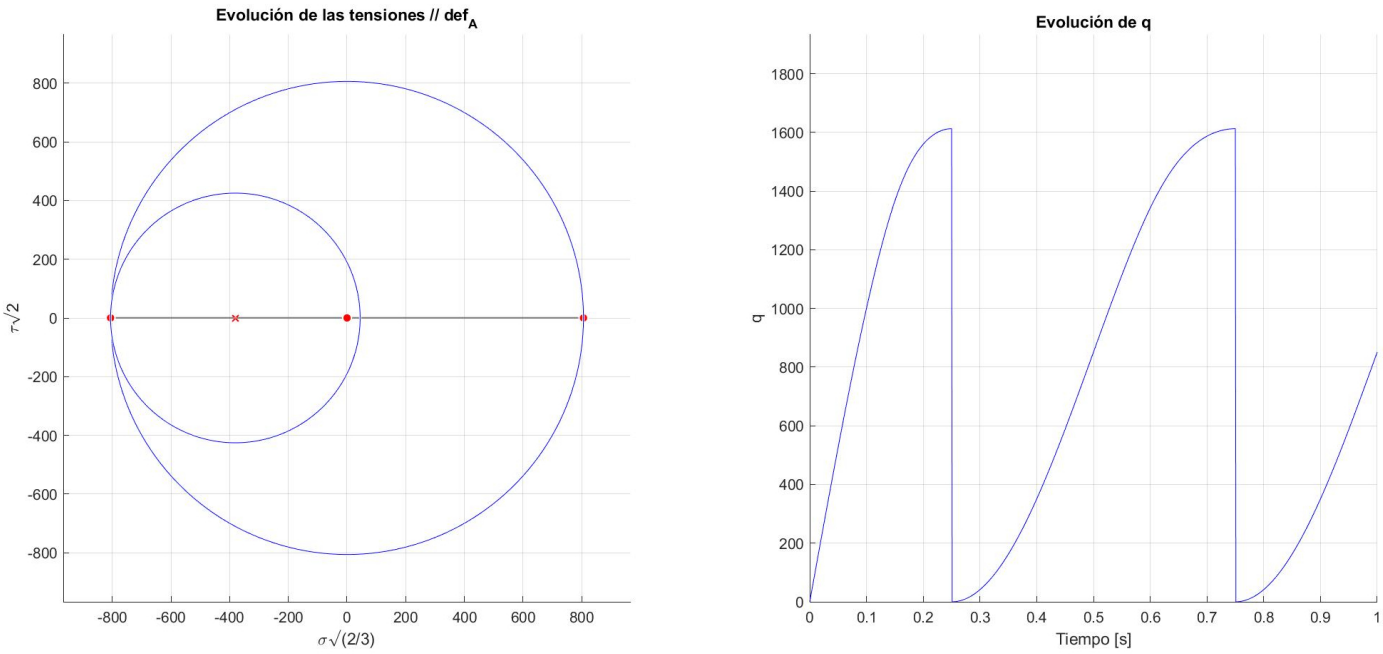

Figura 4.3 Resultado gráfico para el caso A.

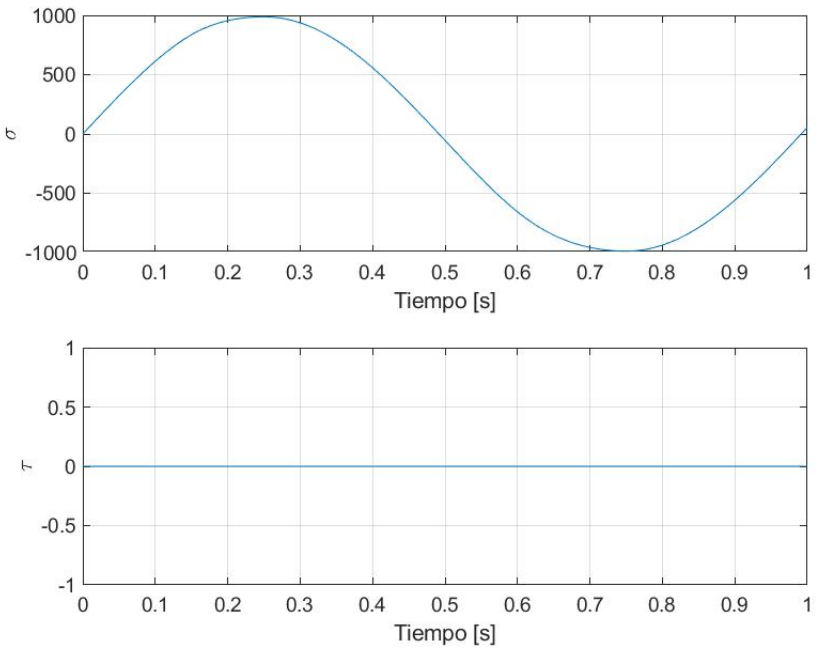

Figura 4.4 Evolución de las tensiones de salida para el caso A.

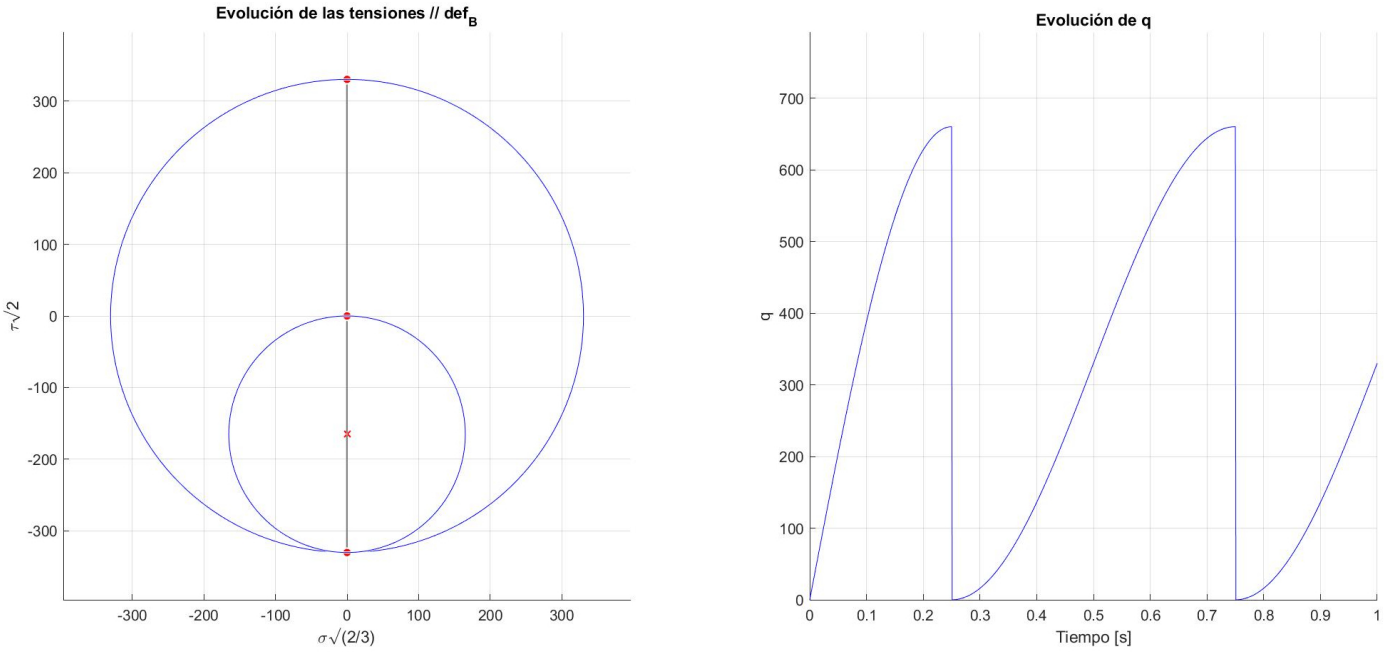

Figura 4.5 Resultado gráfico para el caso B.

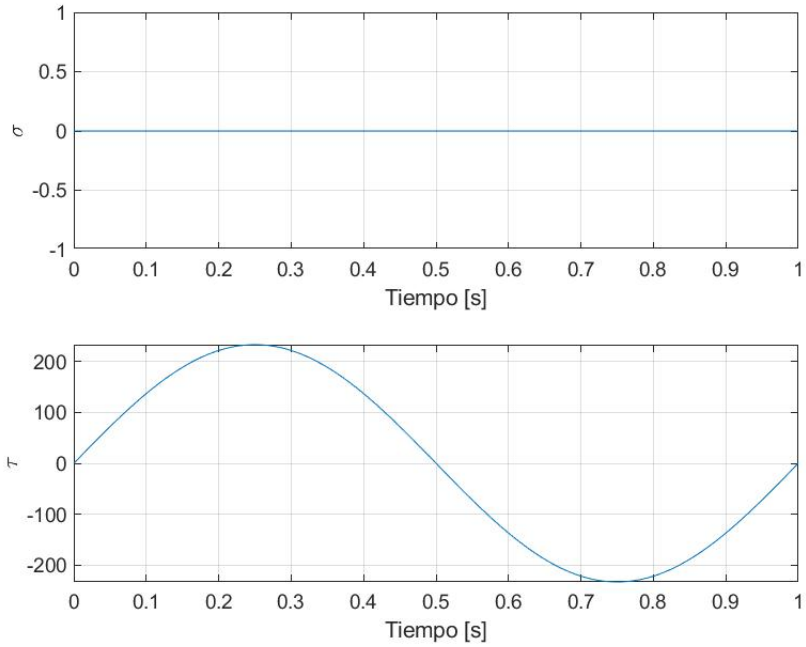

Figura 4.6 Evolución de las tensiones de salida para el caso B.

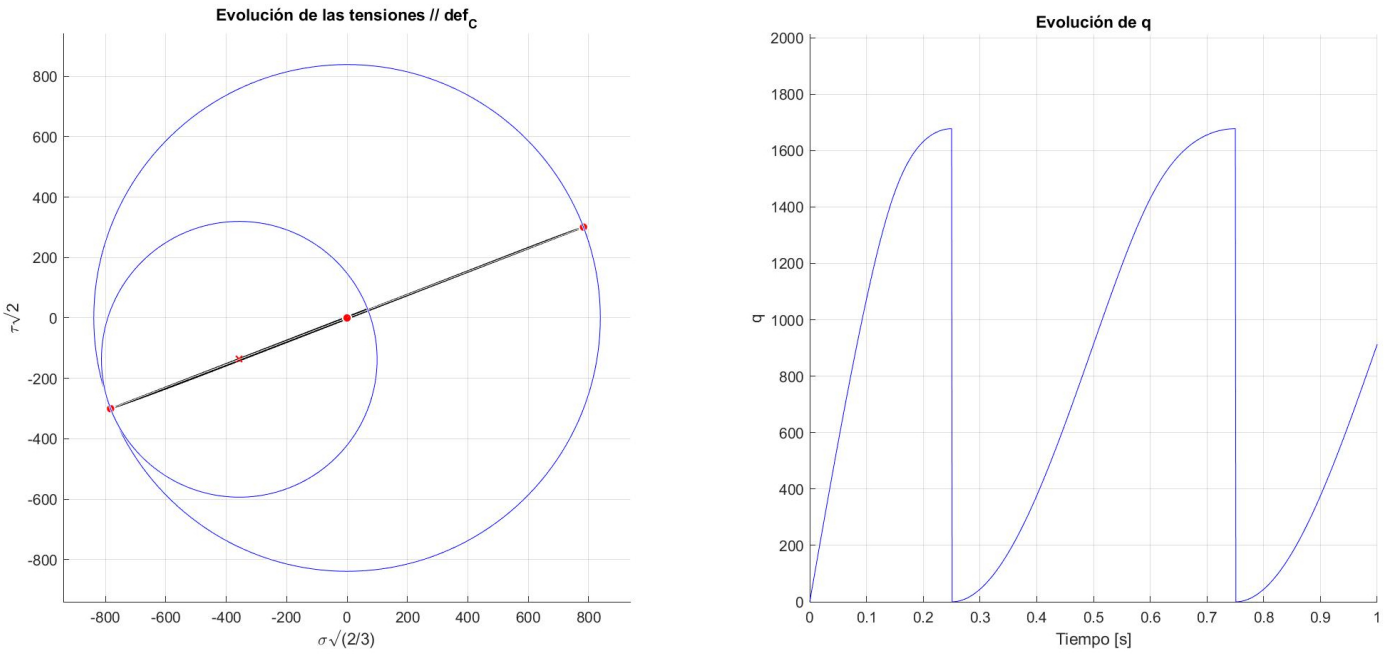

Figura 4.7 Resultado gráfico para el caso C.

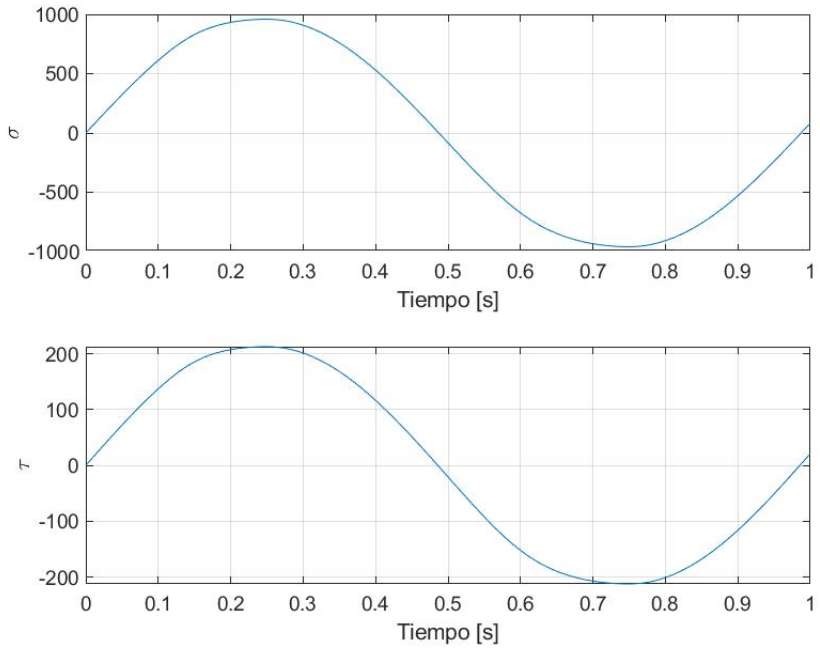

Figura 4.8 Evolución de las tensiones de salida para el caso C.

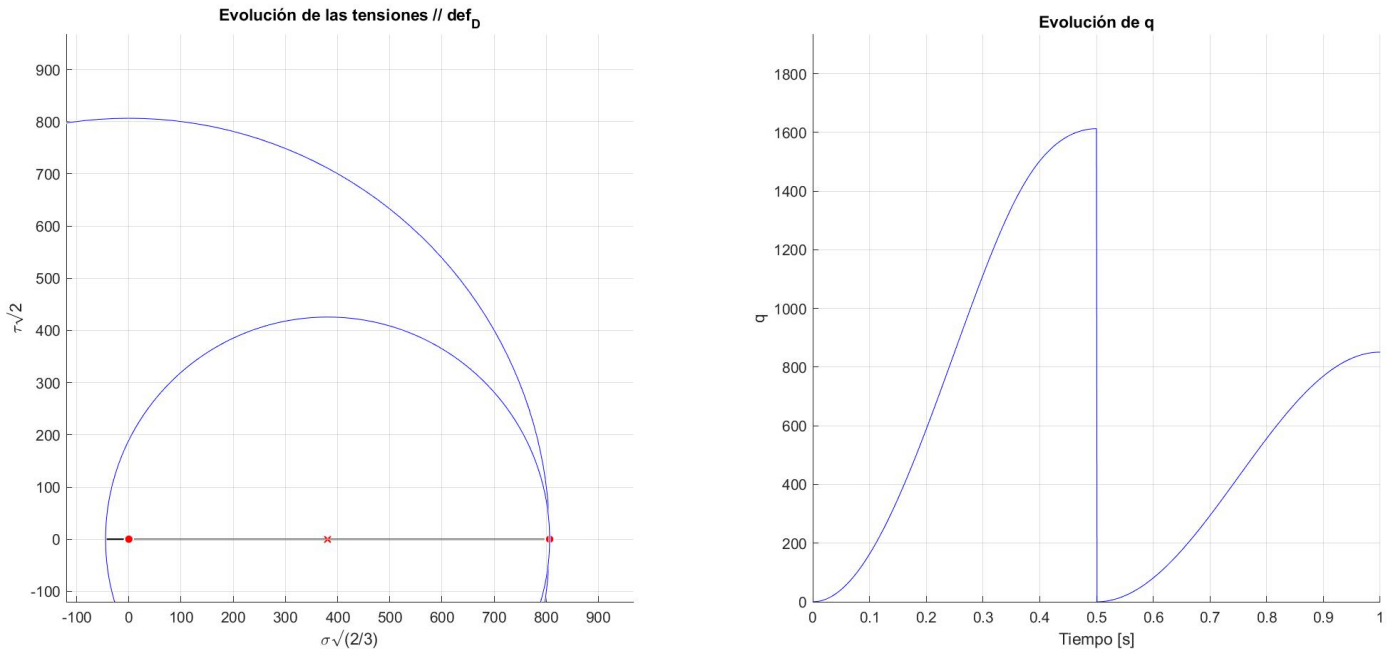

Figura 4.9 Resultado gráfico para el caso D.

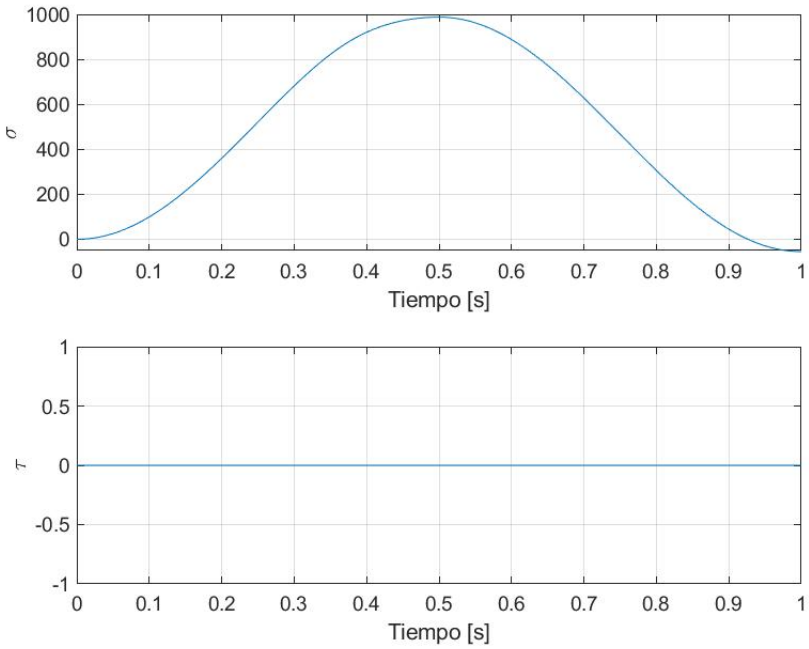

Figura 4.10 Evolución de las tensiones de salida para el caso D.

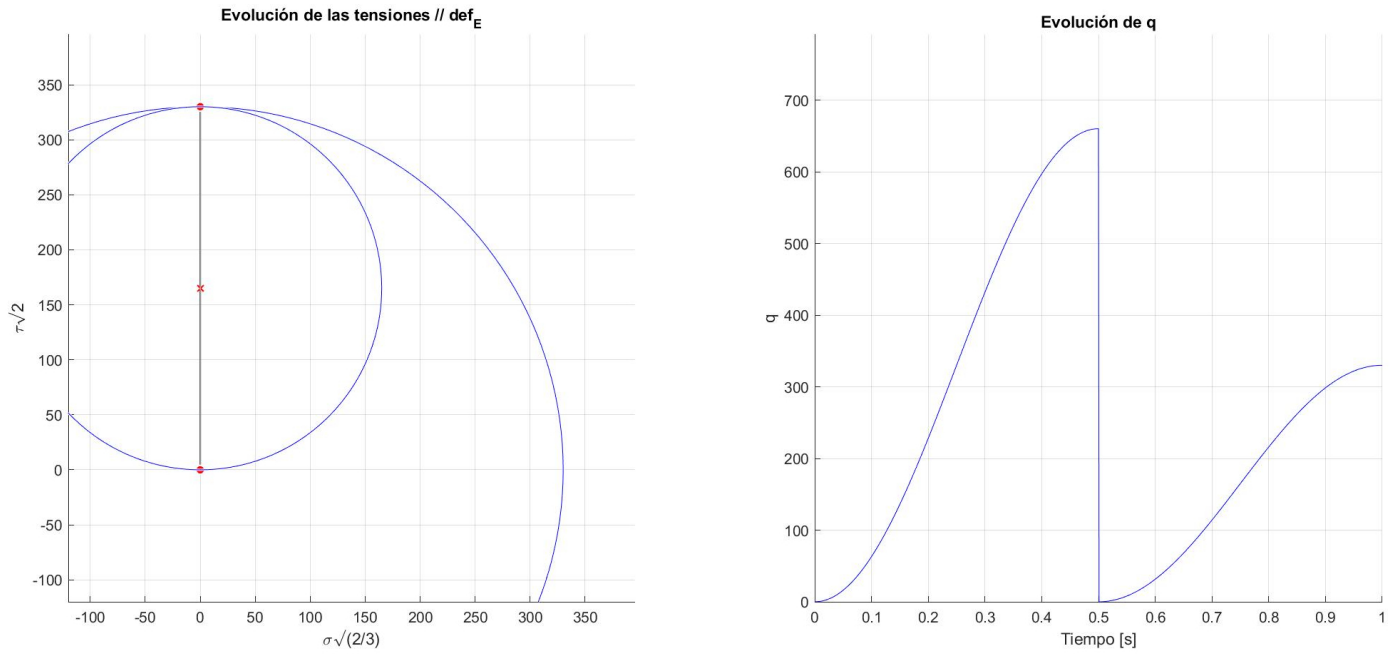

Figura 4.11 Resultado gráfico para el caso E.

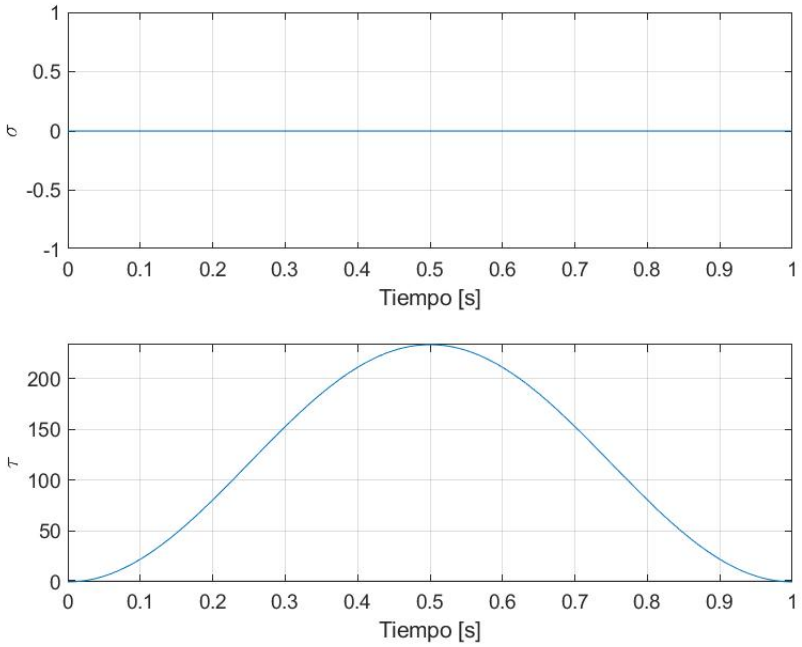

Figura 4.12 Evolución de las tensiones de salida para el caso E.

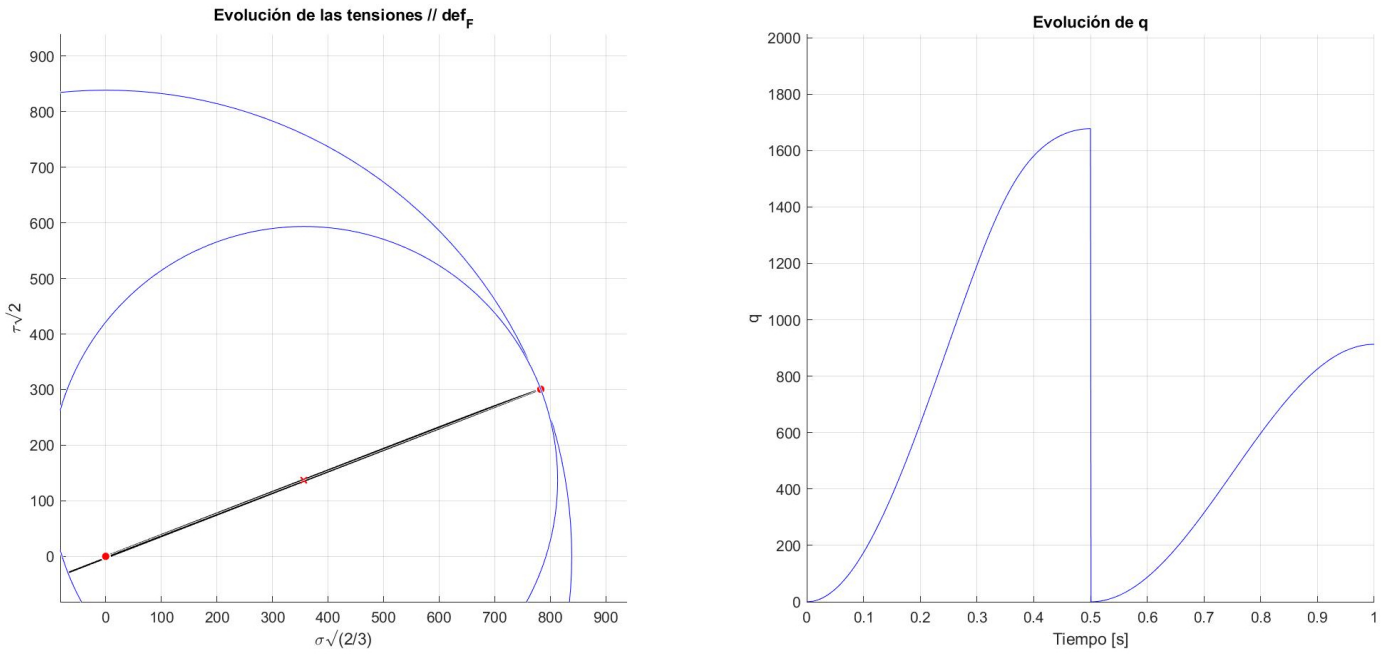

Figura 4.13 Resultado gráfico para el caso F.

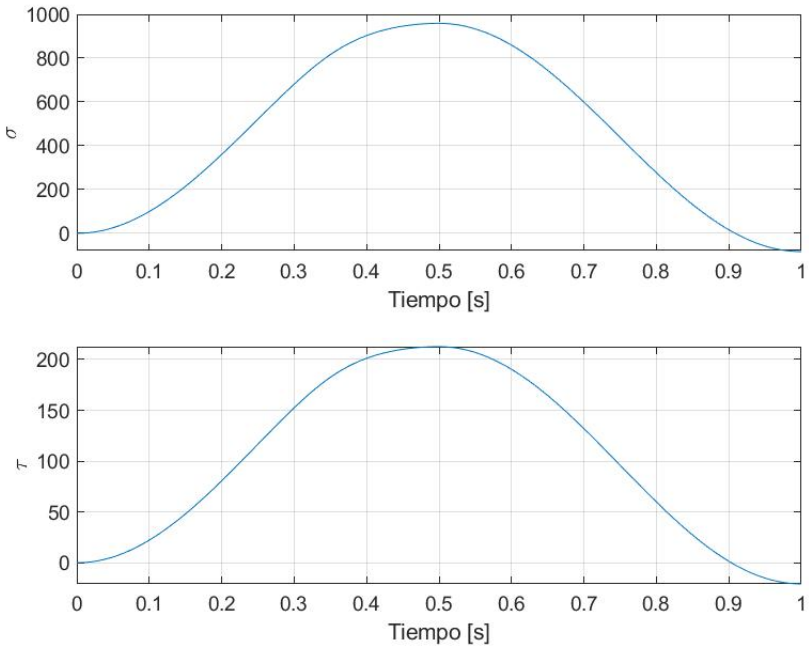

Figura 4.14 Evolución de las tensiones de salida para el caso F.

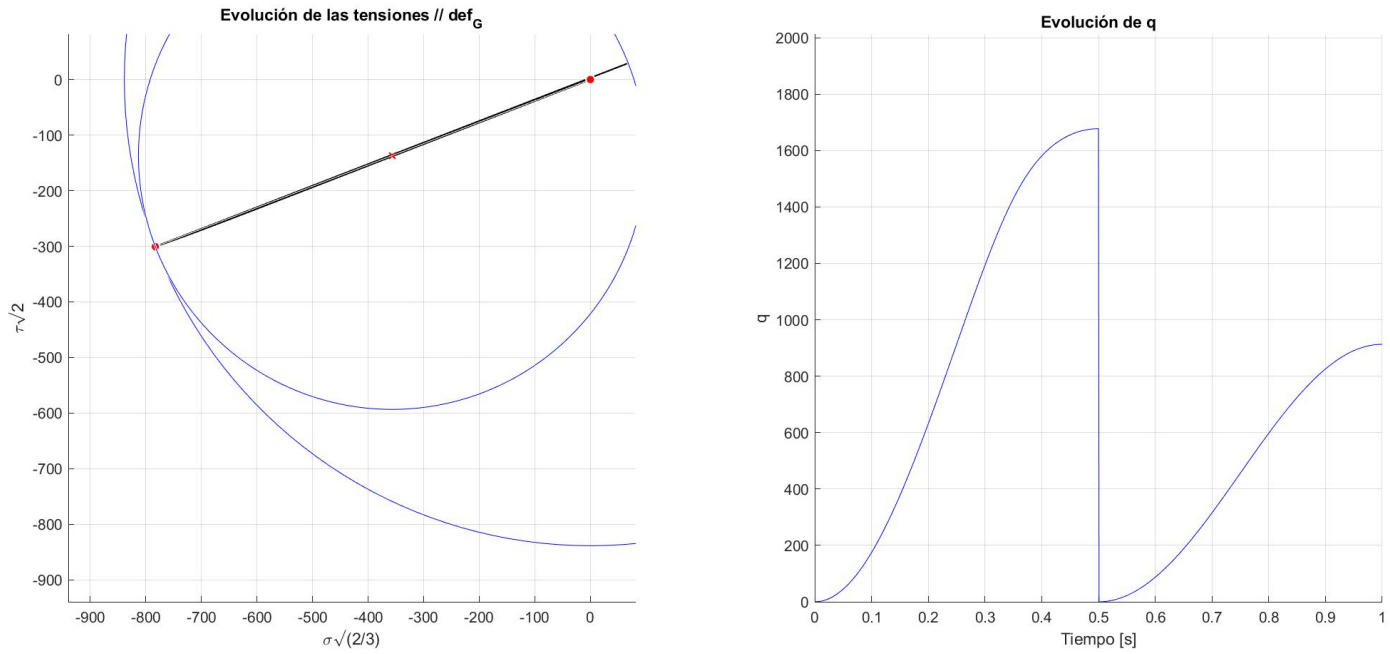

Figura 4.15 Resultado gráfico para el caso G.

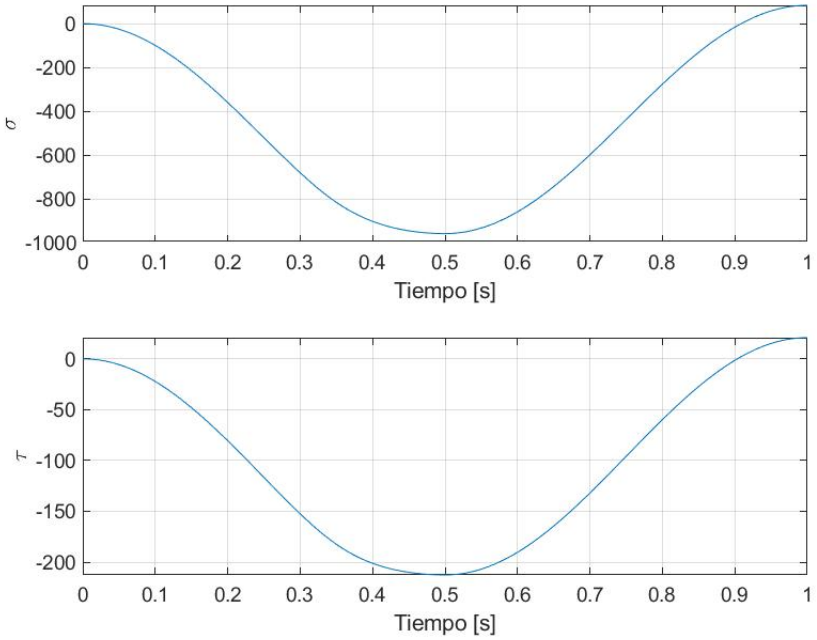

Figura 4.16 Evolución de las tensiones de salida para el caso G.

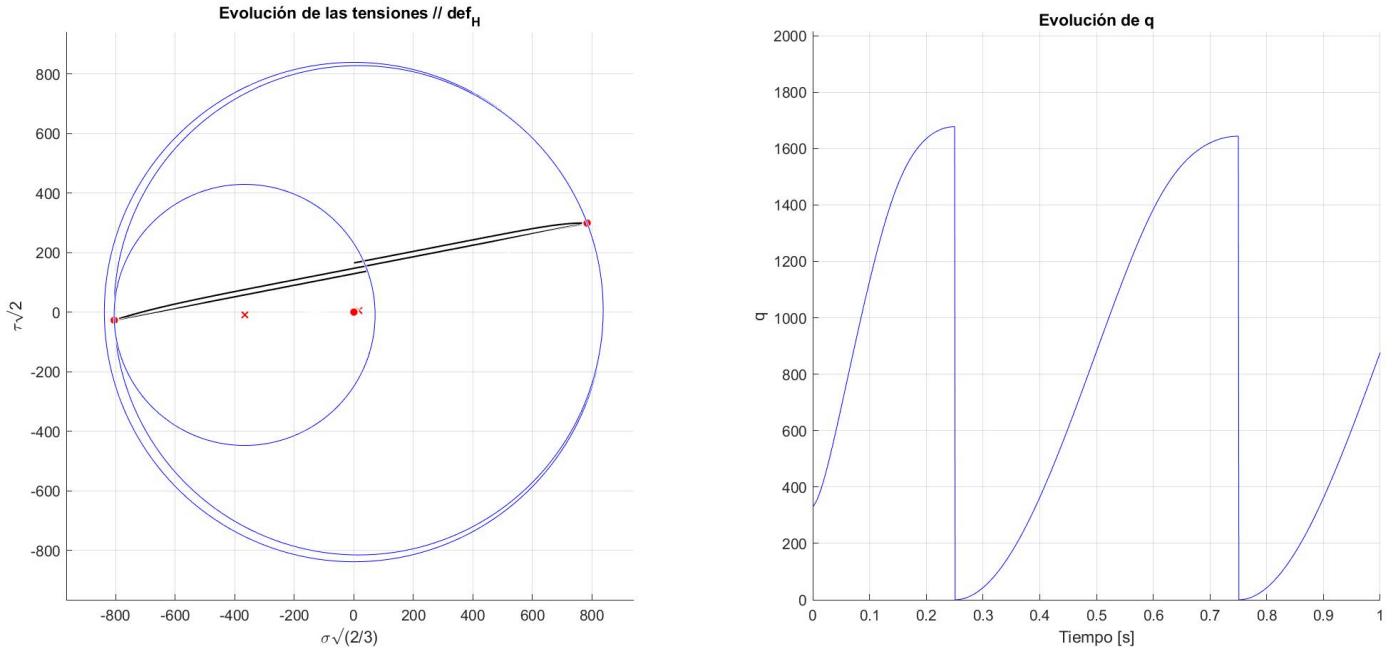

Figura 4.17 Resultado gráfico para el caso H.

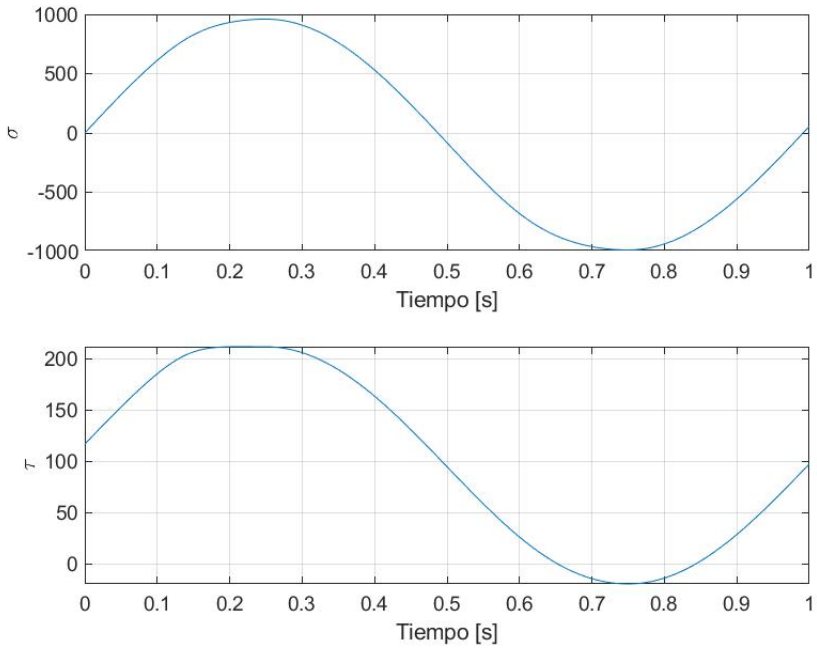

Figura 4.18 Evolución de las tensiones de salida para el caso H.

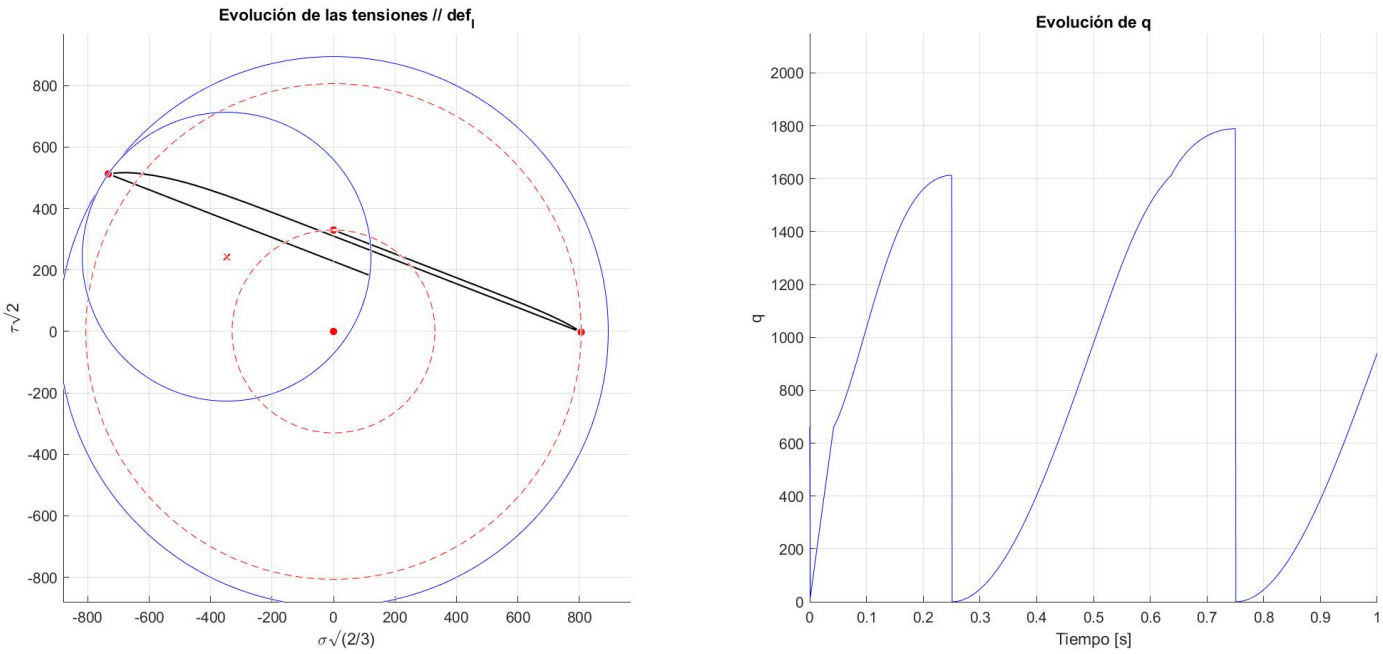

Figura 4.19 Resultado gráfico para el caso I.

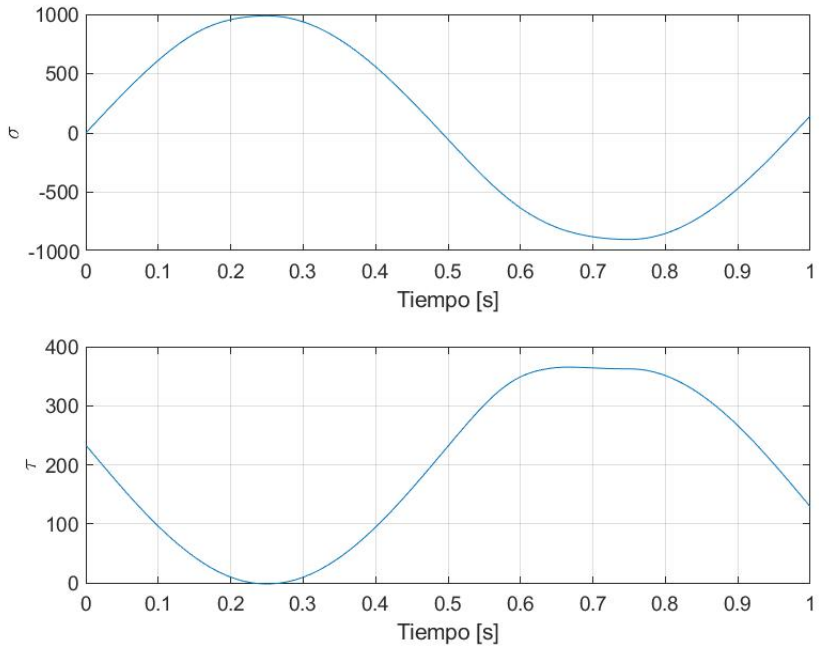

Figura 4.20 Evolución de las tensiones de salida para el caso I.

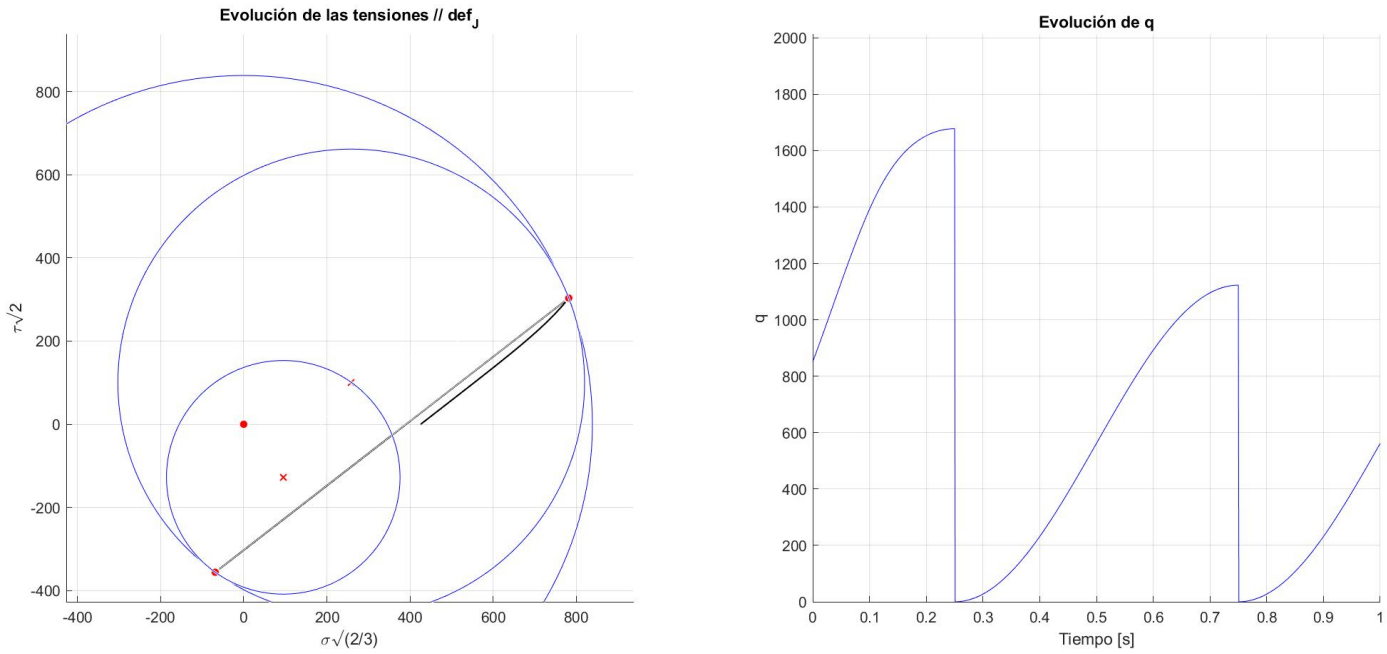

Figura 4.21 Resultado gráfico para el caso J.

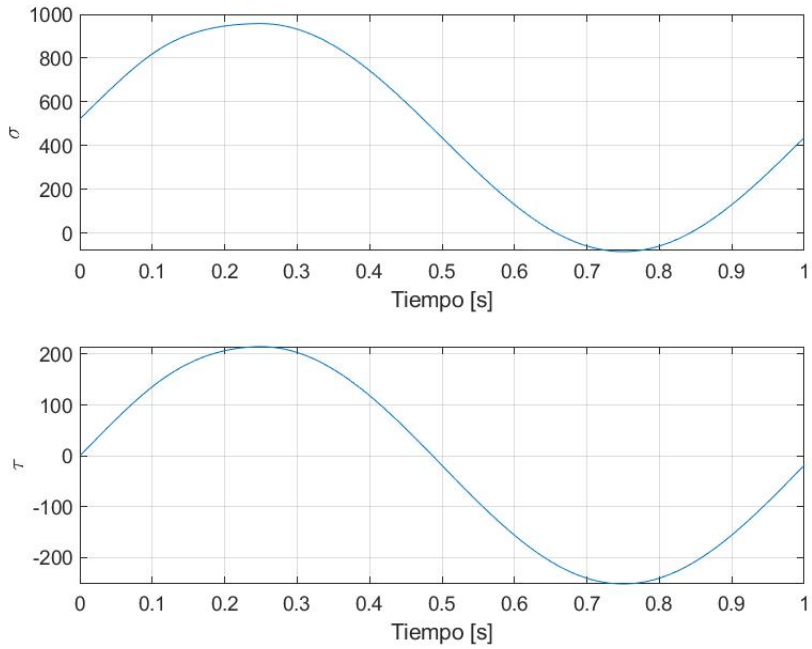

Figura 4.22 Evolución de las tensiones de salida para el caso J.

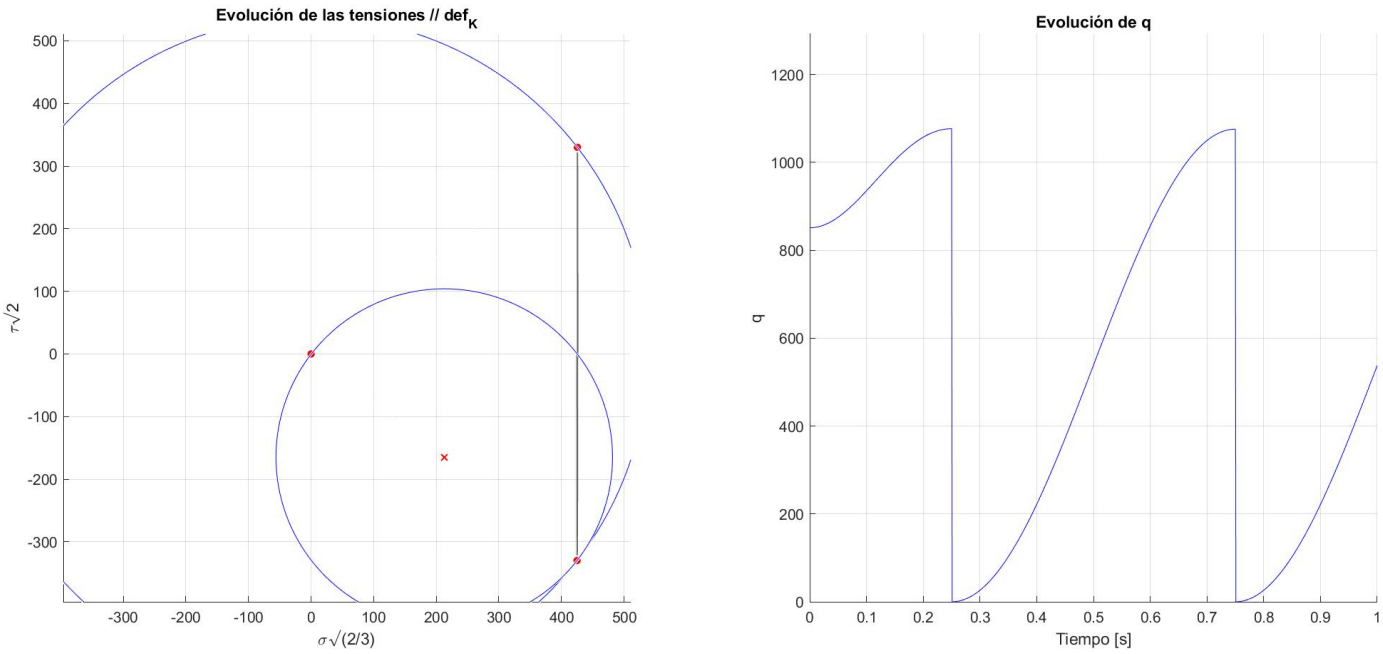

Figura 4.23 Resultado gráfico para el caso K.

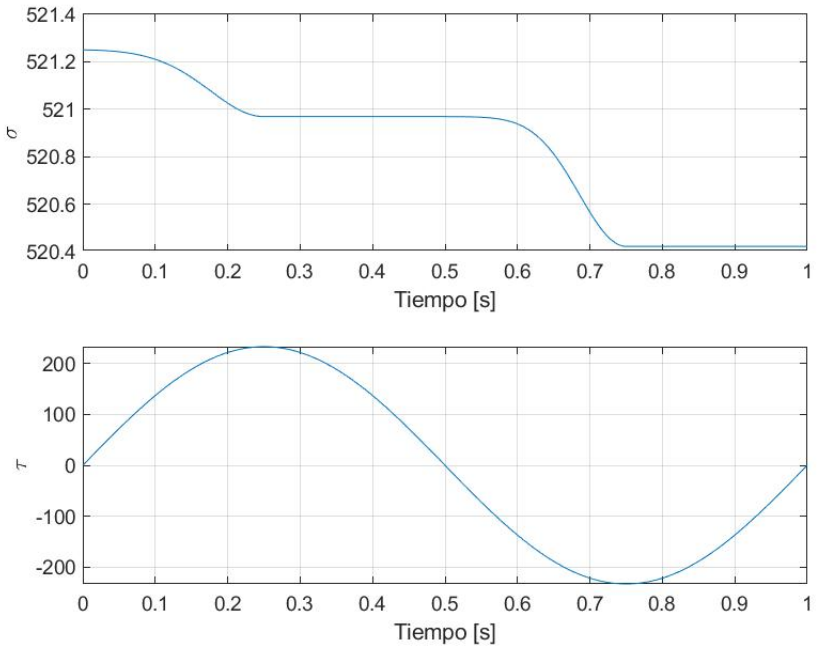

Figura 4.24 Evolución de las tensiones de salida para el caso K.

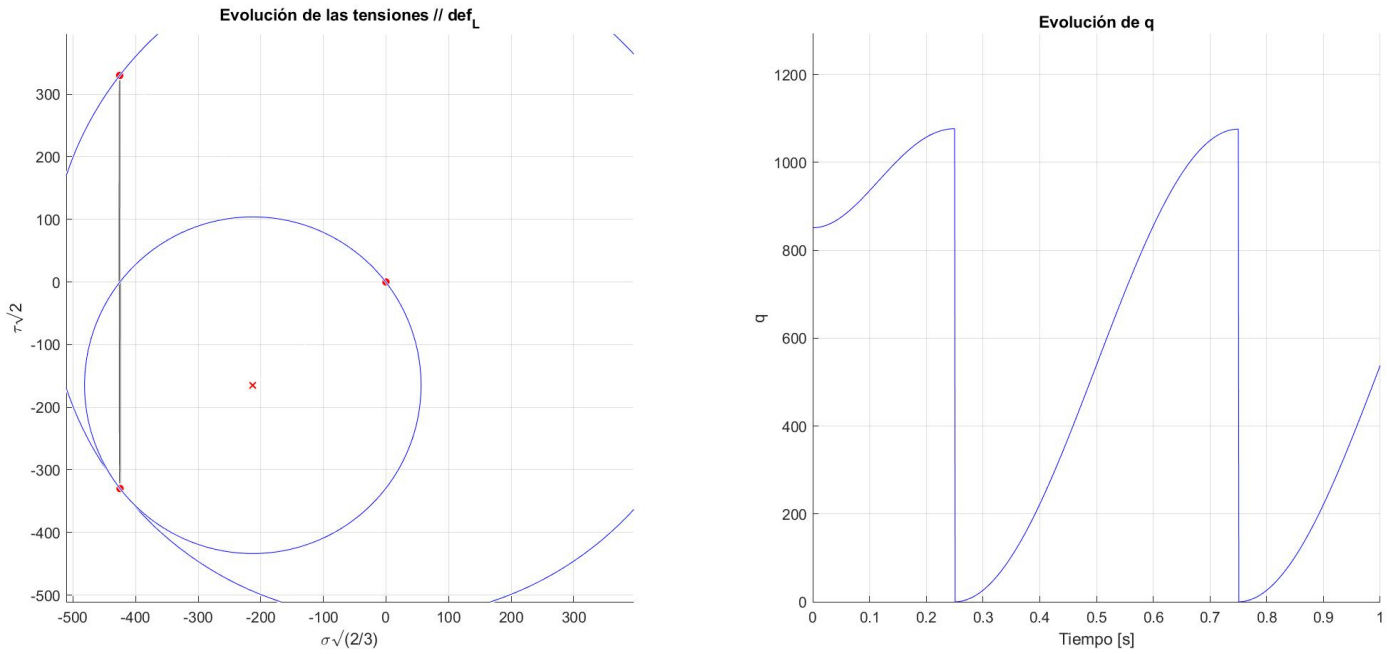

Figura 4.25 Resultado gráfico para el caso L.

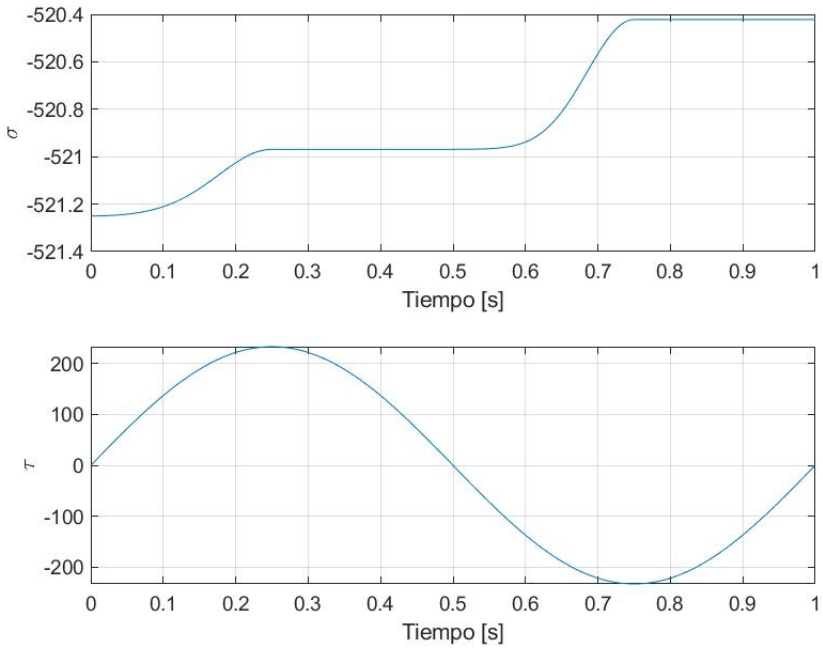

Figura 4.26 Evolución de las tensiones de salida para el caso L.

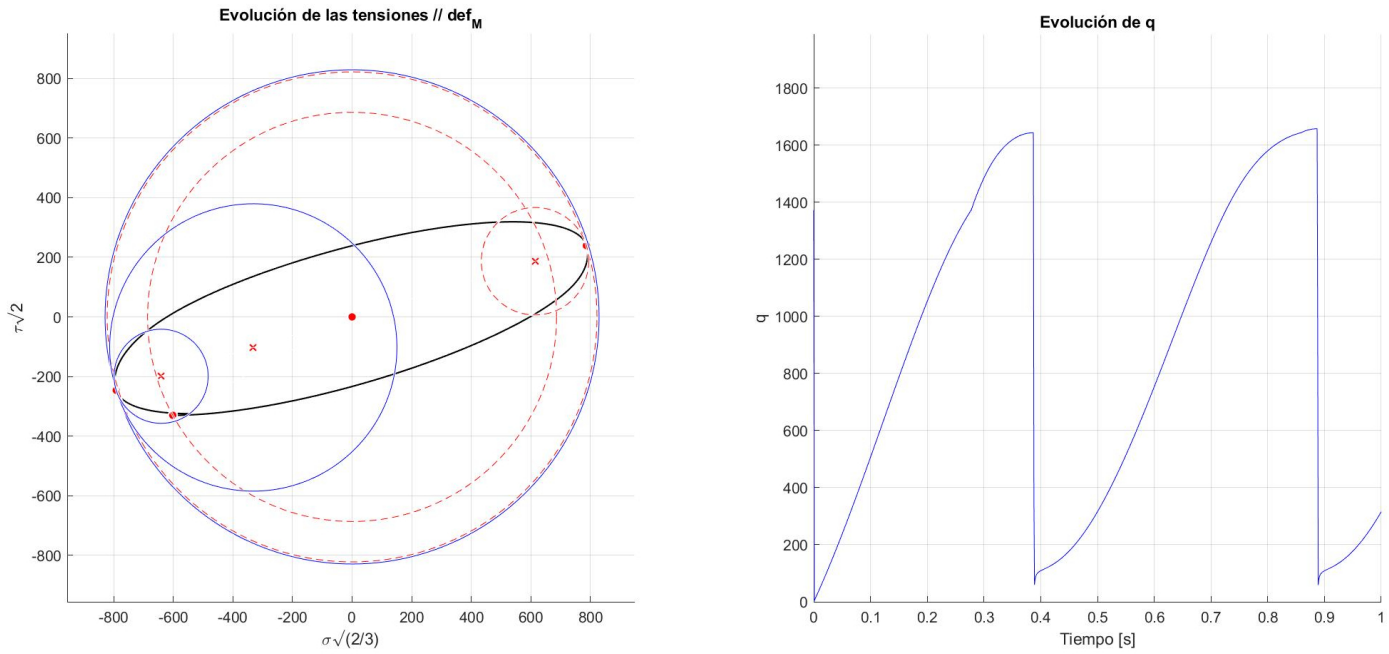

Figura 4.27 Resultado gráfico para el caso M.

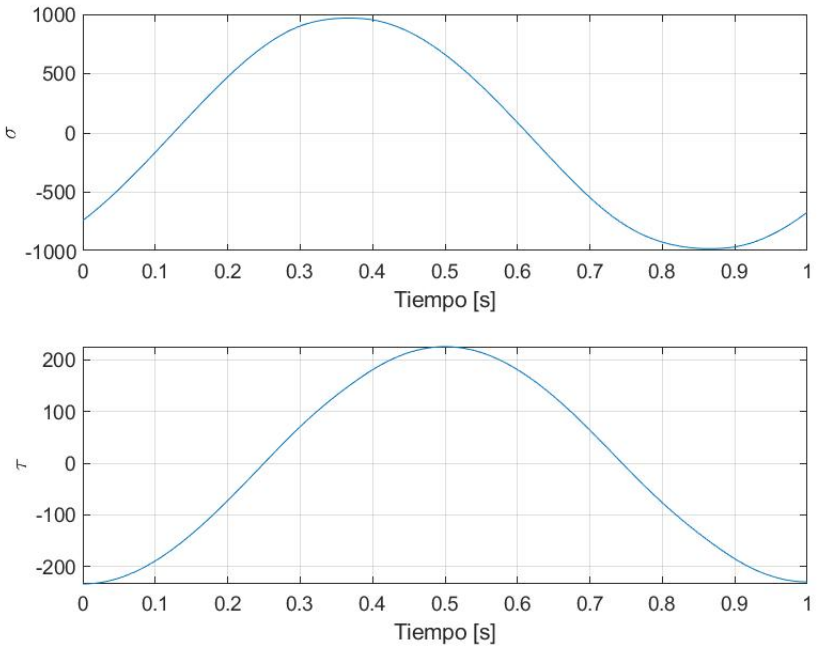

Figura 4.28 Evolución de las tensiones de salida para el caso M.

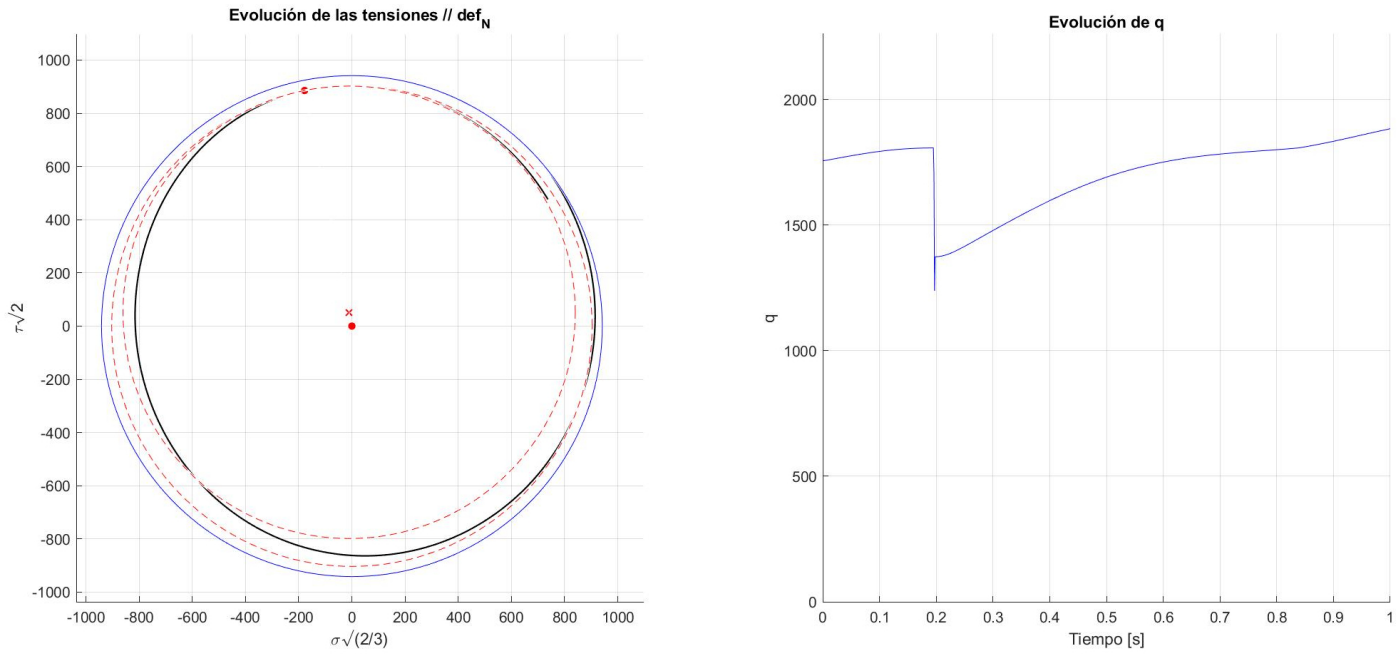

Figura 4.29 Resultado gráfico para el caso N.

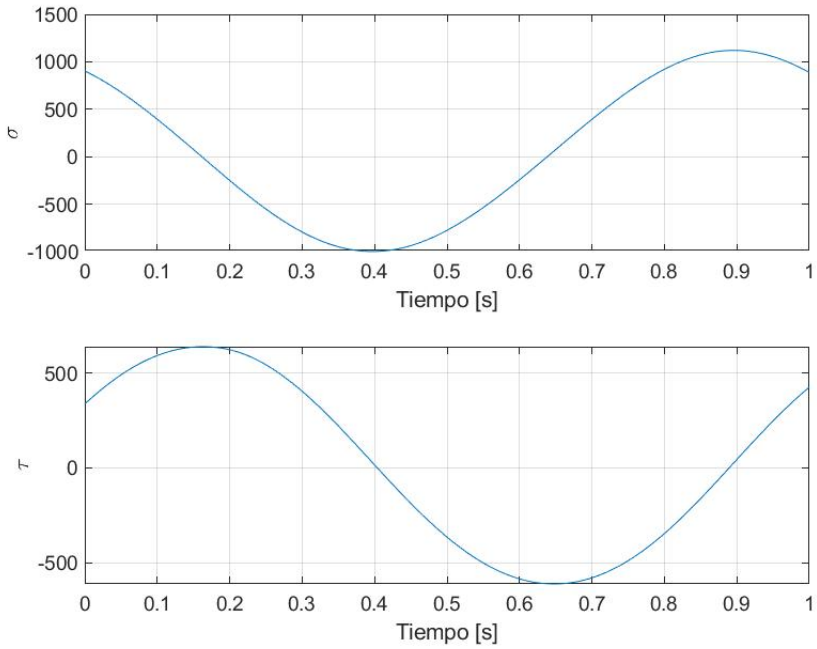

Figura 4.30 Evolución de las tensiones de salida para el caso N.

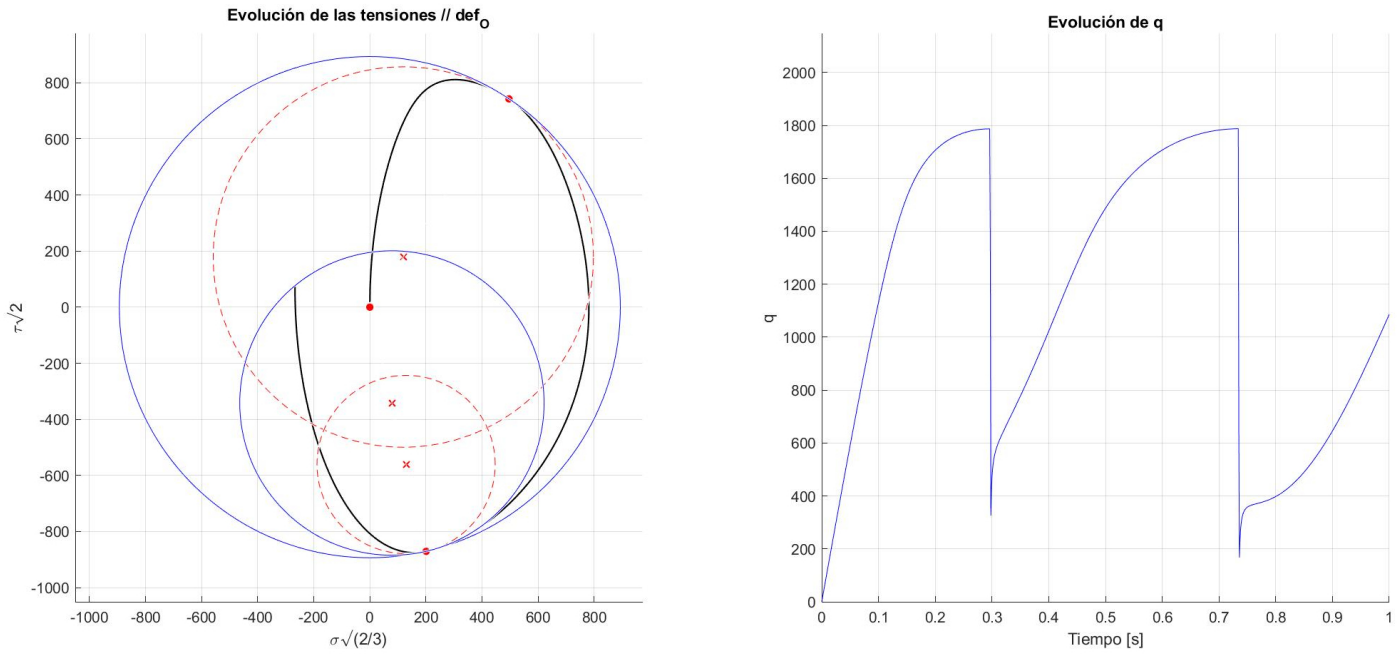

Figura 4.31 Resultado gráfico para el caso O.

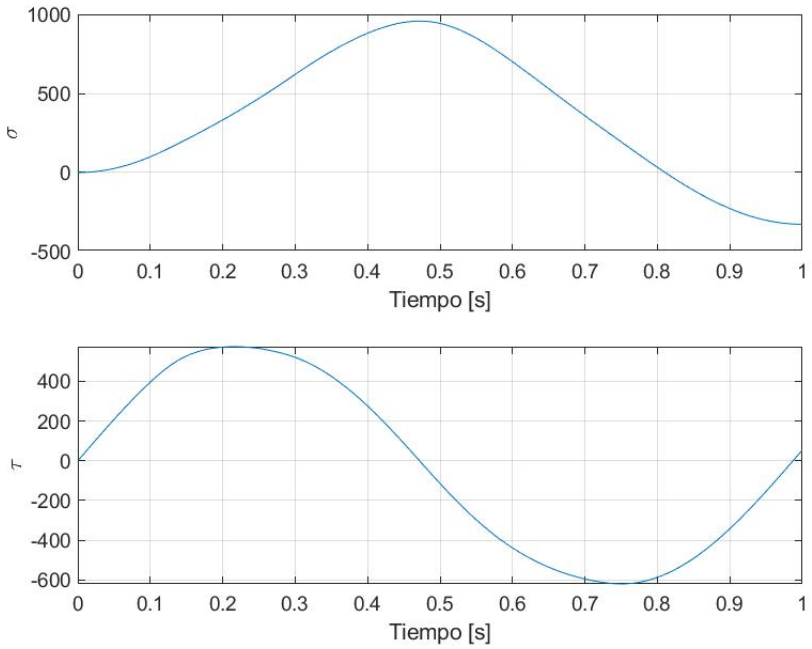

Figura 4.32 Evolución de las tensiones de salida para el caso O.

5 Conclusiones

Este trabajo se ha centrado en el desarrollo de un código versátil y que aporte la máxima cantidad de información a un futuro usuario.

El programa cuenta con un gran variedad de funciones que pueden utilizarse conjuntamente o en solitario para efectuar cálculos y operaciones según el Método de Deformaciones Locales para fatiga multiaxial. Además, se ha hecho un especial énfasis en dotar al usuario de una amplia gama de posibilidades para realizar operaciones. Adicionalmente, se le permite personalizar diversos parámetros para resolver el problema según sus necesidades.

Durante la fase de diseño, todas las funciones han sido probadas para eliminar al máximo los fallos numéricos que podrían invalidar las soluciones, prestando especial atención a las cancelaciones, fenómeno que distorsiona en gran medida los resultados del cálculo. A pesar de los esfuerzos realizados por minimizar la aparición de errores, estos pueden llegar a producirse eventualmente bajo ciertos registros de entrada, por lo que se recomienda comparar las soluciones aportadas utilizando los diferentes métodos y funciones que sirven para lograr el mismo fin.

Finalmente, el programa se postula como una herramienta multitarea para el análisis de registros experimentales, la validación de hipótesis y el cálculo de situaciones sometidas a cargas de fatiga biaxial tensión-torsión.

6 Trabajo futuro

El programa que se ha diseñado cuenta con entidad propia y puede ser considerado un producto cerrado. Aún así, se van a detallar una serie de posibles vías de expansión a tener en cuenta en caso de que fuese necesario ampliar el alcance del mismo. Estos avances se comentan solo de manera explicativa y no forman parte del TFG, pero cabría la posibilidad de implementarlos a largo plazo.

La primera de estas vías estaría centrada en poder analizar varios puntos geométricos de probetas de mayor espesor. Una vez resueltos los desarrollos matemáticos que conllevaría esta ampliación, el código está optimizado para efectuar esta implementación gracias a la POO. Sería necesario declarar un vector de objetos en lugar de una variable $1x1$ y que cada una de las componentes de dicho vector almacenase los datos de varios puntos geométricos. Adicionalmente, se podría añadir como propiedades en la clase punto las coordenadas geométricas o un identificador para cada punto. Finalmente, cuando se hubiese declarado la estructura, habría que programar un nuevo bucle en el que se resolviese el Problema de Valores Iniciales para cada uno de los puntos.

Conjuntamente con el estudio de varias posiciones en la probeta, se podría añadir una función que se encargase de devolver los valores de esfuerzos internos (axil y torsor) a partir de los historiales de tensiones. Estos esfuerzos se compararían posteriormente con el resultado medido por la célula de carga de la máquina de ensayos.

Otro avance importante sería la posibilidad de añadir distintas funciones análogas a las descritas pero que representen el modelo matemático de materiales que exhiban un comportamiento diferente. Esto hace referencia a materiales que no sigan una curva de Ramberg-Osgood o cuyas ecuaciones diferenciales de comportamiento generen otro tipo de matriz $\mathbf{A}(\sigma, \tau)$. De esta manera, se podrían realizar estudios de materiales variados y obtener resultados de distinta naturaleza a los que se han presentado en el apartado 4. En conjunto con lo propuesto en los dos párrafos anteriores, se podrían realizar comparaciones de esfuerzos internos para conocer si esta teoría sería ampliable a otras familias de componentes mecánicos.

Por otro lado, dejando a parte los posibles cambios a realizar en el código, hay que recordar que este proyecto se encuadra dentro de uno mayor: obtener un modelo de predicción de vida a fatiga de la teoría elasto-plástica para fatiga multiaxial.

Asimismo, un desarrollo importante para la aplicación de estas ecuaciones de plasticidad y de los métodos de integración descritos en este TFG sería la implementación de las primeras en un programa de Elementos Finitos. Un ejemplo sería el diseño de rutinas de usuario UMAT para el entorno del programa ABACUS, uno de los software más utilizados en la industria para cálculos elasto-plásticos.

Apéndice A
Manual de Usuario

Este manual pretende ser una guía ilustrativa para que el usuario final opere el programa e interprete los resultados de manera satisfactoria sin necesidad de estar familiarizado con el funcionamiento interno de los códigos. Se recomienda que el usuario añada al *path* de MATLAB la carpeta en la que se encuentran todos los programas para que no se produzcan problemas al realizar llamadas a varias funciones.

A.1 Insertar datos y deformaciones

El usuario final debe introducir dos tipos de datos en el programa: las propiedades físicas del material y el nombre del archivo con el registro temporal del vector de deformaciones.

Las propiedades físicas necesarias son E, G, K y n de la ecuación (1.1). Los datos del material se definen en la clase "datmat" y pueden ser introducidos en las unidades preferidas por el usuario, que ha de ser consciente de que la salida final dependerá de su elección. Por ejemplo, si decide introducir el valor de E, K y G en megapascales, los valores de las tensiones de salida también estarán representados en MPa. Esta declaración debe hacerse antes de ejecutar alguno de los archivos principales y se recomienda guardar la clase "datmat" tras realizar los cambios. No se deben declarar estas propiedades en ningún otro lugar que no sea la clase "datmat". En cada punto del bucle en el que se necesiten estos datos, el programa hará automáticamente una llamada a "datmat" para emplear las propiedades definidas.

Los archivos de deformaciones deben ser documentos .txt en los que el historial esté dispuesto en tres columnas. La primera de ellas debe tener los valores de los instantes de tiempo del intervalo estudiado, las dos siguientes tienen que incluir las componentes del vector de deformaciones para cada uno de esos instantes. No existe limitación en el número de filas del registro, pero las columnas siempre deben mantener el orden siguiente: vector de tiempos, deformaciones normales, deformaciones tangenciales. Es importante que la disposición de los datos sea exactamente la que se describe para que se lean correctamente y no se produzcan errores. En principio, no hay ningún problema en el uso de registros con deformaciones nulas al comienzo, ya que el programa ha sido diseñado para comprobar los valores iniciales y avanzar hasta el instante en el que puede empezar a realizar cálculos sin cometer errores.

Para introducir las deformaciones, se debe escribir el nombre del archivo de texto en la línea correspondiente del fichero principal sin la extensión .txt, como ocurre en la línea 5 del Código 3.15. Se recuerda que el archivo de texto debe estar contenido en la misma carpeta del programa.

Un hecho importante a tener en cuenta es que, según el vector de tiempos de entrada, será necesario emplear una de las dos funciones de diferenciación auxiliares para el cálculo de derivadas.

Si Δt es el mismo a lo largo de todo el intervalo, puede usarse cualquiera de las dos, en cambio, si este incremento es variable, se debe emplear obligatoriamente "CalculaDerivadaSpline".

A.2 Introducir parámetros de funcionamiento y ejecutar el programa

A la hora de ejecutar el programa y obtener los resultados, el usuario puede decidir entre los tres tipos de archivos que se han descrito en el apartado 3:

- **Archivo "mainPOO.m"**: Para resolver el sistema de ecuaciones con paso fijo, que puede ser definido previamente, y almacenar los datos en un objeto.
- **Archivo "mainPOOadapt.m"**: Para resolver el sistema de ecuaciones con paso variable, poder modificar varios parámetros del algoritmo de integración y almacenar los datos en un objeto.
- **Archivo "mainGlobal.m"**: Para resolver el sistema con paso fijo y almacenar los datos en variables globales.

En cualquiera de los casos, se debe ejecutar el archivo "main" que se corresponda con el método preferido y los resultados que se obtengan se habrán calculado mediante el mismo tipo de bucle de memoria y el mismo método de resolución de PVI.

En el caso de los métodos con Δt fijo, se puede modificar dicho paso si se desea, estando fijado en $\Delta t_{def}/10$ por defecto. En el de paso variable también es posible cambiar la tolerancia deseada, así como el número máximo de iteraciones en cada paso y el incremento mínimo de tiempo posible. Todos estos parámetros se modifican en la sección de los archivos "main" en la que se declaran las variables del sistema de resolución de EDOs. Como recomendación, no se aconseja usar una tolerancia menor que 10^{-8} y establecer más de 4 o 5 iteraciones máximas por paso.

Por último, es posible modificar la tolerancia de detección de inversiones y memoria en la línea correspondiente del "main". Este número es nulo por defecto y, según la experiencia a la hora de probar el código, se desaconseja que se aumente a no ser que sea absolutamente necesario. Si se hiciera, no se recomiendan valores mayores a 1 MPa porque se puede llegar a interferir en la solución final.

A.3 Salidas generadas

Una vez que se han establecido los parámetros y los datos de entrada, se debe ejecutar el "main" correspondiente y se obtendrán salidas gráficas y numéricas. En ningún momento debe ejecutarse otro archivo que no sea el fichero principal. Finalmente, las salidas generadas son:

- Una primera salida (figura 1) que evoluciona en tiempo real y que está dividida en dos partes. En la primera de ellas se ve el recorrido de las tensiones a la vez que se representan las circunferencias de cargas, sus centros (marcados con un aspa) y las tensiones de referencia (marcadas por puntos). Las circunferencias se generan en cada inversión y en el momento en el que una es "olvidada" pasa a ser representada con líneas discontinuas. En la segunda parte se ve la representación de la distancia efectiva de tensiones en función del tiempo. La duración de esta representación puede ajustarse cambiando el valor de la variable "duración", que se encuentra en la línea 14 del Código 3.14.
- La segunda de las gráficas que se devuelven ilustra la representación de σ y τ a lo largo del tiempo (figura 2).

- La tercera y última gráfica (figura 3) es una representación del diagrama de deformaciones $\varepsilon - \gamma$ y de estas mismas frente al tiempo. Esto permite al usuario comprobar que los registros de deformaciones introducidos son los correctos.
- Un objeto o estructura global "punto1", donde han quedado registradas todas las variables calculadas para que el usuario tenga fácil acceso a esta información.
- En caso de querer hacer una análisis numérico en lugar de gráfico, se ha declarado una variable "muestra" que devuelve una matriz de n_{tens} valores con columnas que siguen este orden: vector de tiempos de tensiones, σ, τ, q, σ_k, $\sigma_{c,k}$, "reg", "mem" y "prim". Esta matriz podría exportarse a un archivo .xlxs para realizar un estudio en profundidad de los datos obtenidos.

Por último, si se quisieran hacer más comparaciones, se podrían cambiar las funciones auxiliares que calculan q con el coseno del ángulo de los vectores de tensión por aquellas que lo calculan con la ecuación completamente desarrollada, modificando los lugares en los que se llama a "funcioncos" por "funcionqk" y a "funcioncosaux" por "funcionqkaux".

A.4 Algoritmo de tensiones sin integración de deformaciones

Ante la posibilidad de que algún usuario quisiera aplicar este algoritmo a un historial de tensiones en lugar de integrar una evolución de las deformaciones, se ha incluido un archivo extra que efectúa dicha tarea. El nombre de este script es "AlgoritmoTensiones".

Los datos del material son los mismos que los introducidos en "datmat" y deben tener las mismas dimensiones que la entrada de tensiones. El archivo de datos se introduce de la misma manera que si fuesen deformaciones, pero en este caso la segunda columna se corresponde con los valores de la tensión normal y la tercera con los de la tensión tangencial. Al igual que en el caso anterior, es posible declarar el valor de la banda de tolerancia de descargas y memoria. Finalmente, se genera el objeto "punto1", pero con datos exclusivamente en los campos que se han utilizado. Las salidas gráficas en tiempo real son idénticas a las comentadas en el apartado A.3.

A.5 Uso independiente de funciones auxiliares

La mayoría de las funciones auxiliares pueden realizar tareas de manera independiente a los archivos principales. Esto puede ser muy útil para generar datos e información complementaria. A continuación, se detallan todas las tareas y cálculos que se pueden realizar y cuáles son las funciones que se deben usar para ello. Si se desea profundizar en su funcionamiento, se recomienda leer sendos apartados de la sección 3.2.

- **Calcular el módulo de endurecimiento**: Si se da a "HMod" como argumento el valor de q, devolverá el valor de $\phi(q)$ para el caso de descargas. Por otra parte, "HModDes" calcula $\Phi(Q = q/2)$ en descargas.
- **Obtener la matriz $\mathbf{A}(\sigma,\tau)$ para un estado de tensiones**: Pasando como parámetros σ, τ, q y los valores de referencias de tensiones en ese instante, es posible generar la matriz $\mathbf{A}(\sigma,\tau)$ con "GeneraAQ" para cargas y con "GeneraADes" para descargas.
- **Generar el valor de la distancia efectiva de tensiones** q: Conociendo el vector y las referencias de tensiones, se puede emplear "CalculaqCarga" para cargas y "CaclulaqDescos" (que calcula el coseno del vector de cargas) o "CalculaqDesqk" (que emplea la ecuación de descargas completamente desarrollada) para descargas.

- **"CalculaDerivada5Puntos" y "CalculaDerivadaSpline"**: Devuelven las derivadas de un historial de deformaciones (también pueden ser tensiones o cualquier par de variables). La primera de ellas sirve exclusivamente para vectores con paso de tiempo fijo y la segunda también puede emplearse con paso variable.

Índice de Figuras

Índice de Códigos

Bibliografía

[1] *Parámetros y propiedades de ensayo inconel718*, Comunicación privada.

[2] AENOR, *UNE7118: Clases y ejecución de los ensayos de fatiga en los materiales metálicos*, 1958.

[3] Departamento de Matemática Aplicada II Universidad de Sevilla, *Cuadratura y derivación numéricas*, 2018.

[4] Departamento de Matemática Aplicada II. Universidad de Sevilla, *Problemas de ecuaciones diferenciales ordinarias*, 2018.

[5] J R Dormand and P J Prince, *A family of embedded runge-kutta formulae*, 1980.

[6] C. Madrigal, A. Navarro, and C. Vallellano, *Plasticity theory for the multiaxial local strain-life method*, International Journal of Fatigue **100** (2017), 575–582.

[7] MathWorks, *Matlab*, [consulta: 7 mayo 2021]. Disponible en: https://es.mathworks.com/products/matlab.html.

[8] MathWorks, *Share data between Workspaces*, [consulta: 8 junio 2021]. Disponible en: https://es.mathworks.com/help/matlab/matlab_prog/share-data-between-workspaces.html.

[9] MathWorks, *Why use Object-Oriented design*, [consulta: 8 junio 2021]. Disponible en: https://es.mathworks.com/help/matlab/matlab_oop/why-use-object-oriented-design.html.

[10] A. Navarro, *Apuntes del método de las deformaciones locales en fatiga*.

[11] A Navarro, *Plastic flow and memory rules for the local strain method in the multiaxial case*.

[12] A. Navarro, *Differential equations for tension-torsion experiments. Summary*, 2021, Comunicación privada del autor.

[13] NeoNickel, *Alloy 718*, [consulta: 17 mayo 2021]. Disponible en: https://www.neonickel.com/es/alloys/aleaciones-de-niquel/alloy-718/.

[14] Escuela Técnica superior de Ingeniería de Sevilla Área de Ingeniería Mecánica, *Máquina de ensayos biaxial MTS 809*, [consulta: 26 febrero 2021]. Disponible en: http://www.esi2.us.es/ingmec/html/laboratorio/equipos/mts809.htm.