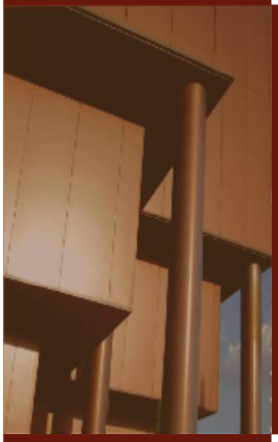
Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías Industriales

Implementación de un método de solución lineal secuencial en el programa de elementos finitos Abaqus para el modelo de interfase elástica lineal frágil

Autor: Enrique Paloma Castro

Tutores: Luis Távara Mendoza, Vladislav Mantič Leščišin



Dep. Mecánica de Medios Continuos y Teoría de Estructuras
Grupo de Elasticidad y Resistencia de Materiales
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2016

Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías Industriales

Implementación de un método de solución lineal secuencial en el programa de elementos finitos Abaqus para el modelo de interfase elástica lineal frágil

Autor:
Enrique Paloma Castro

Tutores:
Luis Távara Mendoza
Vladislav Mantič Leščišin

Dep. Mecánica de Medios Continuos y Teoría de Estructuras
Grupo de Elasticidad y Resistencia de Materiales
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2016

Trabajo Fin de Grado: Implementación de un método de solución lineal secuencial en el programa de elementos finitos Abaqus para el modelo de interfase elástica lineal frágil

Autor: Enrique Paloma Castro

Tutores: Luis Távara Mendoza
Vladislav Mantič Leščišin

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2016

El Secretario del Tribunal

A mi familia

Agradecimientos

Me siento enormemente agradecido a la vida por haber puesto en mi camino a personas, todas ellas, buenas de corazón. Gracias por ese tiempo compartido, por esas experiencias vividas con todas y cada una de ellas. Todas han influido en mí positivamente y han contribuido a formar la persona y el profesional que hoy soy.

Al profesor Federico Paris agradecer su amor y pasión por la docencia. Sus dotes de transmisión, sus ganas de enseñar, su buen trato y preocupación por los alumnos me hicieron descubrir el mundo apasionante de la Elasticidad, a la vez que despertaron en mí la curiosidad por la carrera docente.

A mis tutores Luis Távara y Vladislav Mantic por su paciencia y comprensión. Gracias por sus consejos, apoyo e implicación que han hecho que la realización de este trabajo sea más fácil.

A todos los miembros de mi familia de los que me siento verdaderamente orgulloso y siento verdadera admiración.

A mis padres, Enrique y Mª Paz, por su amor y ternura, por sus ejemplos de vida sencilla, coherente y comprometida. Por enseñarme a disfrutar de las pequeñas cosas que nos regala la vida. Por transmitirme el valor del compromiso, del inconformismo por las injusticias, de la defensa de la dignidad, de la valentía por luchar por un mundo nuevo. Gracias por el regalo de la fe.

A mis hermanas Olga y Virginia, a las que adoro, por sus muestras constantes de cariño, por tener siempre las palabras que necesito, por estar ahí siempre y por ser ejemplos de mujeres trabajadoras y comprometidas.

A mis sobrinas Paula y Gabriela, por ser generadoras de felicidad aún sin saberlo, por despertar en mí infinidad de sentimientos nuevos y sacar lo mejor de mí.

Resumen

Muchos problemas que aparecen en los materiales compuestos presentan fuertes inestabilidades (tipo "snap-back") que los métodos tradicionales de solución (Newton-Raphson, Arc-length, entre otros) implementados en los códigos de elementos finitos comerciales no son capaces de captar. Por medio de una programación en Python [19] se pretende usar las soluciones elásticas que proporciona un programa comercial de elementos finitos (Abaqus) e implementar el método de solución lineal secuencial (SLA) que permitiría captar este tipo de inestabilidades. Este método de solución ha sido implementado previamente en un código de elementos de contorno del Grupo de Elasticidad y Resistencia de Materiales [22-25], lo que ha permitido probar su eficiencia y fiabilidad.

El uso de Python como herramienta de programación de Abaqus [19] nos da la posibilidad de ejecutar diferentes comandos en las etapas de pre-proceso, solución y post-proceso del código. En otras palabras permite un control casi completo de Abaqus por medio de comandos (scripts).

El comportamiento de la interfase se modelará a través de una subrutina de usuario UMAT (programada en Fortran) de un modelo de Interfase Elástica Lineal Frágil (LEBIM) que incluye un modo mixto de fractura.

De manera específica se aplica esta herramienta numérica al modelo del ensayo de tenacidad interlaminar en materiales compuestos, obteniéndose unos resultados similares a los obtenidos por D. Castillo [6]. Concluyendo que esta herramienta permite calcular la carga de fallo en problemas con grietas de interfase de manera satisfactoria.

Índice

Índice de Tablas

Índice de Figuras

Notación

GERM	Grupo de Elasticidad y Resistencia de Materiales
FEM	Método de los Elementos Finitos (Finite Element Method)
BEM	Método de los Elementos de Contorno (Boundary Element Method)
LEBIM	Modelo de Interfase Elástica Lineal Frágil (Lineal Elastic Brittle Interface Model)
SLA	Sequential Linear Analysis
LEFM	Linear Elastic Fracture Mechanics
ODB	Output Database
UMAT	User Material Subroutine
DCB	Doble viga en voladizo

1 INTRODUCCIÓN

El presente trabajo ha sido desarrollado en el Grupo de Elasticidad y Resistencia de Materiales (GERM), Departamento de Mecánica de Medios Continuos y Teoría de Estructura de la Universidad de Sevilla.

Este grupo comenzó a estudiar problemas de fractura en materiales compuestos a comienzos de 1990. Entre sus diversas líneas de investigación se encuentra: El desarrollo e implementación de diferentes modelos de mecánica de la fractura en códigos de Método de Elementos Finitos (FEM) y Método de Elementos de Contorno (BEM). El objetivo de este trabajo es contribuir al avance en dicha línea de investigación.

En los últimos años el GERM de la Universidad de Sevilla ha desarrollado un nuevo modelo de interfase llamado "Modelo de Interfase Elástica Lineal Frágil" (LEBIM). Si bien existía un modelo muy parecido a este para modelar pequeñas capas de adhesivo en las uniones [7], el primer documento donde se implementa este comportamiento para modelar la unión fibra-matriz es en el desarrollado por L. Távara y colaboradores [22], en el que se implementa este comportamiento en un código BEM.

D. Castillo (2014) implementó este modelo de interfase en el programa de elementos finitos Abaqus [6]. Lo aplicó a diversos problemas de fractura de materiales compuestos utilizando los algoritmos de resolución incorporados en Abaqus (Newton-Raphson y Arc-length).

Los modelos de iniciación de grieta en la interfase presentan inestabilidades que hace que la solución sea difícilmente obtenible por los algoritmos tradicionales de resolución. L. Távara y colaboradores [22] han desarrollado un algoritmo de resolución en BEM capaz de obtener la solución aún produciéndose una inestabilidad muy fuerte, este algoritmo es llamado "Sequential Linear Analysis" (SLA).

El presente trabajo intenta dar un paso más en esta línea de investigación y llevar a cabo la implementación de este método de solución lineal secuencial, SLA, en el programa de elementos finitos ABAQUS para el modelo de interfase LEBIM.

Este nuevo aporte permitirá:

- Modelar más fácilmente geometrías complejas
- Tener el control de la propagación de la grieta

El objetivo general a largo plazo de este trabajo es contribuir al desarrollo de criterios de fallo con base física para los materiales compuestos.

1.1 Motivación

El conocimiento actual sobre los daños y mecanismos de fallos en los materiales compuestos no es lo suficientemente profundo como para permitir el desarrollo de criterios de fallo de base cien por cien física. Sin embargo, el uso de estos materiales en diversos sectores industriales como el aeronáutico, automovilístico, naval, de energías renovables, etc, crece de manera continuada en los últimos años. Este hecho hace especialmente interesante investigar en esta área y lograr un mayor entendimiento de los mecanismos de fallos que pueden aparecer en este tipo de materiales.

El uso de modelos alternativos a la Mecánica de la Fractura Lineal Elástica Clásica (LEFM), parece ser una prometedora herramienta, primero para entender estos mecanismos de fallos y en segundo lugar, para desarrollar diseños de estructuras de composite.

1.2 Daño en Materiales Compuestos

Algunos de los daños y mecanismos de fallo más importantes tienen lugar a diferentes escalas del composite. A escala macroscópica, los fallos más importantes que exhiben estos materiales son la propagación de grietas en las capas adhesivas que unen las láminas del composite y la delaminación entre las diferentes láminas. A escala microscópica, es decir del orden del radio de la fibra, se puede producir el fallo de la fibra si la dirección de la carga es la misma que la de la fibra (dirección longitudinal), o que aparezcan grietas en las direcciones transversales a la fibra.

Lo deseable es que el material compuesto trabaje con cargas cuya dirección sea la de la fibra, ya que es esa dirección la que presenta mejores propiedades. En la dirección transversal a la fibra, las propiedades son peores por lo que grietas en este plano son realmente desfavorables. En la mayoría de los casos estas grietas se inician en la interfase que existe entre la fibra y la matriz, y una vez que crece esta se propaga hacia la matriz.

1.3 Modelos de Mecánica de la Fractura no singular

La mayoría de los métodos basados en LEFM permiten modelar la propagación de la grieta pero no la iniciación de la misma. Por ello, durante los últimos años, otros modelos han sido desarrollados como son los Modelos de Zona Cohesiva (CZM) [3, 5, 9, 18] y los Modelos de Interfase Lineal Elástica Frágil (LEBIM) [7, 8, 11]. Estos modelos permiten estudiar tanto la iniciación como el crecimiento de la grieta.

1.3.1 Modelo de Zona Cohesiva

Fue desarrollado en los años 60 para modelar tanto roturas frágiles como roturas dúctiles. Este modelo es ampliamente usado para modelar daños en el hormigón [5]. En los últimos años también se ha aplicado para el estudio de fallo en materiales compuestos, tanto en problemas de delaminación, como en problemas de rotura de la interfase fibra-matriz [4, 12, 13, 14, 21].

La novedad que introdujo este modelo es que partía de la hipótesis de la existencia de una zona cohesiva tras la rotura de la grieta, donde aun estando el material dañado, las tensiones en la zona cohesiva son diferente de cero. De esta manera, se elude la presencia de singularidades de tensiones en el entorno del vértice de la grieta, cosa que si aparece en los modelos basados en la LEFM.

En la Figura 1-1 se muestra el esquema de una grieta bajo este modelo y como sería la evolución de tensiones a lo largo de ella. La diferencia existente entre los diferentes modelos de zona cohesiva es la evolución de las tensiones a lo largo de la zona cohesiva, en la Figura 1-2 se presentan algunos ejemplos de los diferentes modelos utilizados.

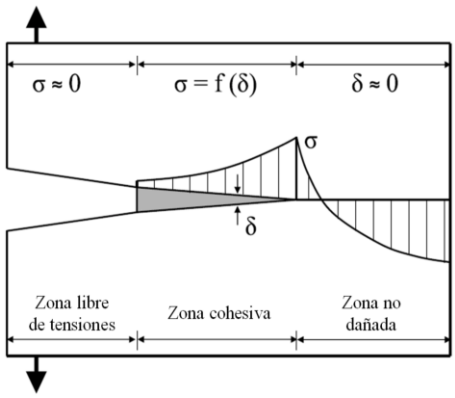


Figura 1-1 Evolución de las tensiones normales según el CZM para el modo I de fractura

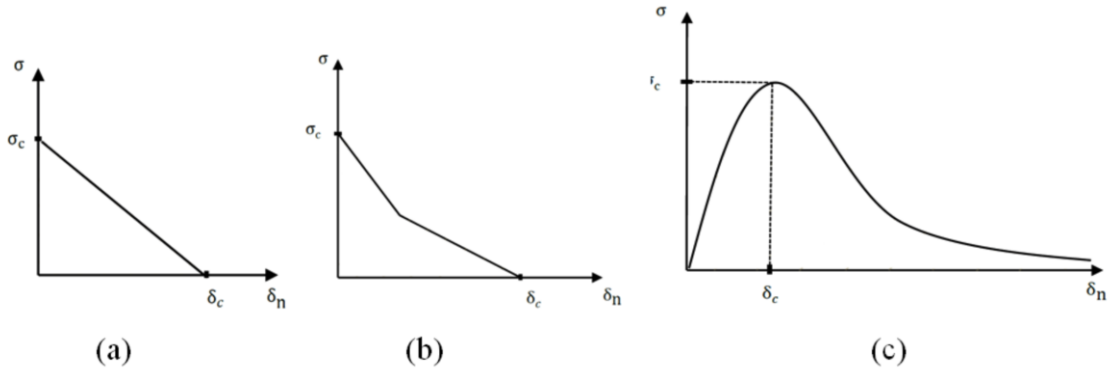


Figura 1-2 Gráficas tensión normal-deformación normal para diferentes modelos cohesivos (a) Lineal (b) Bilineal (c) Xu and Needleman [6]

1.3.2 Modelo de Interfase Elástica Lineal Frágil

Una alternativa a la hora de describir el comportamiento de sólidos unidos es modelar una capa elástica (adhesivo), a veces llamada interfase, como una distribución continua de muelles elásticos lineales con parámetros de rigideces adecuados.

El modelo nació para modelar capas de adhesivo de pequeño espesor [7, 8, 15] y su extrapolación para modelar el comportamiento de una interfase fue llevada a cabo por el GERM de la Universidad de Sevilla, concretamente por Luis Távara y colaboradores [22], denominando a este modelo de interfase como "Linear Elastic Brittle Intefase Model" (LEBIM).

Cabe destacar que este modelo ha sido utilizado en un código basado en BEM, para modelar la rotura de la interfase fibra-matriz en materiales compuestos, tanto en modelos de una fibra [23] como multifibra [24].

Al ser este modelo el empleado en el presente trabajo, se pasa a continuación a describir en primer lugar la ley constitutiva de la distribución de muelles y en segundo lugar el criterio de fallo de la interfase.

1.3.2.1 Ley constitutiva de la distribución de muelles

La distribución continua de muelles está gobernada por una ley constitutiva que prescribe la relación entre tensiones y desplazamientos relativos en la interfase, como se ilustra en la Figura 1-3.

La siguiente ley lineal-elástica (1.1a) relaciona tensiones y desplazamientos relativos en un punto x de la interfase situada en una zona no dañada de la misma, Figura 1-3 a, b.

$$\text{Interfase no dañada} \begin{cases} \sigma(x) = k_n \delta_n(x) \\ \tau(x) = k_t \delta_t(x) \end{cases} \quad \text{para} \quad G(x) < G_c\big(\psi(x)\big) \tag{1.1a}$$

Donde:

- $\sigma(x)$ y $\tau(x)$ son las tensiones normales y tangenciales respectivamente
- δ_n y δ_t son los desplazamientos normales y tangenciales entre puntos opuestos de la interfase
- k_n y k_t son las rigideces normal y tangencial de la distribución de muelles
- G es el índice de liberación de energía
- G_c es la energía de fractura de la interfase (conocida también como tenacidad a la fractura)
- ψ es el ángulo de mixticidad

$$\tan\varphi = \frac{\tau}{\sigma\sqrt{\xi}} \quad \text{donde} \quad \xi = \frac{k_t}{k_n}$$

Cabe destacar en primer lugar que para el caso de una interfase lineal, *G* se define como la energía de deformación elástica almacenada (por unidad de longitud) en un punto x "interfase-muelle" no dañado (segmento de interfase infinitesimal). En segundo lugar que la energía de fractura es función del ángulo de mixticidad del modo de fractura en el punto de la interfase, por lo que se podría obtener diferentes valores de esta variable crítica dependiendo del punto de la interfase en que se encuentre.

Una vez rota la interfase, la siguiente ley constitutiva no lineal (1.1b) es considerada en un punto x de la interfase, Figura 1-3 c, d.

$$\text{Interfase dañada} \begin{cases} \sigma(x) = k_n \langle \delta_n(x) \rangle_- \\ \tau(x) = 0 \end{cases} \tag{1.1b}$$

El operador $\langle\cdot\rangle_+$ se conoce como corchete de Macaulay o función rampa. $\langle\cdot\rangle_+$ parte positiva, $\langle\cdot\rangle_-$ parte negativa.

El comportamiento descrito en las ecuaciones (1.1) se puede describir de la siguiente manera:

Cuando el índice de liberación de energía en un "muelle" alcanza el valor de la tenacidad a fractura, la rigidez tangencial de este muelle pasa a ser cero, y por lo tanto el valor de la tensión tangencial también.

Analizando el comportamiento en la dirección normal, si las tensiones son de tracción, el comportamiento es equivalente al tangencial descrito anteriormente. En cambio, sí a "un muelle roto" se le somete a compresión este sigue teniendo rigidez, es decir, se está utilizando una condición de contacto sin rozamiento (tipo penalti). El uso de esta condición de contacto está basada en la idea de que una vez rota la interfase, parece razonable pensar que esta se puede comprimir con la misma rigidez en la dirección radial (normal si es una capa de adhesivo) que antes de la rotura.

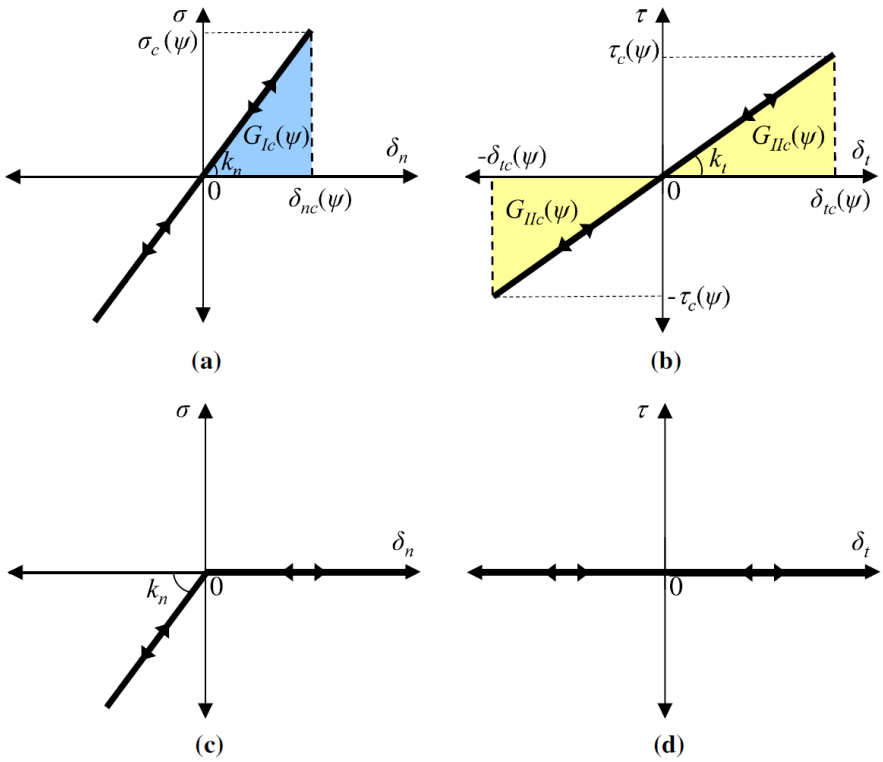


Figura 1-3 Comportamiento (a) normal y (b) tangencial de la interfase según el modelo LEBIM cuando esta no se encuentra dañada. Comportamiento (c) normal y (d) tangencial de la interfase cuando esta se encuentra dañada [17].

1.3.2.2 Criterio de fallo de la interfase

El criterio de fallo de la interfase es definido en términos del índice de liberación de energía G y de la tenacidad a fractura G_c. Un punto de la interfase, no necesariamente un punto del fondo de grieta, rompe cuando G (1.3) alcanza la energía de fractura G_c, la cual depende del ángulo de mixticidad del modo de fractura, es decir

$$G = G_c(\psi) \tag{1.2}$$

G de una grieta en modo mixto en una interfase elástica lineal se puede definir como [16]:

$$G(x) = G_I(x) + G_{II}(x) \tag{1.3a}$$

Con

$$G_I(x) = \frac{\langle\sigma(x)\rangle_+\langle\delta_n(x)\rangle_+}{2} = \frac{\langle\sigma(x)\rangle_+^2}{2k_n} = \frac{k_n\langle\delta_n(x)\rangle_+^2}{2} \tag{1.3b}$$

$$G_{II}(x) = \frac{\tau(x)\delta_t(x)}{2} = \frac{\tau^2(x)}{2k_t} = \frac{k_t\delta_t^2(x)}{2} \tag{1.3c}$$

Verificando que $G_I(x) = 0$ para $\sigma(x) \leq 0$.

Una ley fenomenológica ampliamente aceptada para la energía de fractura de la interfase que es función del modo de mixidad fue propuesta por Hutchinson y Suo

$$\hat{G}_c(\psi) = \frac{G_c}{\bar{G}_{Ic}} = 1 + tan^2((1-\lambda)\psi) \tag{1.4}$$

Donde [22]

$$\bar{G}_{Ic} = \frac{\overline{\sigma}_c^2}{2k_n} \tag{1.5}$$

es la energía crítica a fractura en el modo I. $\overline{\sigma}_c$ representa la tensión normal que provoca la rotura del "muelle" de la interfase y λ es un parámetro de sensibilidad al modo de fractura, cuyo valor suele rondar $0.2 \leq \lambda \leq 0.3$, estos valores hacen que haya una dependencia moderada del modo II de fractura.

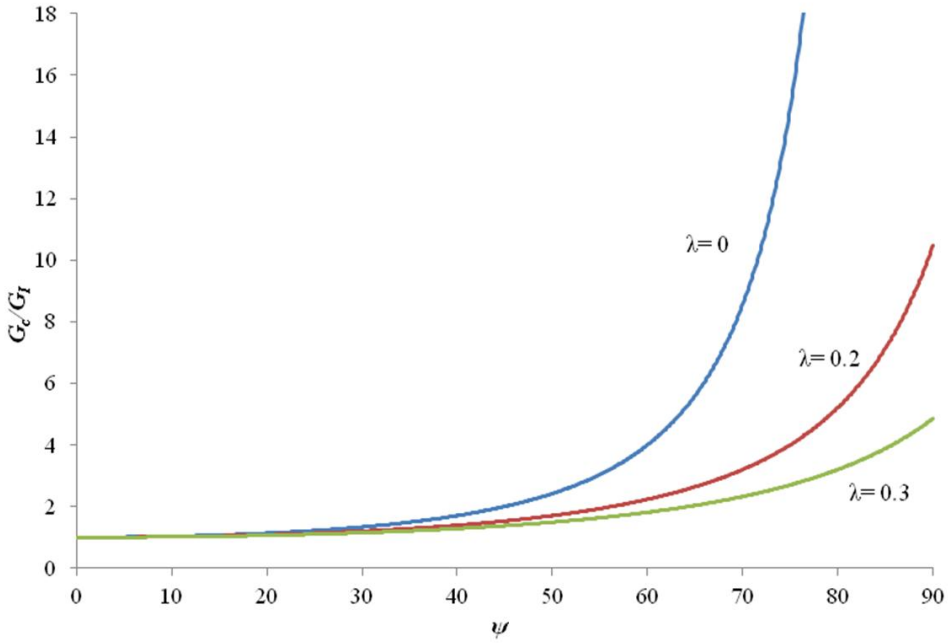


Figura 1-4 Evolución del índice de liberación de energía crítico, G_c, en función de ψ y de diferentes valores de λ [6]

Como se observa en la Figura 1-4, si λ=0 la interfase nunca romperá en modo II puro, debido a la existencia de una asíntota para ψ=90º. Otra conclusión es que la rotura en el modo II será más fácil que aparezca cuanto mayor sea el valor de λ.

Se puede expresar la tensión normal y tangencial crítica en función del ángulo de mixticidad de la siguiente manera [17]:

$$\hat{\sigma}_c(\psi) = \frac{\sigma_c(\psi)}{\overline{\sigma}_c} = \sqrt{\hat{G}_c(\psi)} \cdot \begin{cases} \cos\psi, & |\psi| \leq \frac{\pi}{2} \\ -|\cot\psi|, & |\psi| \geq \frac{\pi}{2} \end{cases} \tag{1.6a}$$

$$\hat{\tau}_c(\psi) = \frac{\tau_c(\psi)}{\overline{\sigma}_c} = \sqrt{\xi}\sqrt{\hat{G}_c(\psi)} \cdot \begin{cases} \sin\psi, & |\psi| \le \frac{\pi}{2} \\ \text{sign}\,\psi\,, & |\psi| \ge \frac{\pi}{2} \end{cases} \tag{1.6b}$$

Las gráficas de las curvas de fallo de la interfase parametrizadas por las ecuaciones (1.6), en el plano de tensiones normalizadas de la interfase $(\sigma/\overline{\sigma}_c\,, \tau/\overline{\sigma}_c)$, considerando $\xi = 0.25$, son mostradas en la Figura 1-5. Una de las conclusiones que se puede sacar de esta gráfica es que si a la interfase se le somete a tensiones normales de compresión hace que aumente la resistencia de esta al modo II.

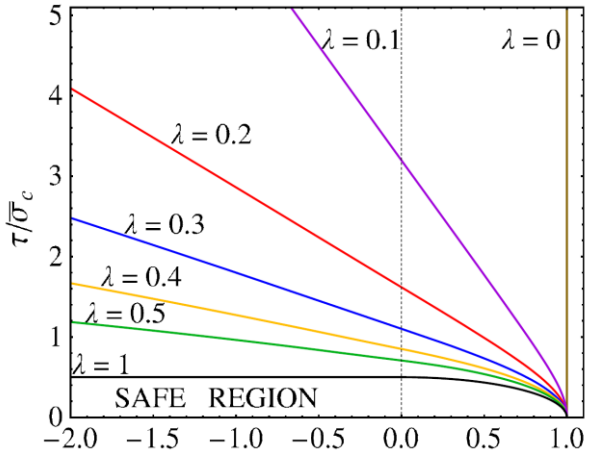


Figura 1-5 Curvas de fallo de la interfase $(\hat{\sigma}(\psi), \hat{\tau}(\psi))$ en el plano $(\sigma/\overline{\sigma}_c\,, \tau/\overline{\sigma}_c)$ usando la ley de Hutchinson y Suo con diferentes valores de λ y $\xi = 0.25$ [17]

1.4 Algoritmos de resolución

A continuación se pasa a describir brevemente los tres algoritmos de resolución utilizados por Abaqus para resolver problemas no lineales: Newton-Raphson, Newton-Raphson añadiendo un factor de amortiguamiento ficticio ("automatic-stabilization" en Abaqus) y el algoritmo de Arc-length de Riks modificado. Destacar que serán explicados teniendo en cuenta las particularidades de su implementación en dicho software.

Como se mencionó al comienzo de este capítulo, los modelos de iniciación de grieta en la interfase presentan inestabilidades que hacen que la solución sea difícilmente obtenible a través de los algoritmos tradicionales. L. Távara y colaboradores [22] desarrollaron un algoritmo de resolución en BEM capaz de obtener la solución aun produciéndose una inestabilidad fuerte, este algoritmo es llamado "Sequential Linear Analysis" (SLA). Al ser empleado en este trabajo se realizará una explicación detallada de este método en el subapartado 1.4.4.

1.4.1 Newton-Raphson

Este algoritmo es uno de los más utilizados para el análisis de elementos finitos a la hora de resolver ecuaciones no lineales.

Es relativamente fácil de visualizar en el caso de una dimensión, es decir, cuando el vector de movimientos nodales a tiene una sola componente. Las ecuaciones planteadas por el teorema de los

trabajos virtuales, particularizada para el caso de los elementos finitos, se puede escribir de forma simbólica de la siguiente manera:

$$F_{int}(a) = F_{ext} \quad (1.7)$$

Esta ecuación se puede visualizar como: los movimientos que generan las fuerzas sobre el sistema, deben de crear una fuerza interna igual a la fuerza externa aplicada.

La carga exterior se divide en incrementos. En Abaqus el usuario puede fijar el valor de estos incrementos de carga o dejar que este elija estos valores de forma óptima utilizando un algoritmo [1]. En cualquier caso la fuerza exterior aplicada se descompone en una suma de incrementos:

$$F_{ext} = \sum_n (F^{n+1} - F^n) = \sum_n \Delta F^n \quad (1.8)$$

Con ayuda de la

Figura 1-6 que representa la gráfica de fuerza-desplazamiento de un problema de una dimensión, se explica a continuación este algoritmo.

Suponiendo que se parte de un punto de equilibrio, es decir que forma parte de la solución, (a^n, F^n) se traza una línea tangente a la curva, cuya pendiente será la rigidez en ese punto, K^n. La aproximación al siguiente punto de equilibrio será el desplazamiento obtenido al aplicar una fuerza F^{n+1} a un sistema que pasa por el punto (a^n, F^n) y que tiene rigidez K^n, es decir:

$$a_1^{n+1} = a^n + \frac{F^{n+1} - F^n}{K^n} \quad (1.9)$$

El residuo se define como la diferencia entre el valor de la fuerza aplicada F^{n+1} y el de la fuerza del sistema real $F_{int}(a^{n+1})$.

$$r_1^n = F^{n+1} - F_{int}(a_1^{n+1}) \quad (1.10)$$

Si el valor de este residuo es lo suficientemente bajo, se tomará el punto (a^{n+1}, F^{n+1}) como el siguiente punto de equilibrio. En caso contrario se refinaría la aproximación trazando una línea tangente por el punto $\left(a_1^{n+1}, F_{int}(a_1^{n+1})\right)$ cuya pendiente sería la rigidez del sistema para ese punto, K_1^n. La nueva aproximación a la solución serían los desplazamientos obtenidos para un sistema cuya rigidez es K_1^n y pasa por el punto $\left(a_1^{n+1}, F_{int}(a_1^{n+1})\right)$ y se le aplica una fuerza F^{n+1}, es decir:

$$a_2^{n+1} = a_1^{n+1} + \frac{r_1^{n+1}}{K_1^n} \quad (1.11)$$

Se vuelve a calcular el residuo para este caso y si cumple con las tolerancias impuestas, se toma este punto como solución. En caso contrario se procedería de forma análoga.

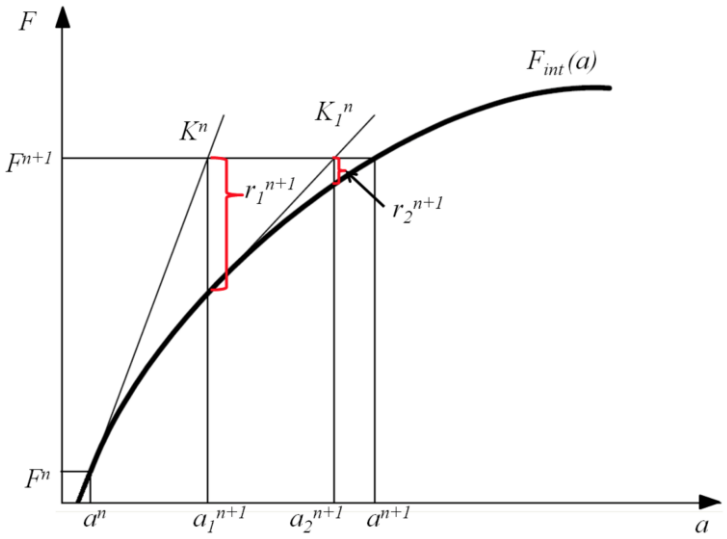


Figura 1-6 Esquema del algoritmo de Newton-Raphson

Lo expuesto anteriormente describe el procedimiento clásico de Newton, cuya mayor desventaja es el cálculo de la rigidez que en caso de un modelo multidimensional es una matriz Jacobiana para cada iteración. Existen modificaciones de este algoritmo, los llamados cuasi-Newton, los cuales para mejorar el tiempo en que se alcanza la solución, modifican el cálculo de la matriz Jacobiana. De los algoritmos cuasi-Newton existen las variantes que utilizan la rigidez o matriz Jacobiana de la primera iteración para todas las demás, o una intermedia entre la de la primera iteración y la que se está utilizando.

Estos algoritmos dan muy buenos resultados para problemas con una no linealidad no muy fuerte, sin embargo sus resultados no son tan buenos para no linealidades fuertes.

1.4.2 Newton-Raphson con factor de amortiguamiento ficticio

Para modelos con una inestabilidad muy fuerte, como son el caso de modelos donde se produce el pandeo o inestabilidades tipo "snap-back" o "snap-through", el algoritmo de Newton-Raphson no puede dar una solución correcta. Esto es debido a que con este algoritmo las cargas exteriores se dividen en diferentes incrementos, haciendo que la carga siempre vaya en aumento, pero en el caso de algunas no linealidades las cargas exteriores deben de disminuir para seguir en equilibrio con las fuerzas internas del sólido.

Una estrategia para resolver este tipo de problemas es la adición de un factor de viscosidad volumétrico ficticio al modelo, el cual crea una fuerza de viscosidad en el modelo (1.12), que se añade a la ecuación de equilibrio (1.13).

$$F_v = cM^*v \qquad (1.12)$$

$$F^{ext} - F^{int} - F_v = 0 \qquad (1.13)$$

Donde c es el coeficiente de amortiguamiento, M^* es una matriz de masa artificial calculada a partir de aplicar una densidad unidad al modelo, $v = \Delta u/\Delta t$ es el vector de velocidades nodales. En la ecuación de equilibrio, F^{ext} representa las cargas externas aplicadas al modelo, F^{int} a las fuerzas interna del modelo y F_v a la fuerza debido a la viscosidad ficticia.

Mientras el modelo es estable, las fuerzas de viscosidad son despreciables. Pero en el momento en el que se produzca un incremento de desplazamientos muy grande (un incremento en v), la fuerza de viscosidad es importante e intenta estabilizar el modelo.

La elección de este factor de viscosidad se puede realizar mediante ensayo y error o indicando a Abaqus el porcentaje total de energía del modelo que se permite disipar debido a la inclusión de la fuerza de amortiguamiento.

1.4.3 Arc-length de Riks modificado

Este algoritmo permite obtener la solución a problemas no lineales estáticos, incluso en los casos en los que el comportamiento exhiba inestabilidades como las del tipo snap-back, snap-through, pandeo o colapso de alguna parte del modelo. En estos casos la curva fuerza-desplazamiento presenta alguna zona de pendiente negativa, es decir de rigidez negativa, y el sólido debe de liberar energía para alcanzar el equilibrio.

La filosofía de este algoritmo es hacer que tanto los grados de libertad del sólido en cada instante de tiempo como las cargas a las que se encuentra el mismo sean variables. Esto se consigue haciendo todas las cargas exteriores proporcionales a un factor λ. De esta manera, el número de variables que definen la solución del problema en este algoritmo son los n grados de libertad más el factor proporcional, es decir, $n+1$ variables.

Este algoritmo supone que la evolución de las variables es suficientemente suave, o dicho de otro modo, que en ningún momento se produce ninguna bifurcación ni pico en la evolución de las variables.

A continuación se desarrolla brevemente el significado geométrico del algoritmo, apoyándose para ello en la Figura 1-7, en la cual se muestra el esquema de un problema de una dimensión. El algoritmo se puede dividir en dos fases, la fase predictiva y la fase correctora. Durante la primera fase se determinará el tamaño de la longitud de arco en cada incremento, este se consigue trazando una tangente a la curva en el punto de equilibrio donde nos encontramos, dando un incremento al factor de carga llegando al punto A^1. Una vez alcanzada comienza la fase correctora, que en este algoritmo consiste en buscar la solución de equilibrio en una línea perpendicular a la trazada en la fase predictiva. Se aplica el algoritmo de Newton-Raphson y se fuerza que la solución se encuentre en dicha línea perpendicular.

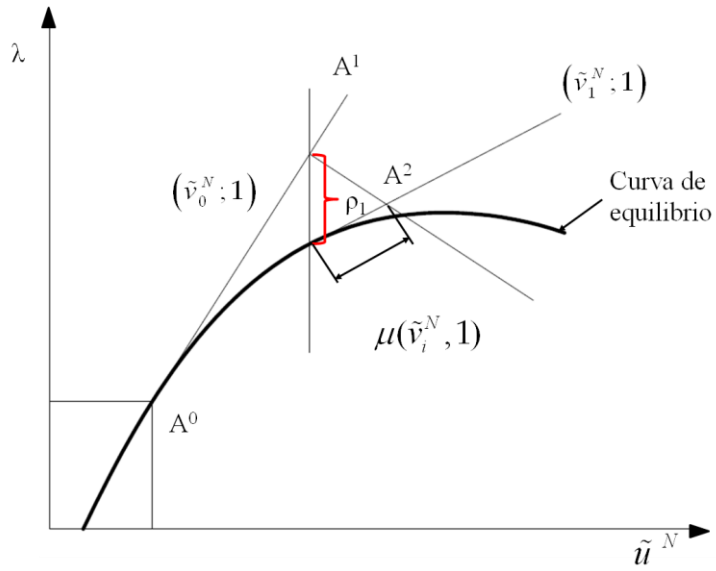


Figura 1-7 Esquema del algoritmo de Riks modificado

1.4.4 Sequential Linear Analysis

El presente algoritmo es desarrollado para su uso junto con la formulación interfase elástica lineal frágil (LEBIM). Esta formulación permite resolver problemas no lineales como un conjunto de diferentes problemas lineales.

La solución numérica del problema no lineal generalmente se basa en una aplicación gradual de las cargas y desplazamientos impuestos, por medio de un factor de carga, $0 \leq F \leq 1$. El procedimiento de solución es dado por una serie de estados lineales "pasos de carga". Al comienzo de cada paso de carga la zona actual de la interfase vinculada a la capa adhesiva es verificada, definiéndose el sistema de ecuaciones lineales del momento.

Al resolver este sistema de ecuaciones la correspondiente solución elástica es obtenida. Después, la solución del problema será dividida entre un número M (a priori desconocido) de pasos de carga donde todas las variables se comportan linealmente:

$$\phi(x,F) = F\Delta_m\phi(x) \qquad (1.14)$$

con $F_{m-1} \leq F \leq F_m, m = 1,\dots,M$ y $F_0 = 0$, y donde $\phi(x,F)$ es el valor de alguna variable del problema en un punto x después de que una fracción F de la carga es aplicada. $\Delta_m\phi(x)$ es el valor del incremento de la variable $\phi(x)$ con respecto a F, y es obtenido en la solución del sistema lineal de ecuaciones correspondiente al paso u_{m^n} de carga.

Esta solución cumple todas las condiciones de la formulación interfase elástica lineal (y además de la formulación de contacto sin fricción) hasta un cierto valor máximo F_m del factor de carga F asociado a este paso de carga. Un nuevo incremento del factor de carga conduce a la ruptura de algunos muelles (o a un cambio en las condiciones de contacto). Por consiguiente, valores de la variable ϕ al final de cada paso de carga son definidos como $\phi(x,F_m) = F_m\Delta_m\phi(x)$ para $F = 1,\dots,M$. Este procedimiento es ilustrado en la Figura 1-8.

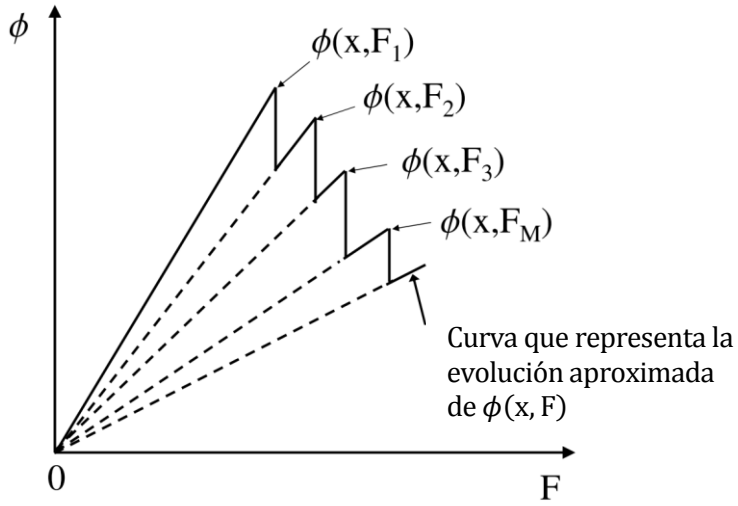


Figura 1-8 Ejemplo de la evolución de una variable $\phi(x,F)$ [22]

En la Figura 1-9 se muestra el pseudocódigo para este procedimiento en el caso de carga proporcional, definido por un valor de carga nominal multiplicado por un factor de carga sin dimensiones $F \geq 0$.

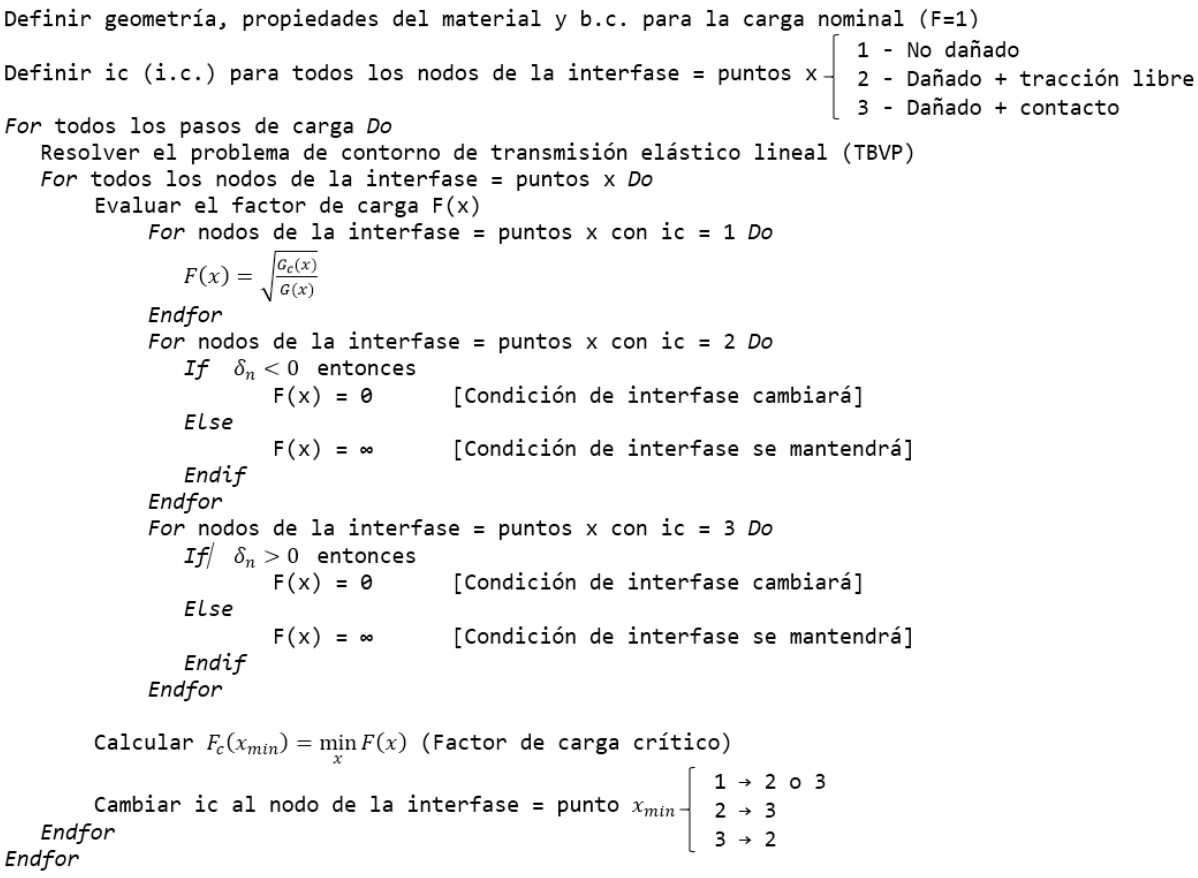


Figura 1-9 Procedimiento para SLA aplicado a un problema con interfase gobernado por el LEBIM, donde b.c. = condiciones de contorno e i.c. = condición de interfase [17]

1.5 Problemas de interfase en materiales compuestos

Algunos de los problemas estudiados previamente por el GERM y que pueden estudiarse usando la presente herramienta son:

- Análisis de una grieta en una capa adhesiva delgada en una probeta DCB.
- Grietas en delaminación de laminados simétricos.
- Test de fragmentación de una única fibra
- Comportamiento micro-mecánica de grietas entre la matriz y la fibra bajo cargas transversales.

1.6 Objetivos y Organización

Los objetivos que pretende alcanzar el presente trabajo son:

- Utilizando el modelo LEBIM en el código de elementos finitos Abaqus y usando las soluciones elásticas que este proporciona, implementar por medio de una programación en Python, el método de solución secuencial-lineal, "Sequential Linear Analysis", el cual permite captar las inestabilidades que aparecen en muchos problemas de materiales compuestos.
- Testear esta herramienta resolviendo un problema conocido: modelo del ensayo de tenacidad a la fractura interlaminar en materiales compuestos en diferentes situaciones de análisis que la herramienta ofrece.

El trabajo se organiza de la siguiente manera, en el capítulo 2 se desarrollan los conocimientos previos que el lector debe tener para entender la estructura interna del código SLA, expuesto en el capítulo 3. En este último capítulo se explica el funcionamiento global y el funcionamiento paso a paso (significado de cada línea del código fuente) de dicho código. En el capítulo 4 se explican los pasos a seguir para el correcto uso de la herramienta y finalmente en el capítulo 5 se presentan los resultados obtenidos con esta herramienta al resolver el problema: modelo de un ensayo de tenacidad interlaminar en materiales compuestos.

2 PROGRAMACIÓN EN PYTHON

En este capítulo se abordarán los conocimientos básicos necesarios sobre la programación en Python que serán necesarios para el correcto entendimiento del código fuente mostrado en el siguiente capítulo, el cual implementa el método de solución secuencial lineal (SLA).

2.1 Módulos y sus métodos

Python lleva integrado un pequeño conjunto de métodos (funciones) enormemente útiles. Aquellos utilizados en el código SLA se listan en la Tabla 2-1, incluyendo una pequeña descripción de cada uno de ellos. Para profundizar se recomienda acudir a la documentación facilitada por Python en la web [20].

Tabla 2-1 Métodos incorporados en Python y empleados en el código

Sintaxis	Descripción
append(obj)	Adjunta el objeto 'obj' a una lista
close()	Cierra un archivo
float()	Convierte en número flotante la variable incluida
int()	Convierte en entero la variable incluida
len(lista)	Devuelve el número de elementos de la lista incluida
open("name.txt",'par')	Manipula archivo de texto plano. Si el parámetro 'par' incluido es: 'r': lee los datos que hay en el archivo 'w': crea un archivo para escribir en él 'a': abre el archivo para posteriormente añadir contenido al final de este
range(inicio, fin, paso)	Devuelve una lista de valores enteros, la cual comienza por el valor incluido como parámetro 'inicio' (en caso de omitirlo comienza en cero) y finaliza con el valor anterior al dado como parámetro 'fin'. En caso de indicar el parámetro 'paso' la lista de valores aumentará según dicho entero.
raw_input()	Lee una línea de la entrada estándar y la devuelve como una cadena
sorted()	Ordena los elementos de una lista
str()	Convierte en una cadena la variable incluida
write()	Escribe en el archivo de texto plano

Todos los demás métodos están repartidos en módulos. Esto es una decisión consciente de diseño, para que el núcleo del lenguaje no se llene como en otros lenguajes de scripts.

Por tanto, los módulos son grupos de métodos y variables alojados dentro de un archivo .py, los cuales facilitan el desarrollo de los scripts a la hora de solucionar un problema. Python provee de un gran abanico de módulos que integran su librería estándar [20] y también existe la posibilidad de crear e incorporar otros nuevos. En el presente trabajo no se aborda la creación de nuevos módulos, sino que se realiza una breve reseña de aquellos que se utilizan posteriormente en el código.

Para poder hacer uso de dichos métodos en los scripts, se puede importar el módulo que contenga el método deseado al completo o simplemente importar el método en concreto que se desee.

2.1.1 Importación de módulos

Es una buena costumbre pero no obligatorio ubicar todas las declaraciones de importación al principio del script, existiendo dos posibilidades para ello, a través de la palabra clave "import" o a través de las palabras claves "from…import".

2.1.1.1 Import

Es la manera más simple y cómoda, ya que importa todos los métodos existentes dentro del módulo indicado.

Sintaxis 2-1. *import* módulo1, módulo2,…móduloN

Si los módulos son importados utilizando esta sintaxis, la llamada de los métodos dentro del código fuente se debe realizar de la siguiente manera: nombre del módulo, seguido de un punto, seguido del nombre de la función a usar.

Ejemplo 2-1. *Importar los módulos os y sys. Usa el método chdir del módulo "os" para cambiar el directorio de trabajo a C:\ejemplos_python*

```
import os, sys
os.chdir('C:\ejemplos_python')
```

2.1.1.2 From…import

Importará única y exclusivamente el método pasado como parámetro. Es muy recomendable su uso ya que ahorra tiempo de procesamiento y recursos de la máquina en cuestión.

Sintaxis 2-2. *from* módulo *import* método1, método2, …métodoN

El uso de esta sintaxis conlleva que a la hora de llamar al método no sea necesario incorporar en la instrucción el nombre del módulo, como anteriormente si sucedía, sino que directamente se indica el nombre del método que se desea utilizar.

Ejemplo 2-2. *Importar los métodos chdir y remove. Usar método chdir para cambiar el directorio de trabajo a C:\ejemplos_python*

```
from os import chdir, remove
chdir('C:\ejemplos_python')
```

2.1.2 Módulos y métodos empleados en el código

A continuación se muestra una breve reseña de aquellos módulos y métodos que aparecen posteriormente en el código SLA, la mayoría de estos módulos se encuentran alojados dentro de la librería estándar de Python y uno de ellos es incorporado al instalar Abaqus en el ordenador.

A lo largo de este subapartado aparece la palabra "path" en varias ocasiones, refiriéndose con ello a la ruta absoluta de un archivo o directorio, es decir, a su localización exacta en el ordenador.

2.1.2.1 Módulo os

Este módulo nos permite acceder a funcionalidades dependientes del sistema operativo, sobre todo aquellas que refieren información sobre el entorno del mismo y permiten manipular la estructura de directorios. Integra el submódulo path (os.path), el cual permite acceder a ciertas funcionalidades relacionadas con los nombres de las rutas de archivos y directorios.

Entre todos los métodos alojados en este módulo, los utilizados en el código se recogen en la Tabla 2-2, en la cual se especifica la sintaxis y descripción de cada método.

Tabla 2-2 Métodos del módulo os

Sintaxis	Descripción
chdir(path)	Cambiar de directorio de trabajo
mkdir(path)	Crear directorio
path.exists(path)	Saber si un archivo existe
remove(path)	Eliminar un archivo

2.1.2.2 Módulo sys

Este módulo es el encargado de proveer variables y funcionalidades directamente relacionadas con el intérprete. En la Tabla 2-3 se muestra la única variable usada de este módulo en el código SLA.

Tabla 2-3 Variable del módulo sys

Sintaxis	Descripción
argv	Crea una lista con todos los argumentos pasados por la línea de comandos. Al ejecutar en CMD: python modulo.py arg1 arg2, retornará una lista: ['modulo.py', 'arg1', 'arg2']

2.1.2.3 Módulo Subprocess

El módulo subprocess permite trabajar de forma directa con órdenes del sistema operativo. Entre los métodos más comunes de este módulo, se encuentra subprocess.call(). El primer argumento de este método call es el comando a ser ejecutado.

El Ejemplo 2-3 muestra cómo se ejecuta la orden sencilla de limpiar la pantalla a través de este método.

Ejemplo 2-3. *Limpiar la pantalla*

```
from subprocess import call
call('cls')
```

El método call es muy importante en este TFG ya que permite llamar a otros programas desde el script de Python. Este punto se aborda en el apartado 2.2 de este mismo capítulo.

2.1.2.4 Módulo shutil

Este módulo ofrece una serie de operaciones de alto nivel sobre los archivos y carpetas. En particular, se proporcionan funciones que soportan la copia, traslado y eliminación de archivos y directorios. Para operaciones de un conjunto de archivos se suele usar este módulo mientras que para operaciones en archivos individuales el módulo os.

En la Tabla 2-4 se recogen los métodos utilizados en el código SLA pertenecientes a este módulo.

Tabla 2-4 Métodos del módulo shutil

Sintaxis	Descripción
move(src, dst)	Mover un archivo o directorio (src) a otra ubicación (dst)
rmtree(path)	Eliminar un directorio y todo su contenido

2.1.2.5 Módulo glob

El módulo glob integra un método, también llamado glob, el cual crea una lista con las rutas de los archivos cuyos nombres coinciden con el patrón dado como parámetro.

Tabla 2-5 Método del módulo glob

Sintaxis	Descripción
glob(pathmodel)	Busca ficheros que verifiquen el patrón (pathmodel), creando una lista con sus rutas

2.1.2.6 Módulo Operator

De este módulo interesa el método itemgetter, el cual es usado junto con el método sorted(). Permite ordenar una lista de tuplas (o de registros) por el índice de uno de sus campos (el índice del primer campo es el 0, del segundo el 1, etc.).

Ejemplo 2-4. *Ordenar lista en función del segundo campo*

```
lista = [('cccc',4444),('d',1),('aa',22),('bbb',333)]
listaord = sorted(lista, key=itemgetter(1))
print('lista ordenada por campo2:', listaord)
#Por pantalla: lista ordenada por campo2: [('d',1),('aa',22),('bbb',333),('cccc',4444)]
```

2.1.2.7 Módulo OdbAccess

Este módulo solo será accesible cuando se haya instalado el software Abaqus en el ordenador. Entre sus métodos se encuentra "openOdb()", el cual permite abrir los archivos ODB para su posterior exploración. En el subapartado 2.3.1 se profundiza en estos tipos de archivos.

Tabla 2-6 Método del módulo OdbAccess

Sintaxis	Descripción
openOdb(namefile.odb)	Crea un objeto Odb con los datos del fichero incluido como argumento. El archivo debe estar alojado en el directorio de trabajo.

2.2 Llamada a otros programas

A lo largo del código SLA se llevan a cabo dos llamadas a otros programas, una primera al software Abaqus para que resuelva el problema enviado junto con la subrutina UMAT y otra al mismo Python para que postprocese a través de otro script los datos obtenidos del archivo ODB creado por Abaqus con los resultados del problema. Para realizar estas llamadas será necesario el uso del módulo subprocess, el cual permite trabajar de forma directa con órdenes del sistema operativo.

2.2.1 Llamada a Abaqus. Resolver archivo inp + UMAT (archivo.for)

A través de la consola del sistema esta llamada se realiza de la siguiente manera:

Instrucción CMD 2-1. *abaqus job*=nombre_arc.salida *input*=nombre_inp *user*=nombre_UMAT

Por tanto, para poder ejecutar esta instrucción desde el script en Python se usa el método call del módulo subprocess, quedando la instrucción como se muestra a continuación.

Instrucción Python 2-1.

subprocess.call(*abaqus job*=nombre_arc.salida *input*=nombre_inp *user*=nombre_UMAT, shell=True*)*

Como primer parámetro de este método se incluye el comando a ejecutar y como segundo parámetro "Shell=True". Con este último parámetro se consigue ejecutar el comando a través del Shell.

2.2.2 Llamada a otro script

A través de la consola del sistema esta llamada se realiza de la siguiente manera:

Instrucción CMD 2-2. *python nombre_ach.postproceso arg1 arg2 …*

Al igual que antes, en Python quedaría de la siguiente manera:

Instrucción Python 2-2. subprocess.call(*python nombre_arc.postproceso arg1, arg2,…,* shell=True*)*

Si el script llamado tiene como objetivo la apertura y exploración de archivos ODB, entonces:

Instrucción Python 2-3. subprocess.call(abaqus *python nombre_arc.postproc. arg1, arg2,…,*shell=True*)*

2.3 Postproceso

2.3.1 Extraer información del archivo ODB. Repositorios

Una vez definido los parámetros del modelo y finalizada la simulación del estado de cargas supuesto, Abaqus genera un archivo de base de datos de salida, Output database (ODB), con extensión .odb, en el cuál se aloja toda la información pertinente al análisis efectuado.

La apertura de este tipo de archivos se lleva a cabo a través de la Instrucción Python 2-4, la cual crea un objeto Odb que puede ser almacenado en una variable, utilizando esta última para la lectura y extracción de los datos deseados.

Instrucción Python 2-4. odb=openOdb(name.odb)

A través de la Figura 2-1 el lector puede hacerse una idea general de la estructura interna de este tipo de archivos. Por un lado se almacenan los datos del modelo y por otro los resultados. Se ha creado un script, mostrado en el Anexo D, el cual permite navegar a través del archivo ODB con el objetivo de facilitar el conocimiento de la estructura de este y localizar la ubicación exacta de la información deseada.

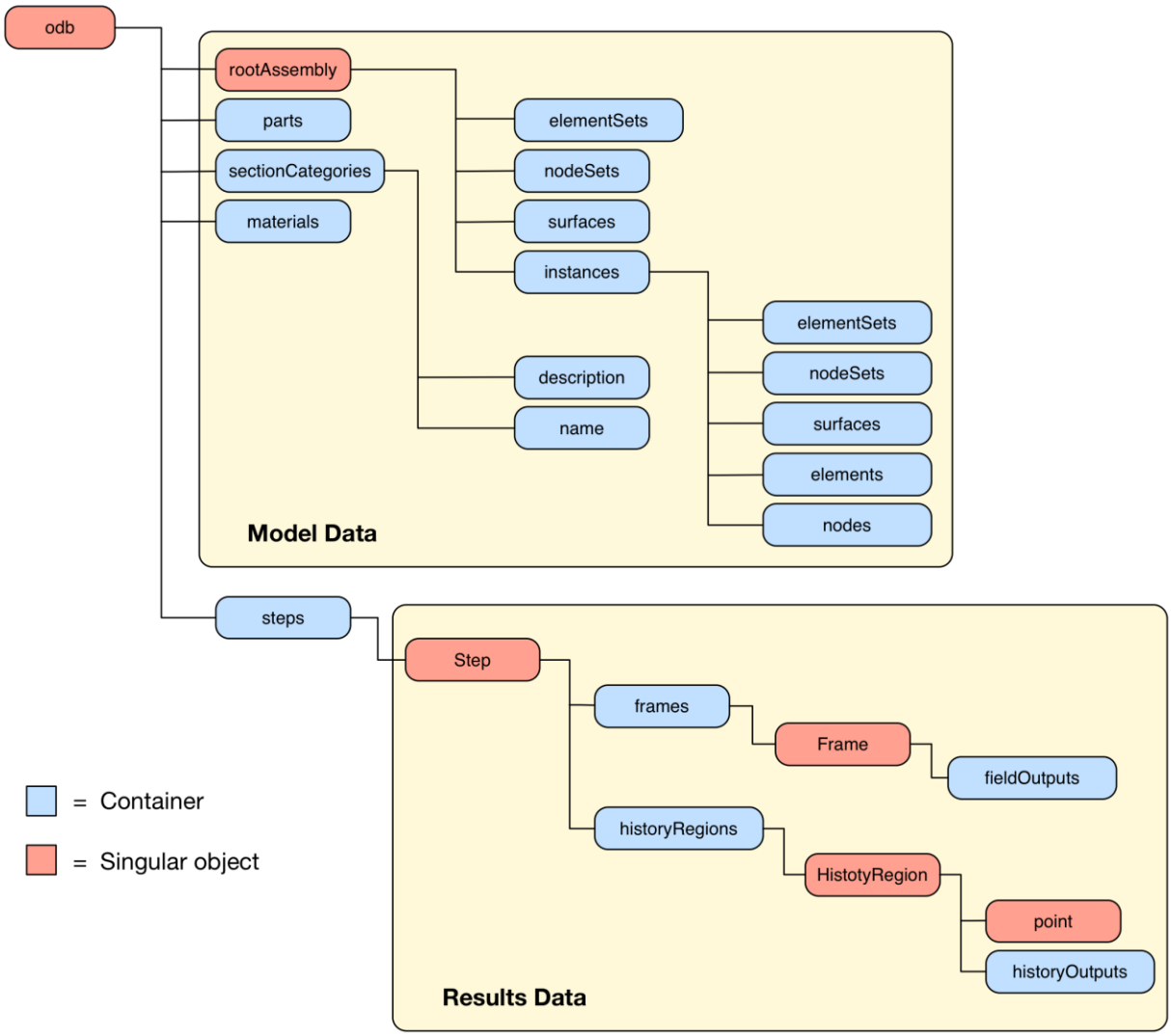


Figura 2-1 Esquema simplificado de la estructura interna de un archivo ODB

Estos archivos están compuestos por un nuevo tipo de dato creado por los desarrolladores de Abaqus llamado repositorio. Estos son contenedores que almacenan un tipo particular de objeto. Por ejemplo, el repositorio 'steps' contiene todos los steps definidos en el modelo. Un repositorio es similar a un diccionario de Python. Sin embargo, sólo un programador puede añadir un objeto a un repositorio y todos ellos son del mismo tipo.

Para acceder a un objeto en una lista, se proporciona el índice entero que especifica la posición del objeto en ella. Por el contrario, se accede a un objeto en un diccionario/repositorio a través de su clave, que puede ser una cadena, un número entero, o cualquier tipo de objeto Python. No hay un orden implícito de las claves en un repositorio. En la mayoría de los casos se asigna una cadena como clave. Esta se convierte entonces en una forma más intuitiva para acceder a los elementos en un diccionario/repositorio.

Como se comentó al principio, el repositorio 'steps' contiene todos los steps definidos en el modelo. Para acceder a un objeto particular se utiliza corchetes y la clave de dicho objeto entre comillas. Por ejemplo, con la línea de código 'odb.steps['Step-1']' se accede al step (objeto particular) llamado Step-1 almacenado en el repositorio 'steps'. Dentro de este step concreto se encuentran alojados otros repositorios como puede ser 'frames', el cual contiene un objeto particular por cada incremento de carga en el step. A este repositorio se accede con la siguiente línea de código 'odb.steps['Step-1'].frames'.

Para reforzar lo explicado anteriormente e introducir el método keys() y otro repositorio llamado fieldOutputs se muestran a continuación dos ejemplos de extracción de datos del archivo ODB.

Ejemplo 2-5. *Abrir archivo "placas.odb", extraer nombre del step y almacenarlo en la variable key_step*

```
odb=openOdb(placas.odb)

key_step=odb.steps.keys()
```

Nota. El objeto Odb es almacenado en la variable 'odb'. Por ello se empieza con la cadena 'odb.' para acceder a los datos de este. El método keys() crea una lista con todas las claves del repositorio, en este caso del repositorio 'steps'.

Ejemplo 2-6. *En el último incremento de carga del step llamado 'Step-1', extraer el valor de la tensión (dirección 1) en el punto de integración 27 (global)*

```
odb.steps['Step-1']].frames[-1].fieldOutputs['S'].values[26].data[0]
```

Nota. El repositorio fieldOutputs contiene aquellos objetos que a la hora de realizar el modelo en Abaqus se hayan indicado. Algunos de ellos son: 'S' (tensiones), 'E' (deformaciones), 'U' (desplazamientos), 'RF' (reacciones) y SDV (variables de estado dependiente). Dentro de este repositorio se encuentra el objeto 'values', el cual por cada punto evaluado contiene diferentes objetos con información acerca de ellos.

2.3.2 Screenshots

Un screenshot es una imagen tomada para registrar los elementos visibles que aparecen en la pantalla. Dicho de otra manera, una captura de pantalla.

Al abrir un archivo odb se ejecuta Abaqus en modo visualización, mostrándose los resultados del análisis. Parece interesante almacenar estas imágenes con los resultados para fututos usos.

Para facilitar esta tarea y que se realice automáticamente se han desarrollado dos scripts llamados ‘screenshot_ejecutar.py’ y ‘screenshot_editar.py’ los cuales se detallan en el Anexo E.

Un ejemplo de screenshot obtenido a partir de los scripts citados anteriormente es el mostrado en la Figura 2-2, el cual pertenece a los resultados obtenidos (variable ‘S’, componente ‘S22’) del modelo del ensayo de tenacidad interlaminar en materiales compuestos.

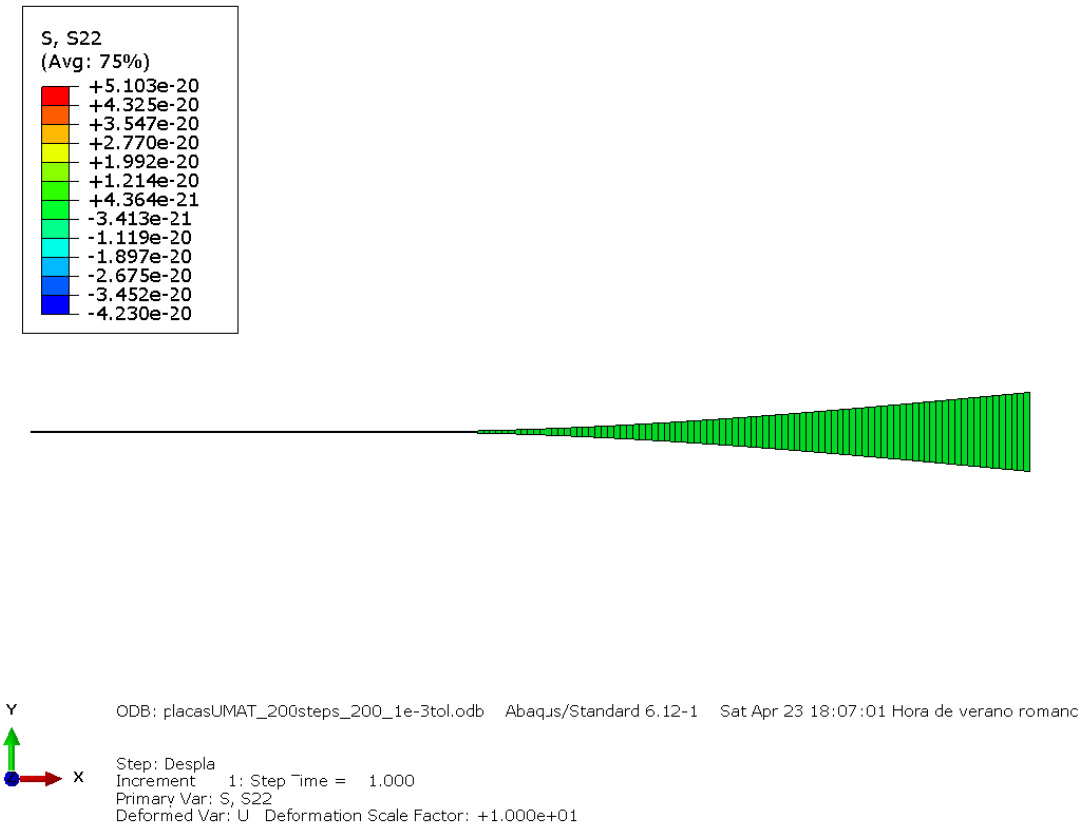


Figura 2-2 Screenshots

3 CÓDIGO SLA: ABAQUS + FORTRAN + PYTHON

El eje central del presente trabajo se basa en la implementación del método numérico de solución (SLA) en el programa de elementos finitos Abaqus para su uso con el modelo de interfase elástica lineal frágil, es decir, en el desarrollo de una herramienta computacional capaz de:

- Enviar a Abaqus el archivo INP junto a la subrutina de usuario (UMAT) para su resolución
- Extraer datos del archivo ODB
- Calcular el(los) punto(s) de integración con factor(es) máximo(s)
- "Dañar" dicho(s) punto(s) de integración
- Repetir estos pasos tantas veces como el usuario desee

Para la implementación del método se han desarrollado dos scripts en el lenguaje de programación Python, el primero llamado "main.py" es la "columna vertebral" del código SLA, en el cual se aloja el bucle y se encarga de realizar las llamadas a los diferentes programas y el otro llamado "readODB.py" es el encargado del postprocesado de los datos de interés alojados en el archivo ODB. Ambos scripts se recogen en el Anexo A.

Por tanto, se parte inicialmente de cuatro archivos, los dos scripts en Python que implementan el método de solución, un archivo .inp y otro .for. El archivo .inp contiene los datos del modelo y de la historia y este debe ser desarrollado por el usuario a través del software Abaqus teniendo en cuenta las anotaciones recogidas en el Anexo C. El archivo .for, como su extensión indica, está programado en Fortran y fue desarrollado por D. Castillo [6] para implementar el LEBIM. Aunque el grueso del código permanece inalterado, ha sufrido una serie de modificaciones para poderlo aplicar a este trabajo, las cuales se recogen en el Anexo B.

3.1 Funcionamiento general

Antes de entrar a desgranar con profundidad toda la estructura del código SLA, sería conveniente tener una idea general de la función que este va a desempeñar. Para ello se modela el código SLA como una caja, Figura 3-1, en la cual entran dos archivos: el modelo a solucionar (archivo.inp) y el modelo de fractura, en nuestro caso LEBIM (archivo.for), obteniéndose n archivos odb (uno por cada iteración) y dos archivos txt, uno de ellos que recoge la evolución de la rotura (información de los puntos de integración con factores máximos de cada iteración) y el otro que almacena los datos necesarios (reacción, factor y desplazamiento en cada iteración) para que posteriormente el usuario pueda graficar la curva fuerza-desplazamiento del análisis.

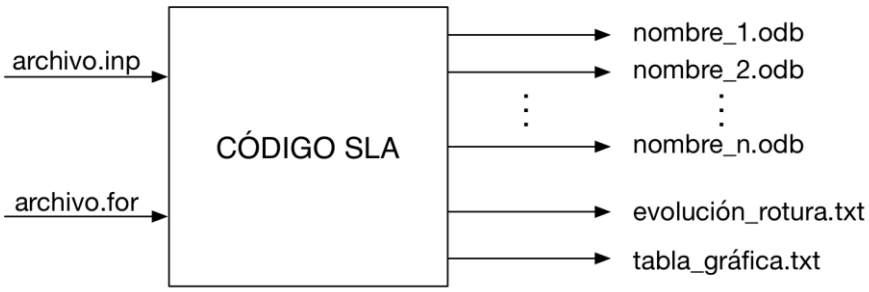


Figura 3-1 Código SLA. Entradas y salidas de archivos

Una visión general del funcionamiento del código se muestra en la Figura 3-2, en la cual se ilustra un diagrama de flujo con las principales tareas que el código va a realizar.

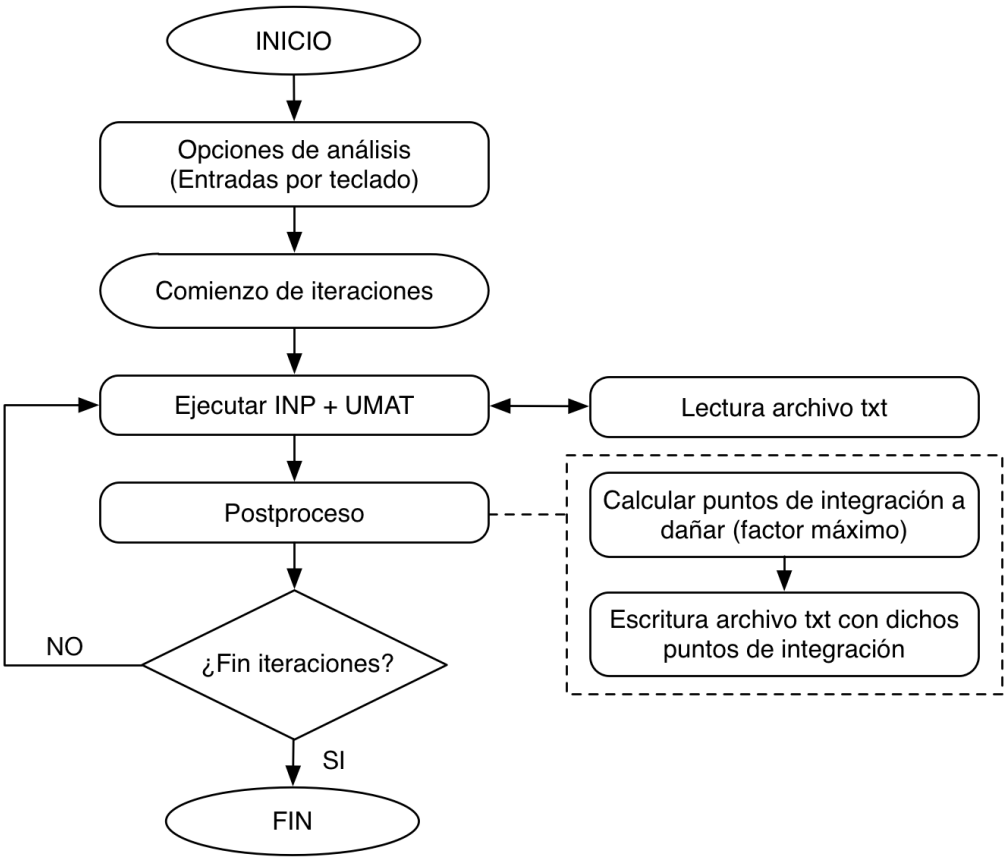


Figura 3-2 Diagrama de flujo del código SLA. Funcionamiento general

3.2 Funcionamiento paso a paso

Una vez comprendido el funcionamiento general del código, se pasa al desglose del mismo. Primero se realiza con el script "main.py" y acto seguido con el script "readOdb.py". Para exponer la programación de los mismos de una forma sencilla y clara se siguen los siguientes pasos:

- Listado de las variables empleadas en el script junto con sus descripciones
- Diagrama de flujo del funcionamiento del script
- Comentarios de cada bloque del diagrama con su correspondiente código fuente

3.2.1 Script "main.py"

En este apartado se listan en primer lugar las variables utilizadas en el desarrollo del script junto con sus descripciones en la Tabla 3-1, a continuación a través del diagrama de flujo de la Figura 3-3 se representa el funcionamiento paso a paso del script y finalmente se comenta cada bloque del diagrama en una secuencia de subapartados.

Tabla 3-1 Variables empleadas en "main.py"

Variable	Descripción
files	Variable de control bucle for. Recibe patrón de los archivos a borrar o mover
files_delete	Lista con las rutas de los archivos a borrar cuyos nombres o extensiones han coincidido con el patrón pasado por la variable files
files_move	Lista con las rutas de los archivos a mover cuyos nombres o extensiones han coincidido con el patrón pasado por la variable files
folder_name	Nombre del directorio donde se almacenan todos los archivos creados durante el análisis
k	Variable de control bucle for. Recibe el número de archivos a borrar
list_delete	Tupla que contiene todos los patrones de los archivos a borrar
list_move	Tupla que contiene todos los patrones de los archivos a mover
name_files	Nombre archivo de salida (ODB) en cada iteración
name_INP	Nombre sin extensión del archivo inp

name_TXT	Raíz del nombre del archivo txt que contiene datos de la evolución de la rotura
name_txt_evolucion	Nombre completo del archivo txt que contiene datos de la evolución de la rotura
name_UMAT	Nombre sin extensión del archivo .for que contiene la subrutina de usuario
post_ODB	Nombre del script con extensión (.py) que contiene el código de postproceso
num_iteraciones	Número de iteraciones a realizar en el análisis
paso	Paso para guardar archivos odb
respuesta_modo	Contiene la opción deseada por el usuario para realizar el análisis
respuesta_ODB	Contiene la opción deseada por el usuario para guardar todos los odb de cada iteración o solo algunos en concreto
step	Variable de control bucle for (bucle iteraciones)
tol_o_ptosint	Valor de la tolerancia o número de puntos de integración a dañar por iteración
working_directory	Ruta directorio de trabajo

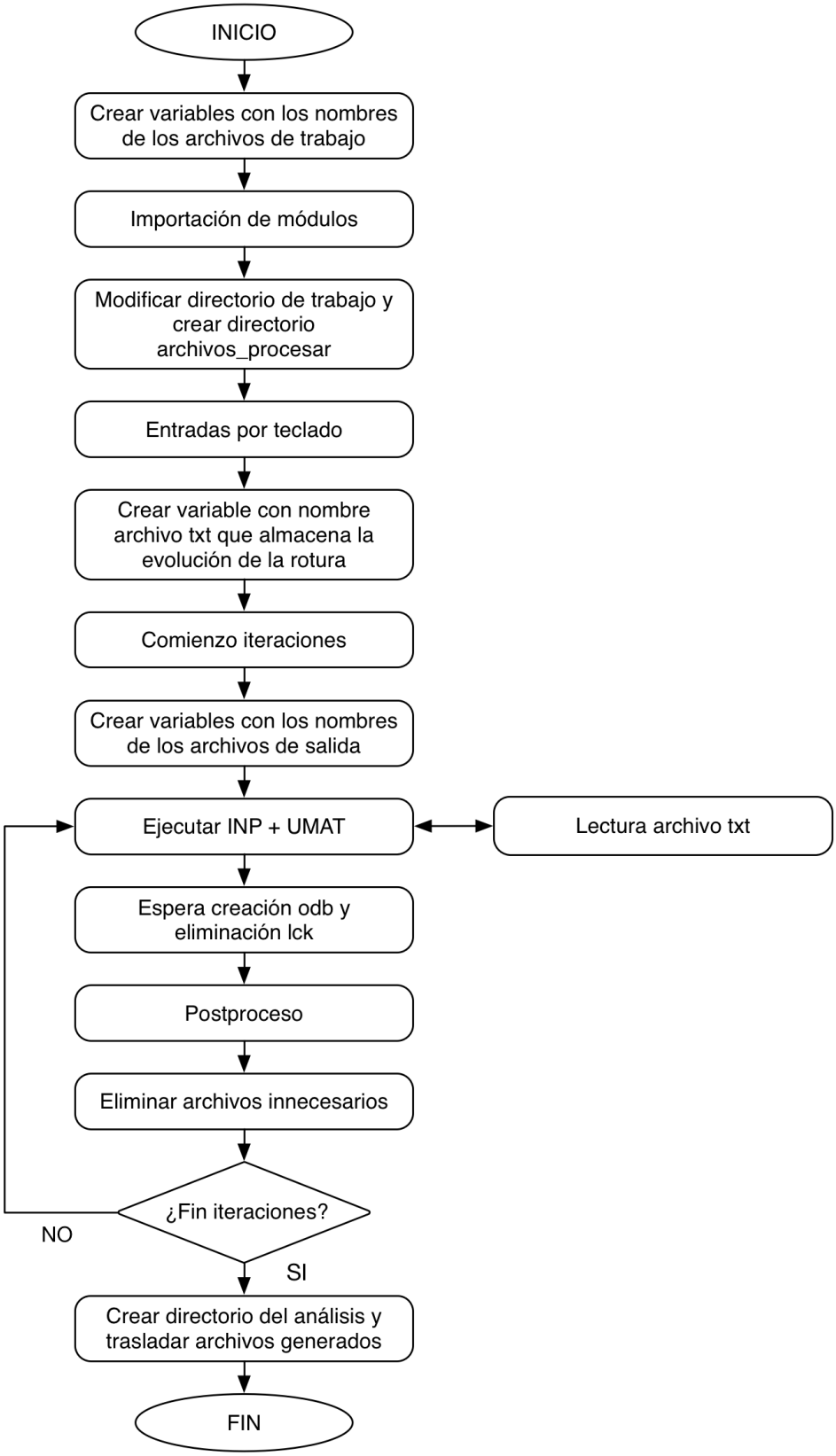


Figura 3-3 Diagrama de flujo script "main.py". Funcionamiento paso a paso

3.2.1.1 Asignación variables. Nombres archivos de entrada

A lo largo de la ejecución del código se llaman a tres archivos: archivo.inp (modelo), archivo.for (comportamiento material) y archivo.py (postproceso). Se ha visto conveniente introducir al comienzo del código la asignación de los nombres de estos archivos a variables, de manera que a través de la modificación de ellos resulte más cómodo para el usuario la realización de análisis con diferentes modelos, comportamientos o postprocesos.

Estos tres archivos deben de estar alojados en el mismo directorio, el directorio de trabajo, cuya ruta es dada a la variable working_directory.

Código 3-1. Asignación variables. Nombre archivos de entrada

```
10  name_INP = 'placasUMAT'
11  name_UMAT = 'lebim'
12  post_ODB = 'readOdb_mejorado' + '.py'
13  working_directory = 'C:\scriptsabaqus\placas_UMAT'
```

3.2.1.2 Importación módulos

Como se explicó en el apartado 2.1, para poder utilizar las herramientas (métodos) que Python facilita al desarrollador, debemos de importar los módulos en los cuales se alojan los métodos deseados. Esto es lo que se recoge en las líneas de código [17-21].

Código 3-2. Importación módulos

```
17  from os import chdir, path, remove, mkdir, system
18  from shutil import move, rmtree
19  from glob import glob
20  from subprocess import call
21  from time import time
```

3.2.1.3 Designación directorio de trabajo y creación directorio archivos_procesar

Los tres archivos de entrada, como se comentó anteriormente deben de estar alojados en el mismo directorio, el directorio de trabajo. Por tanto, en la línea 24 se modifica el directorio de trabajo predeterminado por el dado anteriormente a la variable working_directory. Tras este paso, todos los archivos creados durante el proceso serán almacenados en este directorio, salvo el archivo txt llamado "datos_procesar", el cual almacena los puntos de integración dañados en cada iteración y el elemento (global) al que pertenecen. Este archivo se crea a propósito en una carpeta situada en la raíz llamada archivo_procesar, la cual es creada en la línea 25 del código.

Pasamos a continuación a justificar la creación de dicha carpeta. Durante la ejecución del archivo.inp junto con la UMAT se produce la lectura de un archivo txt (Figura 3-3). Este archivo es el comentado en el párrafo anterior (datos_procesar.txt). Como se puede comprobar en el Anexo B, el código fuente de la UMAT contiene unas líneas en las cuales se manda leer este archivo. Para evitar que el usuario tenga que modificar también este archivo cada vez que cambie de directorio de trabajo, se ha decidido que este archivo txt se aloje en una carpeta genérica (archivo_procesar), diferente a donde el usuario aloja los archivos de entrada (directorio de trabajo).

Código 3-3. Designación del directorio de trabajo y creación directorio archivo_procesar

```
24    chdir(working_directory)
25    mkdir('C:\\archivo_procesar')
```

3.2.1.4 Entradas por teclado

El objetivo de estas líneas de código [32-46] es ofrecer al usuario la posibilidad de configurar su análisis. Las diferentes opciones se muestran por pantalla y el usuario debe ir introduciendo por teclado las particularidades de su análisis.

Las opciones presentadas al usuario son las siguientes:

- Número de iteraciones del análisis (línea 32)
- Si se desea guardar los ODB de cada iteración o solo algunos en concreto (líneas 35-37) (esto último se implementa con una variable paso (líneas 38-39))
- Modo de análisis: tolerancia o puntos de integración a dañar en cada iteración (líneas 42-46)

Código 3-4. Entradas por teclado

```
29    call('cls',shell=True)
31    # Pedir numero de iteraciones
32    num_iteraciones = input("Numero de iteraciones: ")
34    #Preguntar si se guardan todos los ODB y en caso afirmativo el paso
35    respuesta_ODB = raw_input("Ir guardando ODB (y/n): ")
36    while (respuesta_ODB != 'y') and (respuesta_ODB != 'n'):
37        respuesta_ODB = raw_input("Ir guardando ODB (y/n): ")
38    if (respuesta_ODB=='y'): paso = input("Paso: ")
39    else: paso=0
41    #Preguntar si queremos aplicar tolerancia o numero de ptos int a danar
42    respuesta_MODO = input("Aplicar:\n\t1-Tolerancia\n\t
                                  2-Numero de ptos int a danar\nRespuesta (numero): ")
43    while (respuesta_MODO != 1) and (respuesta_MODO != 2):
44        respuesta_MODO= input("Aplicar:\n\t1-Tolerancia\n\t
                                  2-Numero de ptos int a danar\nRespuesta (numero): ")
45    if (respuesta_MODO==1): tol_o_ptosint = raw_input("Tolerancia: ")
46    else: tol_o_ptosint = raw_input("Numero ptos de int a danar: ")
```

3.2.1.5 Asignación variable. Nombre archivo txt evolución de la rotura

Una vez el usuario haya configurado su análisis, se da nombre al archivo txt que contendrá los datos referentes a la evolución de rotura almacenándolo en la variable "name_txt_evolucion". Según el modo de análisis elegido, la estructura del nombre será:

Tabla 3-2 Estructura nombre archivo txt evolución de la rotura

TOLERANCIA (línea 50)	
Nombre	'evolucion_rotura' + número de iteraciones total + iteración actual + valor tolerancia
Ejemplo	evolucion_rotura_200steps_1e-3tolerancia
PUNTOS DE INTEGRACIÓN A DAÑAR (línea 51)	
Nombre	'evolucion_rotura' + número de iteraciones total + iteración actual + ptos int. dañar
Ejemplo	evolucion_rotura_25steps_8ptos_danar

Código 3-5. Asignación variable. Nombre archivo txt evolución de la rotura

```
50      if (float(tol_o_ptosint) < 1):
            name_txt_evolucion = 'evolucion_rotura' + '_' + str(num_iteraciones) + 'steps' + '_'
                                 + tol_o_ptosint + 'tolerancia'
51      else:
            name_txt_evolucion = 'evolucion_rotura' + '_' + str(num_iteraciones) + 'steps' + '_'
                                 + tol_o_ptosint + 'ptos_danar'
```

3.2.1.6 Asignación variable. Nombre archivo de salida Abaqus

Primer paso del bloque iteraciones. Lo primero que se hace es asignar a una variable (name_files) el nombre que tendrá el archivo de salida generado por Abaqus en la iteración. Dependiendo del modo de análisis elegido, el nombre consta de:

Tabla 3-3 Estructura nombre archivo salida ODB

TOLERANCIA (línea 56)	
Nombre	nombre del archivo inp + número de iteraciones total + iteración actual + valor tolerancia
Ejemplo	placasUMAT_200steps_1_1e-3tol
PUNTOS DE INTEGRACIÓN A DAÑAR (línea 57)	
Nombre	nombre del archivo inp + número de iteraciones total + iteración actual + ptos int. dañar
Ejemplo	placasUMAT_25steps_7_8ptos_danar

Código 3-6. Asignación variable. Nombre archivos de salida

```
56    if (float(tol_o_ptosint) < 1):
          name_files = name_INP + '_' + str(num_iteraciones) + 'steps' + '_' + str(step + 1) +
                       '_' + tol_o_ptosint + 'tol'
57    else:
          name_files = name_INP + '_' + str(num_iteraciones) + 'steps' + '_' + str(step + 1) +
                       '_' + tol_o_ptosint + 'ptos_danar'
```

3.2.1.7 Ejecutar INP + UMAT

Esta línea de código simplemente es la particularización de la Instrucción Python 1-1 del apartado 2.2. Con ella se llama a Abaqus para resolver el modelo (archivo.inp cuyo nombre está almacenado en la variable "name_INP") junto con la subrutina de usuario (archivo.for cuyo nombre es almacenado en la variable "name_UMAT").

Código 3-7. Ejecutar INP + UMAT

```
59    call('abaqus Job=' + name_files + ' ' + 'input=' + name_INP + ' ' + 'user=' + name_UMAT,shell=True)
```

3.2.1.8 Tiempo de espera. Creación odb y eliminación lck

Surge la necesidad tras la ejecución anterior de esperar la creación del archivo.odb y de la eliminación del archivo.lck, ya que sin cumplirse esta condición no es posible seguir ejecutando la siguiente línea del código. Para satisfacer esta condición se añade este bucle while.

Código 3-8. Tiempo de espera. Creación odb y eliminación lck

```
62    while (path.exists(name_files + '.odb') == False) or (path.exists(name_files + '.lck') == True):  pass
```

3.2.1.9 Postproceso

Esta línea de código simplemente es la particularización de la Instrucción Python 1-3 del apartado 2.2, la cual llama a otro script. En este caso se llama al script "readOdb.py" cuyo nombre se almacena en la variable "post_ODB". Se pasan las siguientes variables como argumentos: "name_files", "tol_o_ptosint", "name_txt_evolucion", "str(step)" y "working_directory", las cuales están descritas en la Tabla 3-1.

Código 3-9. Postproceso

```
65    call('abaqus python' + ' ' + post_ODB + ' ' + name_files + ' ' + tol_o_ptosint + ' ' + name_txt_evolucion + ' ' +
      str(step) + ' ' + working_directory, shell=True)
```

3.2.1.10 Eliminación archivos innecesarios

En cada iteración Abaqus genera una serie de archivos, de los cuales es de interés el archivo odb. Eliminar todos esos archivos que no interesan es de lo que se encargan las líneas 73-77.

Como se vio en el Código 3-4 existe la posibilidad de guardar el archivo odb generado en cada iteración o según una variable paso. De esta tarea se encargan las líneas 69 y 70.

Código 3-10. Eliminación archivos innecesarios

```
68  # Borrado ODB cumpliendo las condiciones dadas
69  if (respuesta_ODB=='y') and (((step+1)%paso)!=0): remove(name_files + '.odb')
70  elif(respuesta_ODB=='n')and((step+1)!=num_iteraciones):remove(name_files+'.odb')
72  # Eliminar archivos innecesarios
73  list_delete = ('abaqus.*', '*.sta', '*.dat', '*.sim', '*.prt', '*.msg', '*.com', '*.log', '*.jnl')
74  for files in list_delete:
75      files_delete = glob(files)
76          for k in range(len(files_delete)):
77              remove(files_delete[k])
```

3.2.1.11 Acciones post-análisis

Estas líneas se encargan de organizar los archivos generados durante el código, dando la posibilidad de realizar sucesivos análisis sin que exista confusión entre ellos.

Una vez finalizado el análisis se dispone en el directorio de trabajo de varios archivos odb y de dos archivos txt. Por tanto en primer lugar se crea una carpeta dentro del directorio de trabajo cuyo nombre caracteriza el análisis realizado y acto seguido se procede a trasladar los archivos generados a dicha carpeta.

Para poder llevar a cabo sucesivos análisis es necesario eliminar la carpeta archivos_procesar, función que realiza la línea 80.

Código 3-11. Acciones post-análisis

```
79  # Crear carpeta y transladar archivos
80  if (float(tol_o_ptosint) < 1):
82      if (paso>0):
83          folder_name= name_INP + '_' + str(num_iteraciones) + 'steps' + '_' +
84                              str(paso) + 'paso' + '_' + tol_o_ptosint + 'tolerancia'
85      else:
            folder_name= name_INP + '_' + str(num_iteraciones) + 'steps'+ '_' + tol_o_ptosint + 'tolerancia'
86  else:
87      if (paso>0):
88          folder_name= name_INP + '_' + str(num_iteraciones) + 'steps' + '_' + str(paso) +
89                              'paso' + '_' + tol_o_ptosint + 'ptos_danar'
```

```
90          else:
                folder_name= name_INP + '_' + str(num_iteraciones) + 'steps'+ '_' + tol_o_ptosint + 'ptos_danar'
91      mkdir(folder_name)
92      list_move = ('*.txt', '*.odb')
93      for files in list_move:
94          files_move = glob(files)
95          for k in range(len(files_move)):
96              move(files_move[k],folder_name)
97      # Borrar directorio archivos_procesar
98      rmtree('C:\\archivos_procesar')
```

3.2.2 Script "readOdb.py"

Al igual que con el script anterior, se listan en la Tabla 3-4 las variables empleadas en este script, a continuación en la Figura 3-4 se representa el funcionamiento paso a paso del mismo y por último a través de una secuencia de subapartados se irán describiendo los diferentes pasos.

Tabla 3-4 Variables empleadas en "readOdb.py"

Variable	Descripción
archivo1	Variable con valor tipo file. Fichero de datos a procesar
archivo2	Variable con valor tipo file. Fichero de evolución de la rotura
archivo3	Variable con valor tipo file. Fichero de reacciones y desplazamientos
argv[1]	Nombre del archivo odb sin extensión
argv[2]	Valor tolerancia o número de puntos de integración a dañar
argv[3]	Nombre arc.txt que recoge información de la evolución de la rotura
argv[4]	Número de la iteración en la que se encuentra el proceso
argv[5]	Ruta del directorio de trabajo
k	Variable de control bucle for.
key_HisReg	Lista cuyo primer elemento es el nombre (clave) del HistoryRegions
key_step	Lista cuyo primer elemento es el nombre (clave) del step
línea	Lista de cuatro elementos, cada uno un tipo de línea. El objetivo es facilitar el diseño de los archivos txt

name_txt	Nombre del txt que recoge información de la evolución de la rotura
num_filas	Número de filas a añadir en el txt que recoge inf. evol. de la rotura
path_HisReg	Ruta del archivo odb para facilitar el acceso al HistoryOutputs
path_SDV2	Ruta del archivo odb para facilitar el acceso a la información referente al SDV2, es decir, a la variable factor
path_SDV3	Ruta del archivo odb para facilitar el acceso a la información referente al SDV3, es decir, a la variable noel (número de elemento)
RF1	Valor de la fuerza de reacción en la dirección 1
RF2	Valor de la fuerza de reacción en la dirección 2
tabla_all	Array para almacenar toda la información de interés del odb
tabla_ord	Array igual a tabla_all pero ordenado en función de la columna factor, de mayor a menor
tabla_print	Array que almacena la información a pasar a los archivos txt, tanto el que leerá la UMAT como el que recoge inf. de la evol. de la rotura
U1	Valor del desplazamiento en la dirección 1
U2	Valor del desplazamiento en la dirección 2

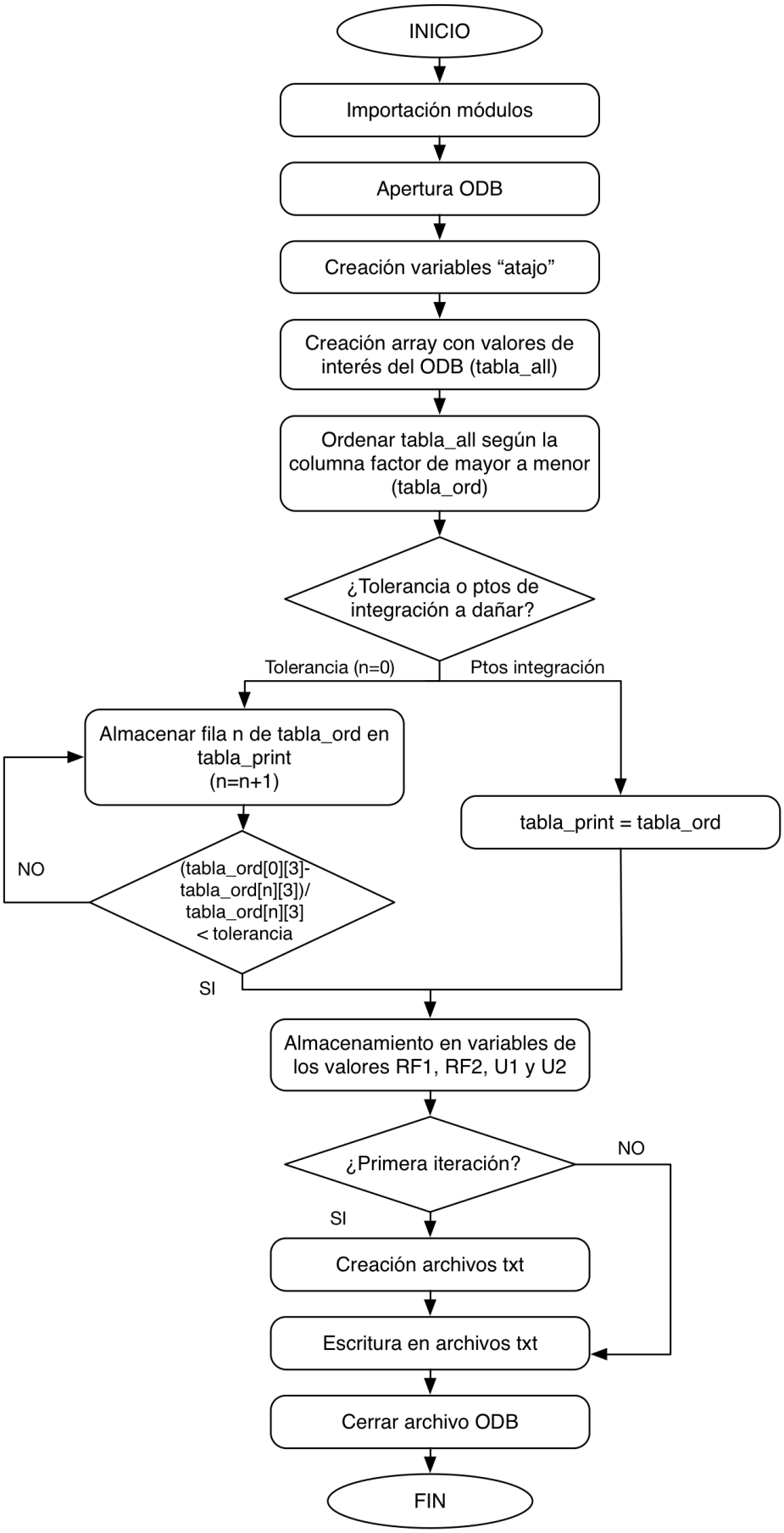


Figura 3-4 Diagrama de flujo script “readOdb.py”. Funcionamiento paso a paso

3.2.2.1 Importación módulos

Los módulos y métodos utilizados en este script se encuentran detallados en el subapartado 2.1.2 del capítulo 2.

Código 3-12. Importación módulos

```
10  from os import chdir
11  from operator import itemgetter
12  from odbAccess import*
13  from sys import argv
```

3.2.2.2 Apertura Odb

A través del método "openOdb()" se abre la base de datos de salida (ODB) creado en cada iteración. Se pasa el nombre de este como parámetro. Como el odb de cada iteración tendrá un nombre diferente, se utiliza la variable argv[1], la cual contiene el nombre del archivo odb pasado como primer argumento en la línea 65 del código main.

Este método crea un objeto Odb, el cual se asigna a la variable odb a través de la que se podrá acceder posteriormente al contenido de esta base de datos.

Código 3-12. Importación módulos

```
14  odb = openOdb(argv[1] + '.odb')
```

3.2.2.3 Creación variables "atajo"

Como se vio en el apartado 2.3 del capítulo anterior, para acceder al contenido del archivo odb y obtener la información de interés, se debe ir "navegando" e indicando los nombres de los repositorios donde se encuentra alojada la información de interés.

Para facilitar el acceso a dicha información a lo largo del código se han creado cinco variables, cuyas descripciones se han detallado en Tabla 3-4.

Código 3-13. Creación variables "atajo"

```
20  key_step = odb.steps.keys()
21  path_SDV2 = odb.steps[key_step[0]].frames[-1].fieldOutputs['SDV2'].values
22  path_SDV3 = odb.steps[key_step[0]].frames[-1].fieldOutputs['SDV3'].values
23  key_HisReg = odb.steps[key_step[0]].historyRegions.keys()
24  path_HisReg = odb.steps[key_step[0]].historyRegions[key_HisReg[0]]
```

3.2.2.4 Creación arrays "tabla_all" y "table_ord"

En la línea 31 se crea un array vacío llamado tabla_all. Con las líneas 32 y 33 se irá llenando este array a través del método append con los datos del odb de interés. Estos datos son: punto de integración, elemento local, elemento global, factor.

Una vez se dispone del array "tabla_all" lleno, este se ordena en función de la columna factor (número 3) de mayor a menor a través del método sorted (línea 36). Dicho array ordenado se almacena en la variable "tabla_ord".

Código 3-14. Creación arrays "tabla_all" y "tabla_ord"

```
31    tabla_all=[]
32    for k in range(len(path_SDV2)):
33      tabla_all.append([path_SDV2[k].integrationPoint, path_SDV2[k].elementLabel,
        path_SDV3[k].data, path_SDV2[k].data])
36    tabla_ord=sorted(tabla_all, key=itemgetter(3), reverse=True)
```

3.2.2.5 Creación array "tabla_print". Modo tolerancia o puntos de integración a dañar

En la línea 39 se crea un array vacío llamado tabla_print, el cual alojará los datos a pasar al archivo txt (evolución de la rotura). Depende del modo de análisis elegido por el usuario, se ejecutarán las líneas 40-43 (tolerancia) o la línea 46 (puntos de integración a dañar).

El array "tabla_ord" se encuentra ordenado de mayor a menor según la columna que contiene los valores de los factores. Por tanto en el caso del modo puntos de integración a dañar, solo habrá que coger el número de filas, empezando desde la primera, que el usuario haya dado como número de puntos de integración a dañar. En este paso simplemente se asigna a tabla_print el array tabla_ord al completo, posteriormente a la hora de la escritura en el archivo txt se pasará el número de filas concretas.

En el caso de modo tolerancia, se recorre tabla_ord desde la primera fila. Esta primera fila contiene información acerca del punto de integración donde se produce el máximo factor. Así que teniendo este valor máximo del factor como referencia vamos a ir recorriendo la columna factor del array tabla_ord y vamos almacenando en tabla_print toda fila cuya diferencia de factores cumpla la condición de tolerancia. Cuando se alcance la fila donde no se cumple la condición de tolerancia, se sale del bucle (comando break), de la búsqueda, ya que ninguna de las posteriores filas cumplirá la condición. De esta manera ahorramos tiempo de computación.

Código 3-15. Creación array "tabla_print". Modo tolerancia o puntos de integración a dañar

```
39    tabla_print=[]
40    if (float(argv[2]) < 1):
41      for k in range(len(tabla_ord)):
42        if (((tabla_ord[0][3] - tabla_ord[k][3])/tabla_ord[0][3]) > float(argv[2])): break
43        tabla_print.append([tabla_ord[k][0], tabla_ord[k][1], tabla_ord[k][2], tabla_ord[k][3]])
46    else: tabla_print=tabla_ord
```

3.2.2.6 Almacenamiento RF1, RF2, U1 y U2

Empleando las variables "atajo" almacenamos en cuatro variables los valores de las fuerzas de reacción en la dirección 1 (RF1) y en la dirección 2 (RF2) y los valores del desplazamiento en ambas direcciones también (U1, U2).

Código 3-16. Almacenamiento RF1, RF2, U1 y U2

```
53    RF1 = path_HisReg.historyOutputs['RF1'].data[1][1]
54    RF2 = path_HisReg.historyOutputs['RF2'].data[1][1]
55    U1 = path_HisReg.historyOutputs['U1'].data[1][1]
56    U2 = path_HisReg.historyOutputs['U2'].data[1][1]
```

3.2.2.7 Creación archivos txt

Estas líneas de código solo se ejecutarán en la primera iteración, cuando el valor de la variable step sea igual a cero (línea 104). El valor de esta variable se almacena en argv[4] al ser step pasado como cuarto argumento en la instrucción de la línea 65 del código main.py.

Se han desarrollado dos funciones, una para crear los archivos txt llamado "creartxt()" y otra para la escritura en ellos llamado "grabartxt()". Durante la ejecución de la primera se crean los archivos txt mostrados en la Tabla 3-5 a través del método open().

Tabla 3-5 Archivos txt creados

NOMBRE ARCHIVO	DESCRIPCIÓN
datos_procesar.txt	Almacena los puntos de integración "dañados" junto con el número del elemento (global) al que pertenecen
evolucion_rotura…txt	Almacena los puntos de integración "dañados" clasificado por iteraciones junto con la siguiente información referente a dichos puntos: Elemento Local, Elemento Global, Factor, u/factor, RF/factor
tabla_gráfica.txt	Almacena valores de las reacciones, desplazamientos y factor en cada iteración para que posteriormente el usuario pueda realizar la gráfica fuerza-desplazamiento del análisis. RF1, RF2, U1, U2 y Factor

Todos estos archivos se crean en el directorio de trabajo excepto el primero, el cual almacena los datos a leer por la UMAT en la siguiente iteración. Este archivo se creará en una carpeta alojada en la raíz (C:\) llamada archivos_procesar. El motivo de esto se explicó anteriormente en el subapartado 3.2.1.3.

Todos los archivos se crean en blanco excepto el último, en el cual se incorpora una cabecera con los nombres de los datos que contendrá cada columna.

Código 3-17. Creación archivos txt

```
64    def creartxt():
65        #Fichero para guardar los datos a procesar
66        chdir('C:\\archivos_procesar')
67        archivo1=open('datos_procesar.txt','w')
68        archivo1.close()
69        chdir(argv[5])
```

```
70          #Fichero que recoge la evolucion de la rotura
71          archivo2=open(name_txt,'w')
72          archivo2.close()
73          #Fichero reacciones y desplazamientos
74          archivo3=open('tabla_grafica.txt','w')
75          archivo3.write('Paso\t\tRF1\t\t\tRF2\t\tU1\t\tU2\t\tFactor')
76          archivo3.close()


104     if argv[4] == '0':
105         creartxt()
```

3.2.2.8 Escritura archivos txt

La escritura en los archivos txt se realiza en cada iteración tras haber procesado los datos del archivo odb y haber obtenido el array tabla_print.

Como se comentó anteriormente, se crea una función llamada "grabartxt()" la cual realiza las tareas de escritura en los tres archivos txt. Las líneas 78 y 79 asignan dependiendo del modo elegido el número de filas a escribir en los archivos. Si el modo es tolerancia, este valor será el número de filas que tenga tabla_print. Sin embargo, si es el otro modo, este valor será el dado por el usuario en el menú inicial, el cual es pasado como segundo argumento en la instrucción de la línea 65 del script main.py.

Se debe determinar el directorio de trabajo donde se encuentre el archivo en el cual se desea realizar la escritura. Por ello a la hora de escribir en "datos_procesar.txt" se utiliza el método "chdir" para cambiar el directorio de trabajo, ya que como se explicó anteriormente este se encuentra en la carpeta "archivos_procesar" alojada en la raíz. Una vez escrito en este archivo se cambia el directorio de trabajo al determinado por el usuario, para la escritura de los otros dos archivos.

Antes de llevar a cabo la escritura a través del método "write" se abre el archivo en el que se desee escribir. A diferencia que en la función "creartxt()" ahora pasamos como segundo parámetro "a", esto permite realizar la escritura donde se dejó el cursor la última vez.

En el archivo "datos_procesar.txt" se escriben dos columnas, una primera con los puntos de integración dañados y otra con los elementos globales en los cuales estos puntos se encuentran.

En el archivo "evolucion_rotura.txt" se escribe en cada iteración una cabecera donde aparece el número de la iteración en la que se encuentra el análisis y los títulos y valores de las seis columnas que contendrá este archivo: Pto.int, ElementoLocal, ElementoGlobal, Factor, u/factor y RF/factor.

En estos dos archivos los datos se obtienen recorriendo el array "tabla_print".

El último archivo llamado "tabla_gráfica.txt" contendrá los valores de las reacciones y desplazamientos en cada iteración. Como en la función creartxt() ya se escribió la cabecera, aquí solo se introduce los valores correspondientes.

Siempre que se termine la escritura en un archivo se cierra el mismo a través del método close().

Código 3-18. Escritura archivos txt

```
77  def grabartxt():
78      if (float(argv[2])<1): num_filas=len(tabla_print)
79      else: num_filas=int(argv[2])
80      #Fichero para guardar los datos a procesar
81      chdir('C:\\archivos_procesar')
82      archivo1=open('datos_procesar.txt','a')
83      for k in range(num_filas):
84          archivo1.write('%d %d\n' % (tabla_print[k][0], tabla_print[k][2]))
85      archivo1.close()
86      chdir(argv[5])
87      #Fichero que recoge la evolucion de la rotura
88      archivo2=open(name_txt,'a')
89      linea=['='*125, '_'*125, '|'*40, '|'*45]
90      archivo2.write('\n\n%s\n%s\t\tPASO%d\t\t\t%s\n%s\n\n'
            %(linea[0],linea[2],int(argv[4])+1,linea[3],linea[0]))
91      archivo2.write('Pto.Int\t\tElementoLocal\t\tElementoGlobal\t\tFactor
            \t\t\tu/factor\t\tRF/factor\n%s\n' %(linea[1]))
92      for k in range(num_filas):
93          archivo2.write('\n%d\t\t%d\t\t\t %d\t\t\t%e\t\t%e\t\t%e\n%s\n'
94              % (tabla_print[k][0], tabla_print[k][1], tabla_print[k][2], tabla_print[k][3],
95              U2/tabla_print[k][3], RF2/tabla_print[k][3], linea[1]))
96      archivo2.close()
97      #Fichero reacciones y desplazamientos
98      archivo3=open('tabla_grafica.txt','a')
99      archivo3.write('\n%d\t\t%e\t\t%e\t\t%e\t\t%e\t\t%e'
100         % (int(argv[4])+1, RF1, RF2, U1, U2, tabla_ord[0][3] ))
101     archivo3.close()


105 grabartxt()
```

3.2.2.9 Cerrar archivo ODB

Es buena costumbre siempre que se termine de manipular un archivo ODB proceder a su cierre. Esto se lleva a cabo a través del método close().

Código 3-19. Cierre archivo ODB

```
109 odb.close()
```

4 TUTORIAL DEL CÓDIGO

Hasta el momento se ha desarrollado la teoría en la cual se basa el código y se ha explicado con detalle su implementación y funcionamiento. Sin embargo, aunque recomendable, el usuario puede no poseer estos conocimientos para su uso. A lo largo de este capítulo se intenta explicar los pasos a seguir por el usuario para el manejo de este código. Estos pasos se desarrollan en diferentes apartados para facilitar al lector su seguimiento.

4.1 Carpeta para el análisis y archivos necesarios

En primer lugar crear una carpeta para alojar los cuatro archivos necesarios para ejecutar el análisis, Tabla 4-1. Copiar la ruta de dicha carpeta ya que posteriormente será necesaria.

Tabla 4-1 Archivos necesarios para el análisis

Nombre archivo	Descripción
main.py	Script principal del código
readOdb.py	Script postproceso
archivo.inp	Datos del modelo
archivo.for	Subrutina de usuario (UMAT)

Los dos primeros archivos implementan el código SLA, por lo tanto son los programados en este TFG. El archivo.for fue desarrollado por D. Castillo [6], aunque el utilizado en el presente trabajo ha sufrido algunas modificaciones, las cuales se detallan en el Anexo B. El nombre dado a este archivo es "lebim.for". El archivo.inp, el cual se obtiene a través de Abaqus, es el único que el usuario debe generar. Se tendrá en cuenta para ello las anotaciones citadas en el Anexo C.

Antes de lanzar el análisis se debe comprobar la existencia de estos archivos en la carpeta creada.

4.2 Modificar "main.py"

Una vez desarrollado el primer paso, debemos abrir el archivo "main.py" mediante un editor de texto para llevar a cabo la modificación de algunas de sus líneas. Un posible editor de texto y de código fuente libre a utilizar es Notepad++.

Las líneas a modificar, mostradas abajo, son desde la diez a la trece, la cuales contienen asignaciones a variables: asignación de nombres de archivos y de la ruta del directorio de trabajo.

```
7   #-----------------------------------------------------------------------------
8   # NOMBRES DE NUESTROS ARCHIVOS DE TRABAJO (MODIFICAR)
9   #-----------------------------------------------------------------------------
10  name_INP = 'placasUMAT'
11  name_UMAT = 'lebim'
12  post_ODB = 'readOdb' + '.py'
13  working_directory ='C:\scriptsabaqus\placas_UMAT'
14  #-----------------------------------------------------------------------------
```

Las líneas diez y trece deben ser modificadas obligatoriamente, mientras que la once y doce son opcionales.

- Línea 10: nombre de nuestro modelo sin extensión, es decir el nombre dado al archivo.inp
- Línea 13: ruta de la carpeta que aloja todos los archivos del análisis, creada en el aparatado anterior

Si el usuario no ha modificado, dándole otro nombre, ni el archivo de postproceso (readOdb.py) ni el archivo que modela el comportamiento del material (lebim.for) no se debe modificar las líneas once y doce. Si por el contrario, si lo ha hecho, están deben ser modificadas, correspondiendo la línea once al nombre dado al archivo.for y la línea doce al nombre dado al archivo de postproceso. Este último debe contener su extensión (.py).

4.3 Lanzar análisis

Este paso puede llevarse a cabo de dos maneras diferentes. La primera y más sencilla seria haciendo doble-click en el archivo "main.py". Automáticamente se abre la consola del sistema (CMD) con la primera pregunta del menú de análisis. La otra forma, sería utilizando la Instrucción CMD 1-2 del capítulo 2. Antes de ejecutar la instrucción hay que estar situado en la carpeta donde se aloje el archivo main.py.

4.4 Menú del análisis

Una vez ejecutado el archivo "main.py" por cualquiera de las vías anteriormente explicadas, el usuario debe configurar su análisis a través del menú mostrado por pantalla. La estructura de este se recoge en el diagrama de flujo de la Figura 4-1 y sus bloques se detallan en sucesivos subapartados.

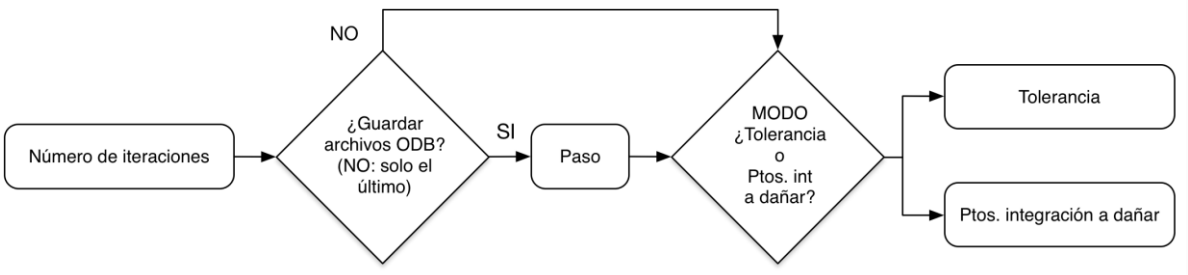


Figura 4-1 Diagrama de flujo del menú de análisis

4.4.1 Número de iteraciones

Este número corresponde al número de veces que vamos a lanzar Abaqus a resolver el problema. Cada iteración tendrá en cuenta los puntos de integración "dañados" en la iteración anterior.

El usuario debe introducir por teclado el número de iteraciones que quiera para su análisis.

4.4.2 Guardar archivos ODB

Esta opción no repercute en el análisis, simplemente es implementada con el objetivo de facilitar al usuario la manipulación de los archivos odb de salida, ya que muchas veces no es necesario el almacenaje de todos ellos para la posterior interpretación del análisis. De esta manera se evita también saturar la carpeta con dichos archivos odb.

La pregunta mostrada por pantalla ("¿Ir guardando ODB (y/n)?) plantea dos posibilidades. Su elección se hará mediante la introducción por teclado del carácter 'y' (afirmación) o 'n' (negación). Si se afirma la pregunta, a continuación el usuario debe introducir un valor de paso. Si es negada se pasará a la siguiente opción del menú.

Carácter 'y': Se almacena el archivo odb generado en la iteración, si el número de la iteración es múltiplo del valor de paso dado por el usuario. Por tanto, a modo de ejemplo si el usuario introduce como valor de paso 1, se guardan todos los archivos odb generados durante el proceso. Si es dado el valor 2, se guardan los odb generados en las iteraciones 2, 4, 6,…

Carácter 'n': Solo se guarda el archivo odb generado en la última iteración.

4.4.3 Formas de fallo: número de puntos de integración o tolerancia

Esta opción si repercute en el análisis al contrario que la anterior.

En cada iteración se postprocesa los resultados obtenidos y se crea una tabla donde una de sus columnas contiene los valores de la variable factor de cada punto de integración, ordenada de mayor a menor. El punto de integración con mayor factor, el primero en la lista, es por donde comenzará o continuará la rotura y por tanto es el punto de integración a "dañar". Se plantean dos posibles opciones. Por un lado, dañar no solo este punto de mayor valor de factor sino el número de puntos de integración que el usuario desee. Y por otro lado, teniendo en cuenta que es posible que existan varios puntos de integración con valores de factor muy cercanos, dañar aquellos que se encuentren dentro de un rango de tolerancia, tomando como referencia el máximo factor. Esta tolerancia es dada como porcentaje.

Por ello se ha decido implementar dos modos de análisis, el primero correspondiente a la primera opción citada anteriormente, llamado "Puntos de integración a dañar" y el segundo correspondiente a la otra opción, llamado "Tolerancia".

Para elegir un modo u otro hay que introducir por teclado el número 1 o 2. Como se muestra en pantalla y se refleja en la Figura 4-2, para elegir el modo "Tolerancia" se introduce el valor 1 y para el modo "Puntos de integración a dañar" el valor 2.

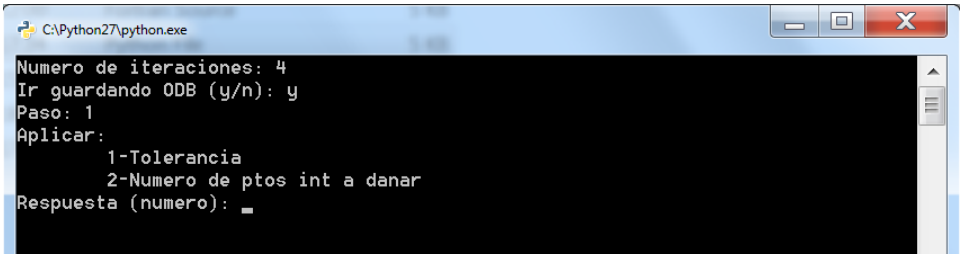


Figura 4-2 Salida por pantalla del menú de análisis

Una vez elegido el modo, se pide por pantalla el valor de tolerancia (porcentaje) o el número de puntos de integración a dañar, según corresponda. En el caso de tolerancia si se pone por ejemplo $1 \cdot 10^{-2}$ significa que la diferencia entre los factores de los puntos de integración dañados en cada iteración se encuentran dentro de este rango de tolerancia.

Con esto se termina todo lo referente a la configuración del análisis y comienza este a correr. Cuando este proceso termine, se dispondrá de los archivos generados durante él, en una carpeta, cuyo nombre caracteriza al análisis, alojada en el directorio de trabajo dado por el usuario.

5 MODELO DEL ENSAYO DE TENACIDAD A LA FRACTURA INTERLAMINAR EN MATERIALES COMPUESTOS. APLICACIÓN GIC

En este capítulo se modela el ensayo que se emplea en los materiales compuestos para medir la tenacidad a fractura interlaminar, G_{Ic}. Su implementación se llevará a cabo mediante el software Abaqus, utilizando el modelo LEBIM y el algoritmo de resolución SLA desarrollados en este trabajo.

La medición de la tenacidad a la fractura interlaminar en uniones composite-composite está bien descrita en [2, 10] mediante la prueba de DCB.

5.1 Descripción de la probeta de ensayo

La probeta utilizada es una "Double Cantilever Beam" (DCB), cuyas dimensiones se recogen en la Figura 5-1 (a) Esquema de la probeta DCB, (b) probeta con mordazas para el ensayo, (c) configuración del ensayo [22]

. Está compuesta por dos laminados unidireccionales unidos por una capa de adhesivo. Hay una grieta preexistente de 25 mm de longitud, Figura 5-1a. La carga se aplica a través de unas mordazas, estando el punto de aplicación de la carga a 13 mm del extremo de la probeta, Figura 5-1b, y el ensayo se realiza con un control en desplazamientos, Figura 5-1c.

El laminado es fabricado usando un composite de fibra de carbono-epoxy 8552/AS4 (capas a 0º y la dirección de referencia del material coincide con la dirección longitudinal de la probeta). El material adhesivo es el EA 9695 K.05, con base de resina epoxy y con una malla soporte de poliéster.

Tabla 5-1 Dimensiones probeta DCB

	Longitud (mm)	**Ancho (mm)**	**Espesor (mm)**
Laminado	250	25	1.5
Capa de adhesivo	225	25	0.01

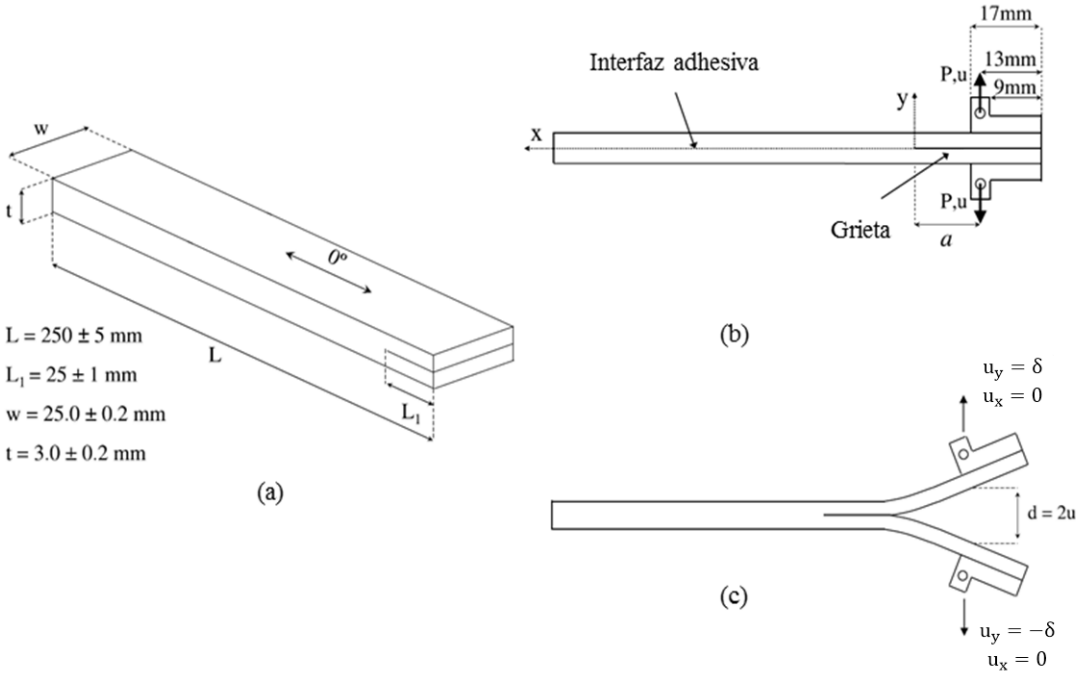


Figura 5-1 (a) Esquema de la probeta DCB, (b) probeta con mordazas para el ensayo, (c) configuración del ensayo [22]

5.2 Modelo numérico

El modelo resuelto cumple con la normativa ISO 15024 [10]. Para este ensayo se parte de la hipótesis de deformación plana.

Los laminados de composite son modelados como materiales ortótropos homogéneos, cuyas propiedades se recogen en la Tabla 5-2.

Tabla 5-2 Propiedades del laminado

Propiedades del laminado	
Módulo de elasticidad en la dirección de la fibra	$E_x = 135\ GPa$
Módulo de elasticidad en la dirección perpendicular a la fibra dentro del plano	$E_y = 10\ GPa$
Módulo de elasticidad en la dirección perpendicular a la fibra perpendicular al plano	$E_z = 10\ GPa$
Módulo de cizalladura en el plano xy	$G_{xy} = 5\ GPa$
Módulo de cizalladura en el plano xz	$G_{xz} = 5GPa$
Coeficiente de Poisson en el plano xy	$\nu_{xy} = 0.3$
Coeficiente de Poisson en el plano yz	$\nu_{yz} = 0.3$
Coeficiente de Poisson en el plano xz	$\nu_{xz} = 0.3$

Las propiedades empleadas para modelar el material adhesivo EA 9695 K.05 con el LEBIM son las recogidas en la Tabla 5-3.

Tabla 5-3 Propiedades del adhesivo EA9695 K.05

Propiedades del adhesivo	
Lambda	$\lambda = 0.25$
Tensión crítica de rotura	$\sigma_c = 15\ MPa$
Rigidez del adhesivo en dirección normal al adhesivo	$k_n = 150\ GPa/m$
Relación de rigideces	$\xi = 0.25$
Espesor	$h = 0.01\ mm$

Este problema ha sido resuelto utilizando el programa de elementos finitos Abaqus. Para modelar el ensayo, la probeta ha sido dividida en dos laminados de material compuesto unidos por una capa de adhesivo cuyas dimensiones se recogen en la Tabla 5-1.

Se utiliza la subrutina UMAT para modelar el comportamiento del material adhesivo asociando esta a una zona del modelo compuesto por 161 elementos del tipo CPE4. Se impone que solamente haya un elemento en el espesor del adhesivo. Las dimensiones de cada elemento son: longitud 1.398mm y ancho 0.01mm. La elección de la longitud del elemento, 1.398mm, está basada en los estudios realizados por D.Castillo [6]. El cual concluye que si el tamaño del elemento es parecido al tamaño de la malla de poliéster del material adhesivo se obtendrán resultados más cercanos a los reales.

La carga del modelo estará aplicada a 13 mm del extremo de la probeta donde se encuentra la grieta, en ambos laminados. En él se impondrá un desplazamiento vertical de 1 mm, positivo para el caso del laminado que se encuentra arriba y negativo para el que se encuentra debajo. Para impedir los movimientos como sólido rígido se ha impedido el desplazamiento horizontal en ambos puntos.

5.3 Resultados numéricos

Como se ha comentado en diferentes puntos del trabajo, el análisis se puede realizar de dos modos diferentes: tolerancia o puntos de integración a dañar. En este subapartado se presentan los resultados obtenidos al ejecutar el análisis según el modo tolerancia con valores: $2{\cdot}10^{-1}$, $1{\cdot}10^{-1}$, $1{\cdot}10^{-2}$ y $1{\cdot}10^{-3}$ y según el modo puntos de integración a dañar, para valores de 8, 4, 2 y 1.

Los resultados obtenidos se presentarán a continuación mediante curvas fuerza-desplazamiento. Al final del capítulo se realizarán una serie de comentarios acerca de estos resultados.

5.3.1 Modo Tolerancia: $2{\cdot}10^{-1}$, $1{\cdot}10^{-1}$, $1{\cdot}10^{-2}$, $1{\cdot}10^{-3}$

Cabe destacar que para valores de tolerancia: $1{\cdot}10^{-1}$, $1{\cdot}10^{-2}$, $1{\cdot}10^{-3}$ se han obtenido los mismos resultados, de manera que solo se mostrará a continuación los resultados obtenidos para uno de ellos, Figura 5-3.

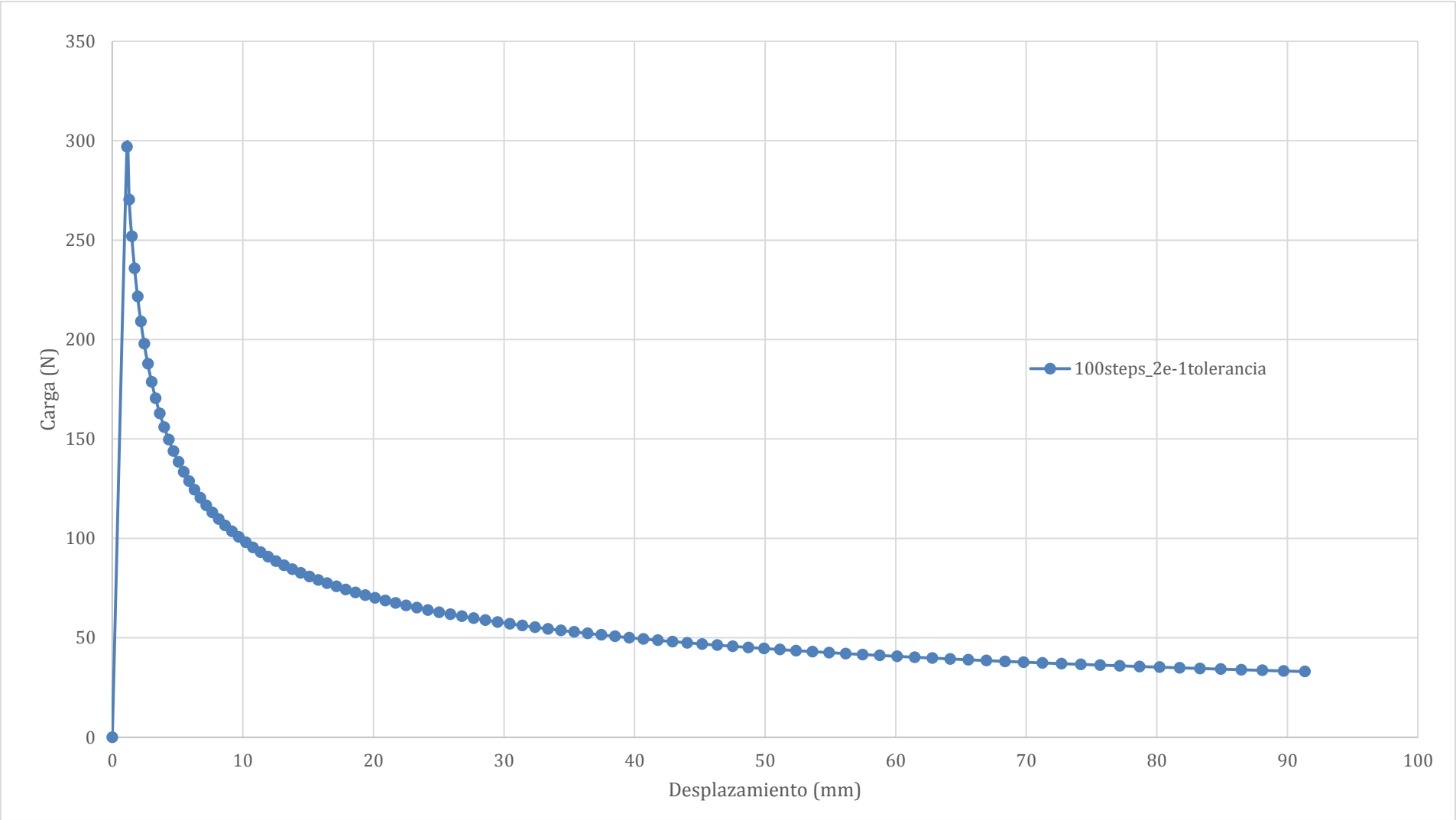


Figura 5-2 Curva fuerza-desplazamiento entre mordazas. Modo tolerancia: $2 \cdot 10^{-1}$

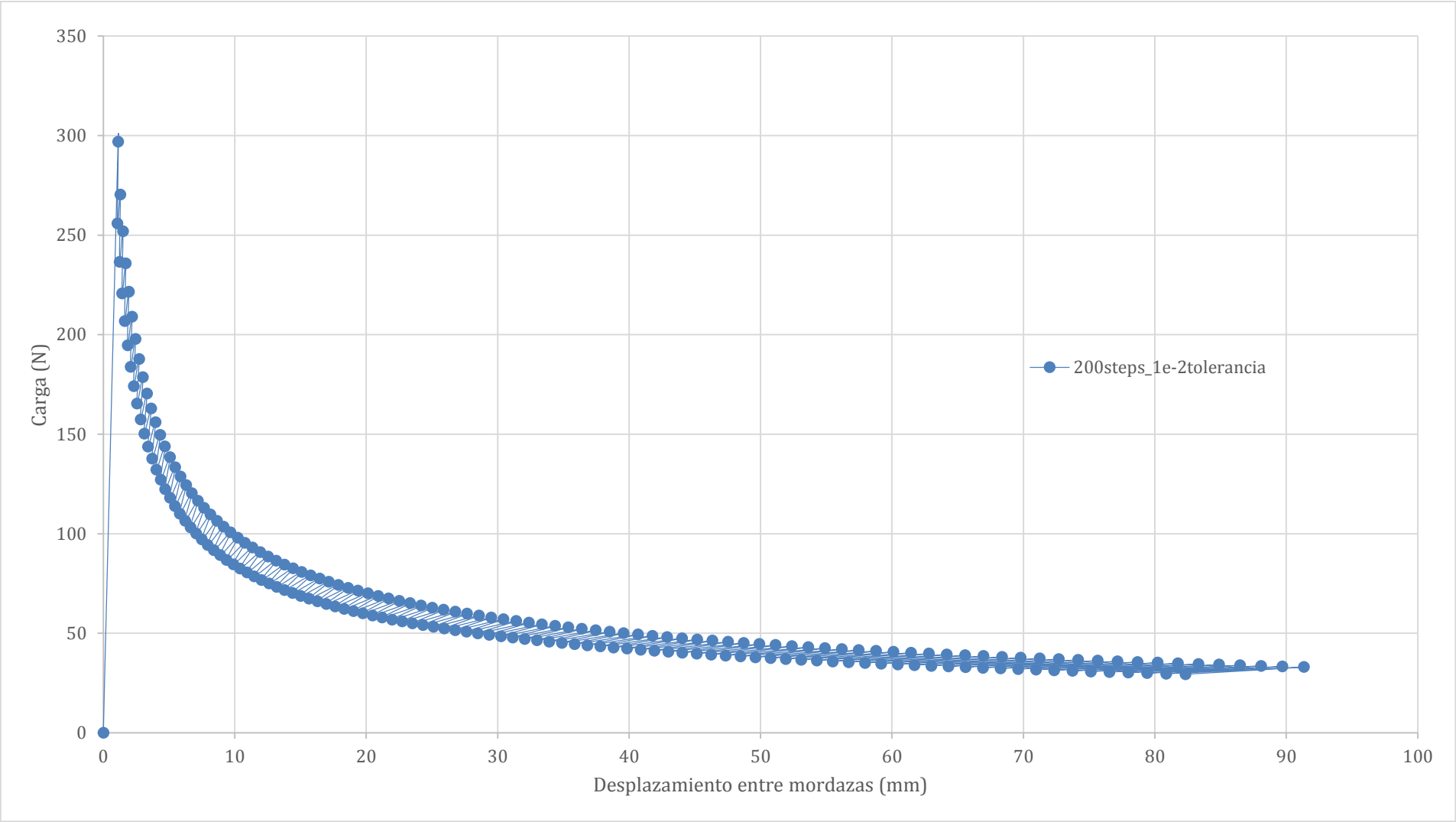


Figura 5-3 Curva fuerza-desplazamiento entre mordazas. Modo tolerancia: $1 \cdot 10^{-2}$

5.3.2 Modo número puntos de integració: 8, 4, 2 y 1

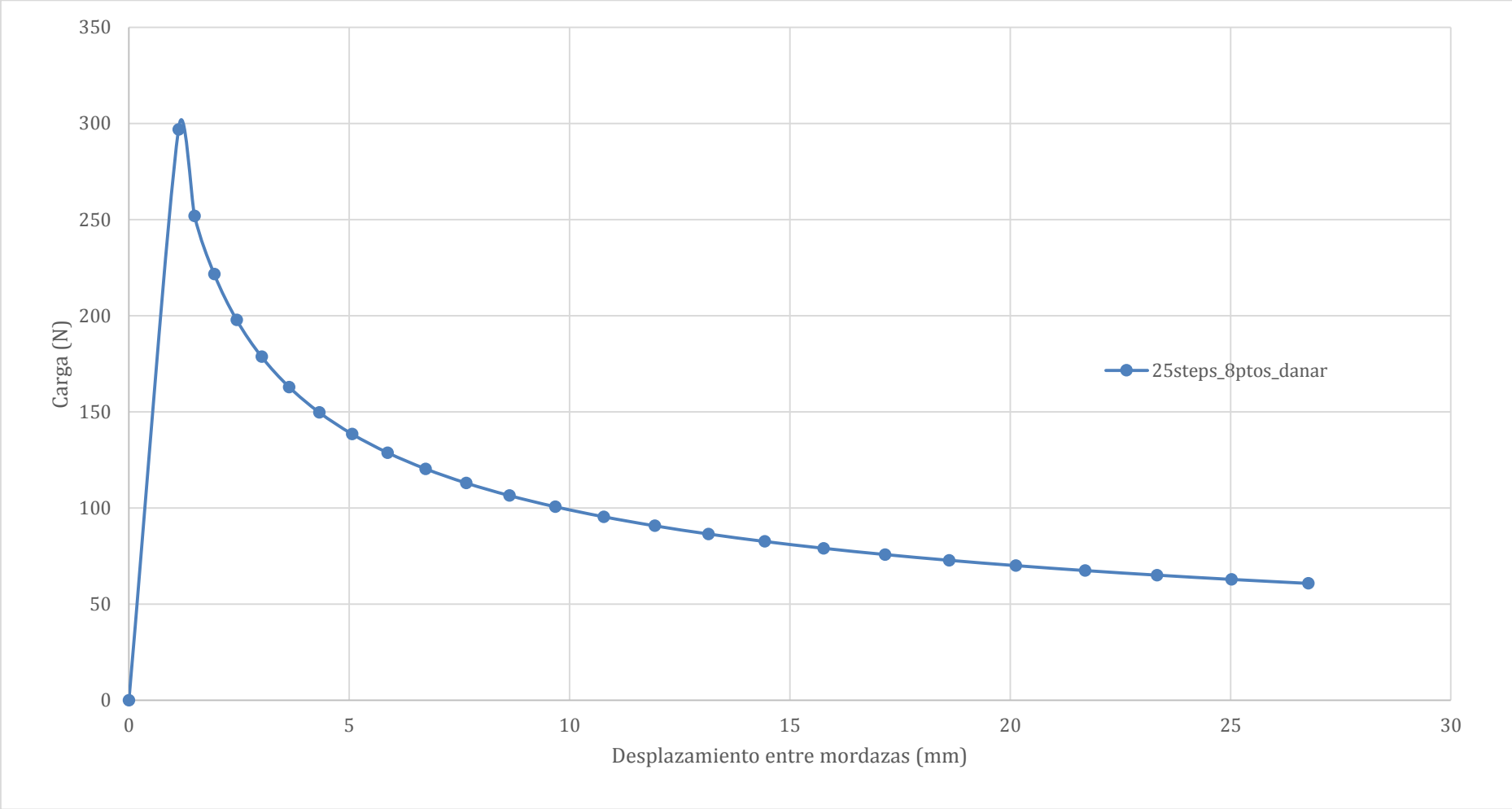


Figura 5-4 Curva fuerza-desplazamiento entre mordazas. 8 ptos int. dañar

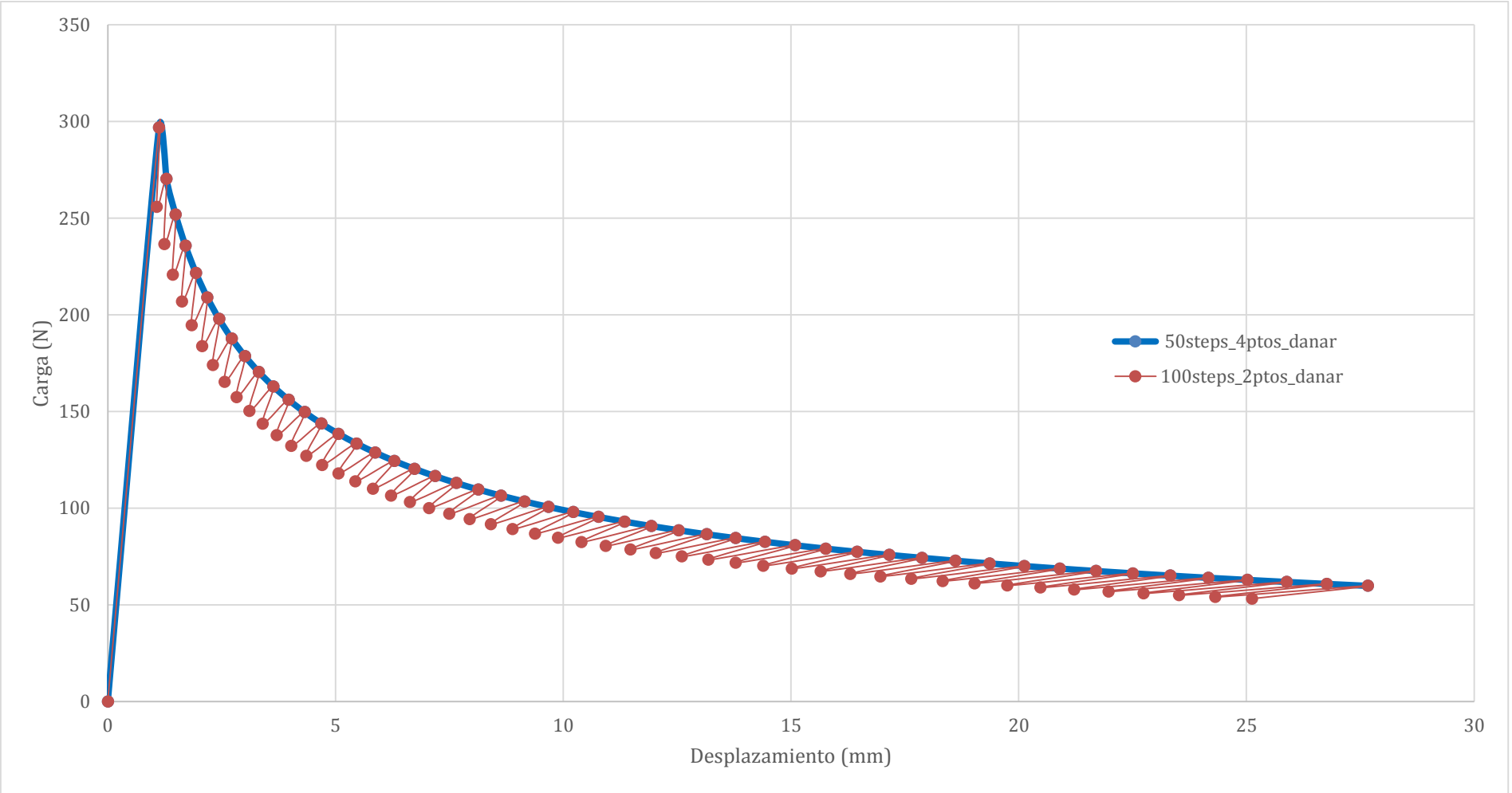


Figura 5-5 Curva fuerza-desplazamiento entre mordazas. 2 y 4 ptos int. dañar

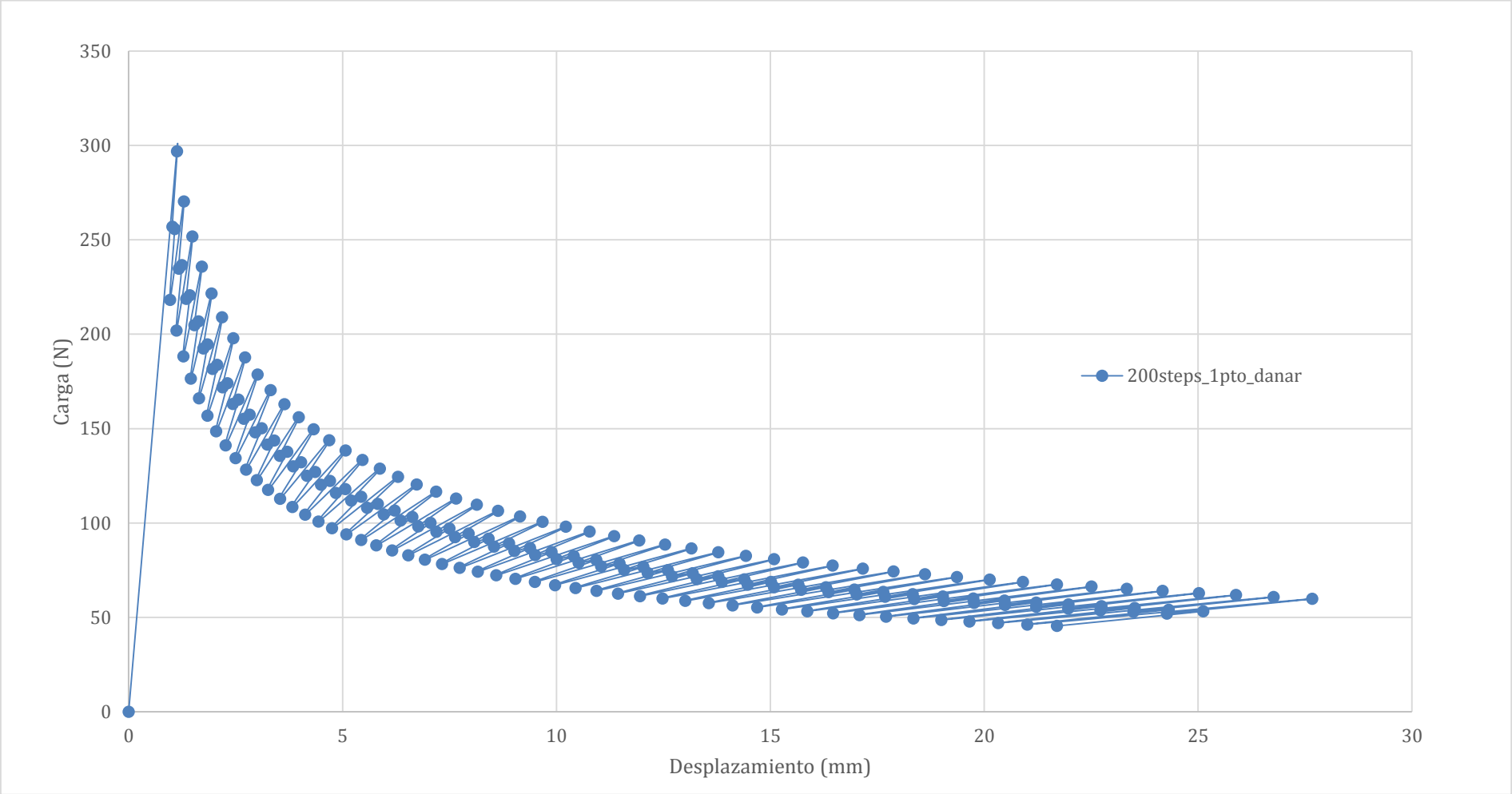


Figura 5-6 Curva fuerza-desplazamiento entre mordazas, 1 pto int. dañar

Destacar primero que la subrutina UMAT se aplica en los puntos de integración y no en los nodos, por lo que los resultados en tensiones de los nodos pueden no ser exactos debido a la extrapolación que realiza Abaqus de los resultados obtenidos en los puntos de integración.

Se han presentado los resultados de los diferentes casos en curvas fuerza-desplazamiento, donde el eje horizontal corresponde a la distancia entre los puntos de aplicación del desplazamiento y el eje vertical a la fuerza que debemos de aplicar para obtener ese desplazamiento.

Acerca de los resultados obtenidos decir:

1. La carga máxima que se obtiene en todos los casos es la misma: 296.92 N, siempre para el mismo desplazamiento entre mordazas: 1.12 mm.
2. Una vez "dañado" un elemento, el siguiente lo hace a menor carga aplicada y a mayor desplazamiento entre mordazas.
3. Al dañar elemento a elemento o de dos en dos, Figura 5-2, Figura 5-5 y Figura 5-4, se obtienen curvas lineales.
4. Se dañan todos los puntos de integración de un elemento antes de saltar al siguiente.
5. Con tolerancias por debajo del 20% siempre se obtienen los mismos resultados, se "dañan" de dos en dos los puntos de integración. Esto es debido a que ambos puntos tienen el mismo valor de factor. Con tolerancias del 20% se "daña" elemento a elemento.
6. Al dañar los puntos de integración de uno en uno y recordando que cada elemento está compuesto por cuatro puntos se puede apreciar que se va dañando elemento a elemento siguiendo la siguiente secuencia: el primer punto de integración "se daña" a mayor carga y mayor desplazamiento de mordazas que los otros, los siguientes dos puntos lo hacen a la misma carga y desplazamiento. El último punto del elemento "se daña" a menor carga y desplazamiento. Una vez dañado el elemento, el primer punto de integración del siguiente lo hará a mayor carga y desplazamiento entre mordazas que para los tres puntos de integración del elemento ya dañado anteriormente pero a menor que al primero.
7. En la Figura 5-5 y Figura 5-6**¡Error! No se encuentra el origen de la referencia.** a medida que se van dañando elementos la diferencia de carga aplicada entre los primeros puntos de integración "dañados" y los últimos de un mismo elemento disminuye, al contrario que el desplazamiento entre mordazas que aumenta.

A. ANEXO: CÓDIGO SLA

```
# ******************************************************************************
# SCRIPT: MAIN.PY
# Autor: Enrique Paloma Castro, GERM (2016)
# ******************************************************************************
#-------------------------------------------------------------------------------------------------
# ||||||   NOMBRES DE NUESTROS ARCHIVOS DE TRABAJO (MODIFICAR)    ||||||
#-------------------------------------------------------------------------------------------------
name_INP = 'placasUMAT'
name_UMAT = 'lebim'
post_ODB = 'readOdb' + '.py'
working_directory = 'C:\scriptsabaqus\placas_UMAT'
#-------------------------------------------------------------------------------------------------

#-------------------------------------------------------------------------
# ||||||||          IMPORTACION DE MODULOS              ||||||||
#-------------------------------------------------------------------------
from os import chdir, path, remove, mkdir, system
from shutil import move, rmtree
from glob import glob
from subprocess import call
#-------------------------------------------------------------------------
```

```
# Indicar directorio de trabajo y crear directorio en la raiz para el archivo 'datos_procesar.txt'
chdir(working_directory)
mkdir('C:\\archivos_procesar')
#-----------------------------------------------------------------------------------------------------------------------------------------------------------
# ||||||||||||||||||||||||||||||||                    ENTRADAS POR TECLADO                    ||||||||||||||||||||||||||||||||
#-----------------------------------------------------------------------------------------------------------------------------------------------------------
call('cls',shell=True)
# Pedir numero de iteraciones
num_iteraciones = input("Numero de iteraciones: ")

# Preguntar si se guardan todos los ODB y en caso afirmativo el paso
respuesta_ODB = raw_input("Ir guardando ODB (y/n): ")
while (respuesta_ODB != 'y') and (respuesta_ODB != 'n'):
	respuesta_ODB = raw_input("Ir guardando ODB (y/n): ")
if (respuesta_ODB=='y'):
	paso = input("Paso: ")
else:
	paso=0

# Preguntar si queremos aplicar tolerancia o numero de ptos int a danar
respuesta_MODO = input("Aplicar:\n\t1-Tolerancia\n\t2-Numero de ptos int a danar\nRespuesta (numero): ")
while (respuesta_MODO != 1) and (respuesta_MODO != 2):

	respuesta_MODO = input("Aplicar:\n\t1-Tolerancia\n\t2-Numero de ptos int a danar\nRespuesta (numero): ")

if (respuesta_MODO==1):
	tol_o_ptosint = raw_input("Tolerancia: ")
else:
	tol_o_ptosint = raw_input("Numero ptos de int a danar: ")
```

```
# Dar nombre al archivo txt con el numero de iteraciones y la tolerancia
if (float(tol_o_ptosint) < 1):
	name_txt_evolucion = 'evolucion_rotura' + '_' + str(num_iteraciones) + 'steps' + '_' + tol_o_ptosint
		+ 'tolerancia'
else:
	name_txt_evolucion = 'evolucion_rotura' + '_' + str(num_iteraciones) + 'steps' + '_' + tol_o_ptosint
		+ 'ptos_danar'
#-----------------------------------------------------------------------------------------------------------------------------------------------

#-----------------------------------------------------------------------------------------------------------------------------------------------
# |||||||||||||||||||||||||||||||||  ITERACIONES  |||||||||||||||||||||||||||||||||
#-----------------------------------------------------------------------------------------------------------------------------------------------
for step in range(num_iteraciones):
	# Dar nombre a los archivos de salida en la iteracion
	if (float(tol_o_ptosint) < 1): name_files = name_INP + '_' + str(num_iteraciones) + 'steps' + '_'
		+ str(step + 1) + '_' + tol_o_ptosint + 'tol'
	else: name_files = name_INP + '_' + str(num_iteraciones) + 'steps' + '_' + str(step + 1) + '_'
		+ tol_o_ptosint + 'ptos_danar'

	# Ejecutar inp + UMAT
	call('abaqus Job=' + name_files + ' ' + 'input=' + name_INP + ' ' + 'user=' + name_UMAT,shell=True)

	# Esperamos a que se cree nuestro archivo ODB y se elimine el LCK
	while (path.exists(name_files + '.odb') == False) or (path.exists(name_files + '.lck') == True): pass
```

```
        # Postprocesado ODB
        call('abaqus python' + ' ' + post_ODB + ' ' + name_files + ' ' + tol_o_ptosint + ' ' + name_txt_evolucion
               + ' ' + str(step) + ' ' + working_directory,shell=True)

        # Borrar ODB cumpliendo las condiciones dadas
        if (respuesta_ODB=='y') and (((step+1)%paso)!=0):
               remove(name_files + '.odb')
        elif (respuesta_ODB=='n') and ((step+1)!=num_iteraciones):
               remove(name_files + '.odb')

        # Eliminar archivos innecesarios
        list_delete = ('abaqus.*', '*.sta', '*.dat', '*.sim', '*.prt', '*.msg', '*.com', '*.log', '*.jnl')
        for files in list_delete:
               files_delete = glob(files)
               for k in range(len(files_delete)):
                      remove(files_delete[k])
#-----------------------------------------------------------------------------------------------------------------------------------------------

#-----------------------------------------------------------------------------------------------------------------------------------------------
# |||||||||||||||||||||||||||||||||  ORGANIZAR ARCHIVOS  |||||||||||||||||||||||||||||||||
#-----------------------------------------------------------------------------------------------------------------------------------------------
# Borrar directorio 'archivos_procesar'
rmtree('C:\\archivos_procesar')
# Crear carpeta y trasladar archivos
if (float(tol_o_ptosint) < 1):
       if (paso>0):
               folder_name= name_INP + '_' + str(num_iteraciones) + 'steps' + '_' + str(paso) + 'paso' + '_'
                      + tol_o_ptosint + 'tolerancia'
```

```
        else:
                folder_name= name_INP + '_' + str(num_iteraciones) + 'steps' + '_' + tol_o_ptosint + 'tolerancia'
else:
        if (paso>0):
                folder_name= name_INP + '_' + str(num_iteraciones) + 'steps' + '_' + str(paso) + 'paso' + '_'
                        + tol_o_ptosint + 'ptos_danar'
        else:
                folder_name= name_INP + '_' + str(num_iteraciones) + 'steps' + '_' + tol_o_ptosint + 'ptos_danar'
mkdir(folder_name)
list_move = ('*.txt', '*.odb')
for files in list_move:
        files_move = glob(files)
        for k in range(len(files_move)):
                move(files_move[k],folder_name)
```

```
# ******************************************************************************
# SCRIPT: READODB.PY
# Autor: Enrique Paloma Castro, GERM (2016)
# ******************************************************************************
#------------------------------------------------------------------------------
# ||  IMPORTACION DE MODULOS Y APERTURA DE ODB   ||
#------------------------------------------------------------------------------
from os import chdir
from operator import itemgetter
from odbAccess import*
from sys import argv
odb = openOdb(argv[1] + '.odb')
#------------------------------------------------------------------------------

#-------------------------------------------------------------------------------------------------------------
# ||||||||||||||                    DEFINICION DE VARIABLES                              ||||||||||||||||
#-------------------------------------------------------------------------------------------------------------
key_step = odb.steps.keys()
path_SDV2 = odb.steps[key_step[0]].frames[-1].fieldOutputs['SDV2'].values
path_SDV3 = odb.steps[key_step[0]].frames[-1].fieldOutputs['SDV3'].values
key_HisReg = odb.steps[key_step[0]].historyRegions.keys()
path_HisReg = odb.steps[key_step[0]].historyRegions[key_HisReg[0]]
#-------------------------------------------------------------------------------------------------------------
```

```
#-----------------------------------------------------------------------------------------------------------------------------------------------
# ||||||||||||||||||                CREACION DE TABLA CON LOS PUNTOS DE INTEGRACION A DANAR                ||||||||||||||||||
#-----------------------------------------------------------------------------------------------------------------------------------------------
# Crear tabla con todos los valores del ODB que nos interesa
tabla_all=[]
tabla_print=[]
for k in range(len(path_SDV2)):
	tabla_all.append([path_SDV2[k].integrationPoint, path_SDV2[k].elementLabel, path_SDV3[k].data,
		path_SDV2[k].data])

# Ordenar de mayor a menor la tabla anterior en funcion de la cuarta columna (SDV2 == factor)
tabla_ord=sorted(tabla_all, key=itemgetter(3), reverse=True)

# En el caso de aplicar "tolerancia"
if (float(argv[2]) < 1):
	for k in range(len(tabla_ord)):
		if (((tabla_ord[0][3] - tabla_ord[k][3])/tabla_ord[0][3]) > float(argv[2])): break
		tabla_print.append([tabla_ord[k][0], tabla_ord[k][1], tabla_ord[k][2], tabla_ord[k][3]])

# En el caso de aplicar "numero de puntos de integracion a danar"
else: tabla_print=tabla_ord
#-----------------------------------------------------------------------------------------------------------------------------------------------
```

```
#-------------------------------------------------------------------------------
# ||||||              OBTENER RF1, RF2, U1, U2                    ||||||
#-------------------------------------------------------------------------------
RF1 = path_HisReg.historyOutputs['RF1'].data[1][1]
RF2 = path_HisReg.historyOutputs['RF2'].data[1][1]
U1 = path_HisReg.historyOutputs['U1'].data[1][1]
U2 = path_HisReg.historyOutputs['U2'].data[1][1]
#-------------------------------------------------------------------------------

#-------------------------------------------------------------------------------------------------------------------------------------------------------
#|||||||||||||||||||||||||||||||||||||                    FICHEROS DE TEXTO                    |||||||||||||||||||||||||||||||||||||
#-------------------------------------------------------------------------------------------------------------------------------------------------------
#Crear fichero y guardar datos
name_txt = argv[3] + '.txt'
def creartxt():
        #Fichero para guardar los datos a procesar
        chdir('C:\\archivos_procesar')
        archivo1=open('datos_procesar.txt','w')
        archivo1.close()
        chdir(argv[5])

        #Fichero que recoge la evolucion de la rotura
        archivo2=open(name_txt,'w')
        archivo2.close()
```

```
        #Fichero reacciones y desplazamientos
        archivo3=open('tabla_grafica.txt','w')
        archivo3.write('Paso\t\tRF1\t\t\tRF2\t\tU1\t\tU2\t\tFactor')
        archivo3.close()

def grabartxt():
        if (float(argv[2])<1): num_filas=len(tabla_print)
        else: num_filas=int(argv[2])

        #Fichero para guardar los datos a procesar
        chdir('C:\\archivos_procesar')
        archivo1=open('datos_procesar.txt','a')
        for k in range(num_filas):
                archivo1.write('%d %d\n'
                        % (tabla_print[k][0], tabla_print[k][2]))
        archivo1.close()
        chdir(argv[5])

        #Fichero que recoge la evolucion de la rotura
        archivo2=open(name_txt,'a')
        linea=['='*125, '_'*125, '|'*40, '|'*45]
        archivo2.write('\n\n%s\n%s\t\tPASO %d\t\t\t%s\n%s\n\n'
                %(linea[0],linea[2],int(argv[4])+1,linea[3],linea[0]))
        archivo2.write('Pto.Int\t\tElemento Local\t\tElemento Global\t\tFactor\t\t\tu/factor\t\tRF/factor\n%s\n'
                %(linea[1]))
```

```
        for k in range(num_filas):
                archivo2.write('\n%d\t\t%d\t\t\t %d\t\t\t%e\t\t%e\t\t%e\n%s\n'
                        % (tabla_print[k][0], tabla_print[k][1], tabla_print[k][2], tabla_print[k][3],
                                U2/tabla_print[k][3], RF2/tabla_print[k][3], linea[1]))
        archivo2.close()

        #Fichero reacciones y desplazamientos
        archivo3=open('tabla_grafica.txt','a')
        archivo3.write('\n%d\t\t%e\t\t%e\t\t%e\t\t%e\t\t%e'
                % (int(argv[4])+1, RF1, RF2, U1, U2, tabla_ord[0][3] ))
        archivo3.close()

if argv[4] == '0':
creartxt()
grabartxt()
#-----------------------------------------------------------------------------------------------------------------------------------------------------
# Cerrar output databases
odb.close()
```

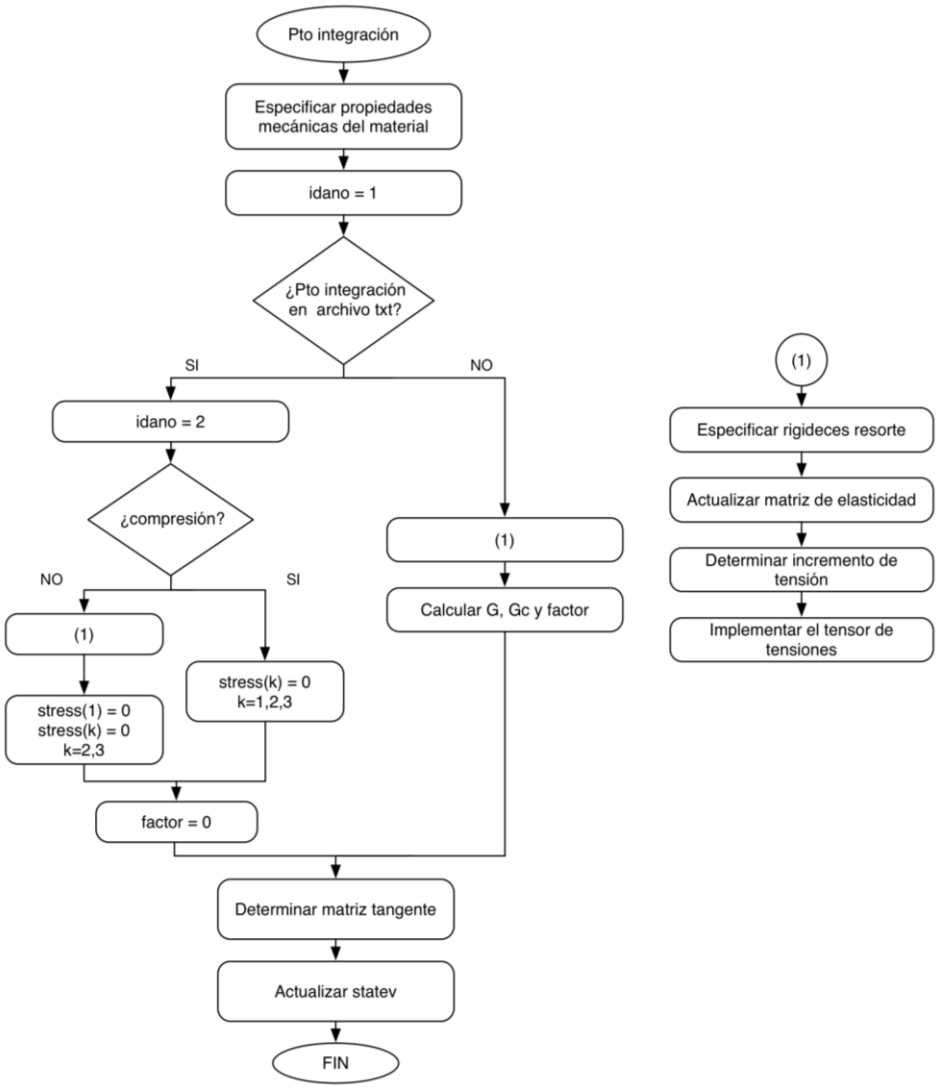
B. ANEXO: SUBRUTINA UMAT MODIFICADA

En este anexo se presenta el código de la subrutina UMAT, el cual implementa el modelo de interfase LEBIM. Fue desarrollado por D.Castillo [6] pero para su uso en este trabajo ha sufrido algunas modificaciones.

Se han incorporado tres variables de estado dependientes (statev), variables que almacenarán información de cada punto de integración. Estas son:

- Idano: contiene un 1 si el punto de integración no está "dañado" y un 2 si sí lo está.
- Factor: raíz cuadrada de Gtot entre Gc
- Noel: elemento global donde se encuentra el punto de integración

También se han añadido unas líneas de código (enmarcadas más abajo) cuyo fin es la lectura del archivo "datos_procesar.txt" que contiene los puntos de integración "dañados". La UMAT se ejecuta una vez por cada punto de integración, por tanto si el punto de integración que se está evaluando aparece en el archivo txt, este debe ser "dañado", por lo que se modifica la variable "idano" perteneciente a este punto de 1 a 2. El funcionamiento de la subrutina modificada se recoge en el siguiente diagrama de flujo.



```
      SUBROUTINE UMAT(STRESS,STATEV,DDSDDE,SSE,SPD,SCD,
     1 RPL,DDSDDT,DRPLDE,DRPLDT,
     2 STRAN,DSTRAN,TIME,DTIME,TEMP,DTEMP,PREDEF,DPRED,CMNAME,
     3 NDI,NSHR,NTENS,NSTATV,PROPS,NPROPS,COORDS,DROT,PNEWDT,
     4 CELENT,DFGRD0,DFGRD1,NOEL,NPT,LAYER,KSPT,KSTEP,KINC)
c
      IMPLICIT NONE
c
      CHARACTER*80 CMNAME
      REAL*8 STRESS(NTENS),
     1 DDSDDE(NTENS,NTENS),
     2 DDSDDT(NTENS),DRPLDE(NTENS),STATEV(nstatv),
     3 STRAN(NTENS),DSTRAN(NTENS),TIME(2),PREDEF(1),DPRED(1),
     4 PROPS(NPROPS),COORDS(3),DROT(3,3),DFGRD0(3,3),DFGRD1(3,3)

      REAL*8 M,N,ID,TOLER
      PARAMETER (M=6,N=6,ID=6,TOLER=1.D-6)
      INTEGER PT,ELEM
      REAL*8 factor
      INTEGER error_ap,error_lec
      REAL*8 DSTRESS(4),DDS(4,4),t,psig,a,tc,lambda
      REAL*8 sigmacb,sigmac,Gi,Gii,Gcorte,thoc,h
      REAL*8 E,xnue,ebulk3,eg2,elam,trval,Gs,Gn,Gt,Knn,Ktt,Kss,K33
      INTEGER k1,k2,k,j,i,ndi,nshr,ntens,nprops,noel,npt
      INTEGER layer,kspt,kstep,kinc,IDANO,nstatv
      REAL*8 sse,spd,scd,rpl,drpldt,dtime,temp,dtemp,celent,pnewdt
      REAL*8 Gc,GIcb,pi,psiGcrit,Gtot,kmuelle
      dds=0.d0
      t=0.d0
      psig=0.d0
      a=0.d0
      tc=0.d0
      lambda = PROPS(1)
      sigmac=0.d0
      thoc=0.d0
      sigmacb = PROPS(2)
      Gi=0.d0
      Gii=0.d0

c     Especificacion de las propiedades mecanicas del material
      h=PROPS(5)
      kmuelle = PROPS(3)
      GIcb = (sigmacb**2)/(2*h*kmuelle)
      pi = 4.d00*DATAN(1.D00)

      IDANO=1
      OPEN (7,FILE='C:\archivos_procesar\datos_procesar.txt',
     +      STATUS='OLD', ACTION='READ', IOSTAT=error_ap)
      error_lec=0
      DO WHILE ((error_ap.eq.0).and.(error_lec.eq.0))
        READ (7,*,IOSTAT=error_lec) PT, ELEM
        IF ((PT.eq.NPT).and.(ELEM.eq.NOEL)) THEN
            idano=2
            EXIT
        ENDIF
      ENDDO
      CLOSE(7)
```

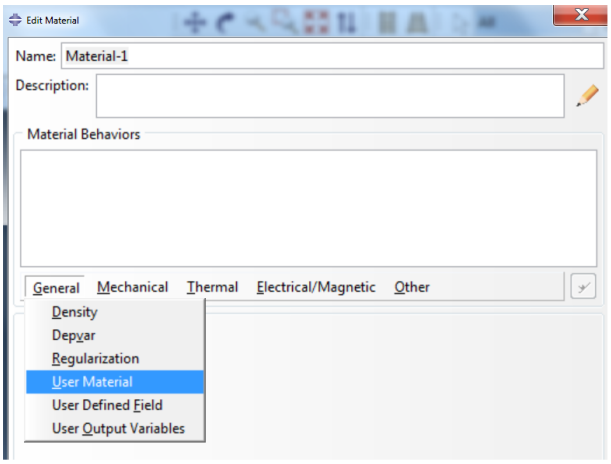
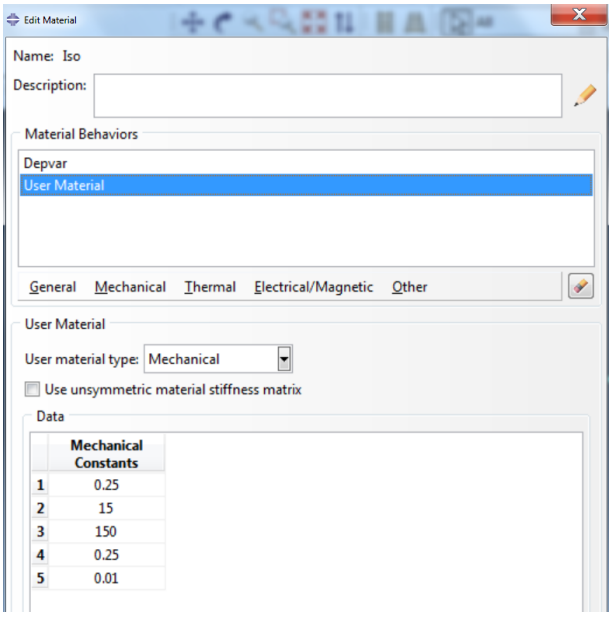
```
c     Se entra en el bucle principal
      IF (idano.eq.1) THEN

c     Rigideces del resorte
      Knn=h*kmuelle
      Kss=Knn/1d18
      Ktt = PROPS(4)*Knn
      K33=Knn/1d18


c     Actualizacion de la matriz de elasticidad
      DDS(1,1)=Knn
      DDS(2,2)=Kss
      DDS(3,3)=K33
      DDS(4,4)=Ktt
c     DDS(1,1) es la radial y DDS(2,2) es en la direccion teta

c     Determination del incremento del tensor de tensión
      DO k=1,4
         DSTRESS(k)=DDS(k,k)*DSTRAN(k)
      ENDDO

c     Implementación del tensor de tensión
      DO k=1,4
         STRESS(k)=STRESS(k)+DSTRESS(k)
      ENDDO

c     Cálculo de t, Gi, Gii, psig
      t=((STRESS(1))**2+(STRESS(4))**2)**(0.5d0)
      Gi=(STRESS(1))**2/(2*Knn)
      Gii=(STRESS(4))**2/(2*Ktt)
      psig=datan2(STRESS(4)*dsqrt(Knn/Ktt),STRESS(1))
      IF (STRESS(1).GT.0.D0)THEN
      Gtot=Gi+Gii
      ELSE
      Gtot=Gii
      ENDIF
      Gc=GIcb*(1+(dtan(psig*(1-lambda)))**2)
      psiGcrit=pi/(2*(1-lambda))
      IF(abs(psig).ge.psiGcrit) Gc=GIcb*1.d8
      factor = dsqrt(Gtot/Gc)
      ENDIF


      IF (idano.eq.2) THEN
            IF (DSTRAN(1).lt.0) THEN
                  Knn=h*kmuelle
                  Ktt=Knn/1d18
                  Kss=Knn/1d18
                  K33=Knn/1d18

                  DDS(1,1)=Knn
                  DDS(2,2)=Kss
                  DDS(3,3)=K33
                  DDS(4,4)=Ktt

                  STRESS(1)=DDS(1,1)*DSTRAN(1)
                   DO k=2,4
                     STRESS(k)=0.d0
                   ENDDO
```

```
            ELSE
                  Knn=h*kmuelle/1d18
                  Ktt=Knn/1d18
                  Kss=Knn/1d18
                  K33=Knn/1d18

                  DDS(1,1)=Knn
                  DDS(2,2)=Kss
                  DDS(3,3)=K33
                  DDS(4,4)=Ktt

                  DO k=1,4
                    STRESS(k)=0.d0
                  ENDDO
            ENDIF
            factor=0.d0
      ENDIF

c     Determinación de la matriz TANGENTE
      DO i=1,4
         DO j=1,4
           DDSDDE(i,j)=DDS(i,j)
         ENDDO
      ENDDO

      statev(1) = idano
      statev(2) = factor
      statev(3) = noel

      RETURN
      END
```

C. ANEXO: CREACIÓN ARCHIVO INP. CONSIDERACIONES

A la hora de desarrollar el modelo en Abaqus, se debe tener en cuenta las siguientes consideraciones, para poder usar el archivo INP generado junto con el código que plantea este trabajo.

<table>
<tr>
<td>Crear material cuyo comportamiento sea el modelado por la subrutina UMAT

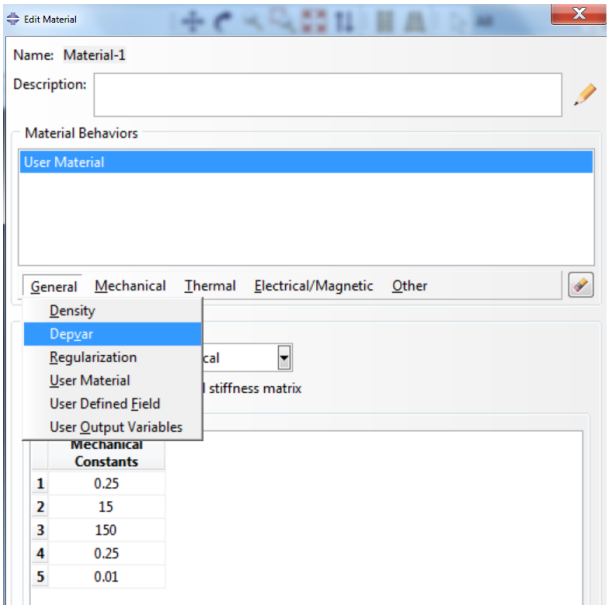
Edit Material/General/User Material</td>
<td> 
</td>
</tr>
<tr>
<td>Propiedades de la interfase
Lambda = 0.25
sigmacb = 15MPa
Kmuelle = 150 GPa/m
ktt/knn = 0.25
h = 0.01 mm</td>
<td> 
</td>
</tr>
</table>

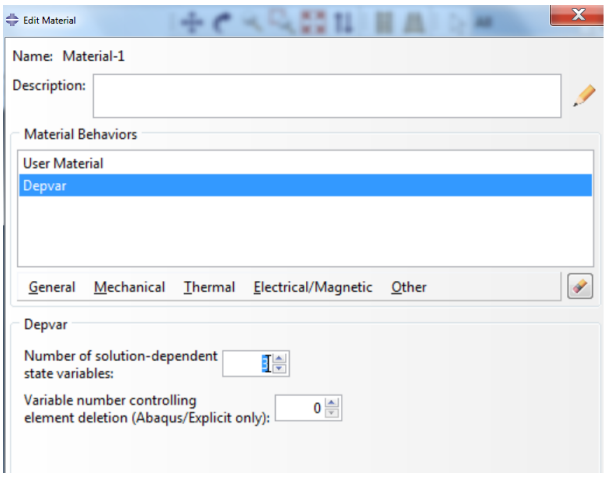
DEPVAR

Edit Material/General/Depvar

Esta variable equivale a la variable STATEV en Fortran explicada en el Anexo B. A través de ellas se intercambia información entre el código Fortran y Abaqus. En el postproceso son utilizadas para extraer información de los puntos de integración.

Como se comentó se utilizan tres de estas variables (idano, factor, noel). Por tanto se deberá poner un tres en la opción 'Number of solution-dependent state variables'

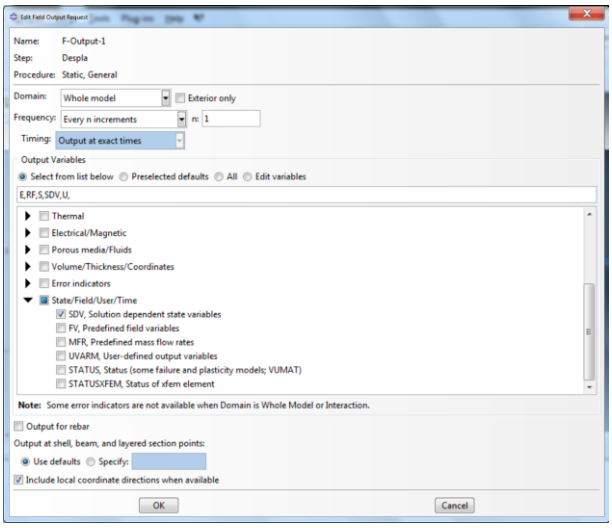


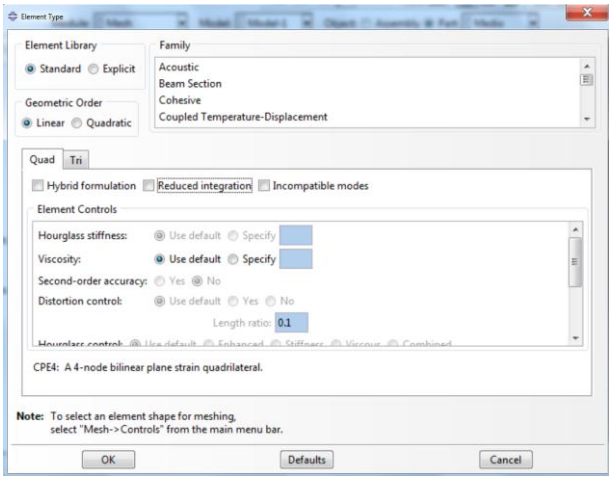
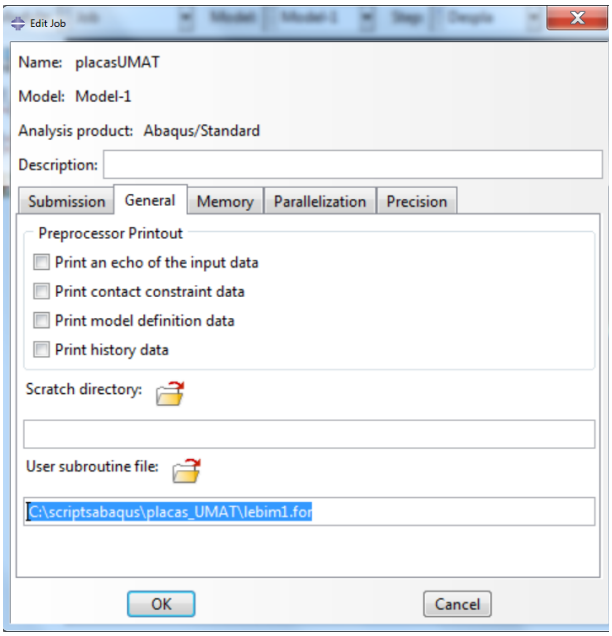


FieldOutputs

Edit Field Output Request/'State/Field/User/Time'

Para poder visualizar las variables SDV posteriormente en los resultados, se deberá seleccionar en 'Output Variables'



Reduced integration *Mesh/Element Type/Reduced integration* A la hora de realizar el mallado se deberá deshabilitar la opción 'Reduced integration'	
Subrutina UMAT *Job/Edit Job/General/User subroutine file* Click en la carpeta y seleccionar el archivo fortran que contenga la subrutina UMAT o bien poner directamente la ruta absoluta de dicho archivo	

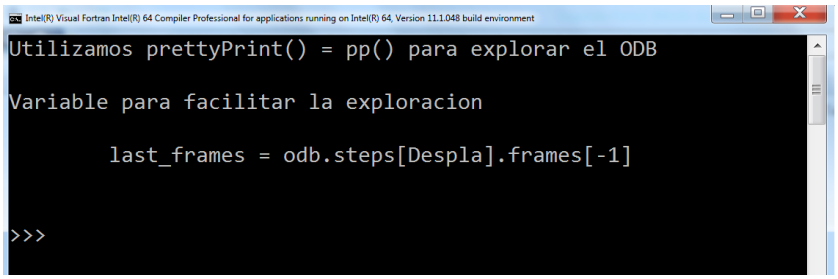
D. ANEXO: SCRIPT EXPLORAR ODB

En este anexo se muestra el código fuente del script desarrollado por el autor llamado "explorarOdb.py". Con él se pretende que el lector pueda navegar a través del archivo ODB y conozca la estructura interna de este, localizando la ubicación exacta de la información deseada.

Este archivo debe ser ejecutado de la siguiente manera:

1. Abrir consola del sistema (CMD)
2. Acceder a la ruta donde se localice el archivo 'explorarOdb.py'
3. Ejecutar la línea: 'abaqus python'. Tras ello deberá aparecer el símbolo '>>>'
4. Ejecutar la línea: execfile('name.odb'), donde 'name' es el nombre del archivo ODB que queramos explorar

Una vez realizado estos pasos, deberá de mostrarse por pantalla lo mismo que en la siguiente figura.



A través del método pp(), se podrá ir explorando el ODB y ver lo que contiene cada repositorio. Se ha creado una variable llamada 'last_frames' para facilitar la exploración (last_frames = odb.steps[name_step].frames[-1]).

Tras su uso recordar cerrar el ODB con la instrucción "odb.close()".

```
# ******************************************************************************
# Explorar ODB
# Autor: Enrique Paloma Castro, GERM (2016)
# ******************************************************************************
#Introducir por teclado ruta del archivo ODB a explorar
path_Odb = raw_input("Path ODB: ")
name_step = raw_input("Name step: ")

#Borrar pantalla
from os import system
system('cls')

#Para poder utilizar prettyPrint(), el cual para facilitar su uso lo hemos cambiado por "pp"
from textRepr import prettyPrint as pp
print ("Utilizamos prettyPrint() = pp() para explorar el ODB\n")
```

```
#Abrimos odb
from odbAccess import*
odb = openOdb(path_Odb)

#Variable para facilitar la exploracion
last_frames = odb.steps[name_step].frames[-1]
print 'Variable para facilitar la exploracion \n\n\tlast_frames =
              odb.steps[%s].frames[-1]\n\n' %name_step

#RECORDAR CERRAR EL ODB DESPUES DE EXPLORAR CON -> odb.close()
```

E. ANEXO: SCRIPTS SCREENSHOTS

Una vez finalizado el análisis, se tiene un determinado número de archivos ODB de los que interesa realizar screenshots de los resultados. Para realizar esta tarea automáticamente se han desarrollado los dos scripts que se presentan a continuación, 'screenshots_ejecutar.py' y 'screenshots_editar.py'.

La forma de usarlo es la siguiente:

1. Se deben situar ambos scripts en el directorio donde se quiera crear la carpeta 'Screenshots' que contenga los screenshots de los resultados.
2. Modificar el script 'screenshots_editar.py'. Líneas de código en negrita.
    a. for k in range(a,b): Dar valores a los parámetros del método range() (a y b), según sea el número de archivos ODB a los que se le quiera realizar los screenshots.
    b. Path: esta variable contiene la ruta absoluta de los archivos ODB. Estos archivos siguen un patrón, solo varia un número entre ellos, lo cual se indica con 'str(k)'. Para aclarar este paso se muestra a continuación un ejemplo.

Ejemplo. *Screenshots de los 200 archivos ODB situados en la carpeta 'placasUMAT_200steps_1paso_1e-3tolerancia'*

Para recorrer los 200 archivos, los parámetros del método range() deben ser: a=1, b=201; quedando la línea de código de la siguiente manera: 'for k in range(1,201):'

Los archivos tienen los siguientes nombres:

$$placasUMAT\_200steps\_\mathbf{1}\_1e{-}3tol.odb$$
$$placasUMAT\_200steps\_\mathbf{2}\_1e{-}3tol.odb$$
$$\vdots$$
$$placasUMAT\_200steps\_\mathbf{200}\_1e{-}3tol.odb$$

El número que varía lo cambiamos por: "+ str(k) +", quedando:

placasUMAT_200steps_+ str(k) +_1e-3tol.odb'

Indicando la ruta absoluta, la variable 'path' en el caso particular del desarrollador quedaría:

path = 'C:/scriptsabaqus/placas_UMAT/placasUMAT_200steps_1paso_1e-3tolerancia/placasUMAT_200steps_' + str(k) + '_1e-3tol.odb'

```
# ******************************************************************************
# SCRIPT: SCREENSHOTS_EJECUTAR.PY
# Autor: Enrique Paloma Castro, GERM (2016)
# ******************************************************************************
#-----------------------------------------------
# ||||  IMPORTACION DE MODULOS  ||||
#-----------------------------------------------
from subprocess import call
from os import mkdir
from shutil import move
from glob import glob
#-----------------------------------------------
# Llamar a Abaqus para que ejecute el script
call('abaqus cae noGUI=screenshots_editar.py',shell=True)
#-----------------------------------------------
# ||||        ORGANIZAR ARCHIVOS        ||||
#-----------------------------------------------
mkdir('Screenshots')
mkdir('S22')
mkdir('Idano')
carp='S22'
list_move = ('S22_*', 'idano*')
for files in list_move:
    files_move = glob(files)
    for k in range(len(files_move)):
        move(files_move[k],carp)
    carp='Idano'
move('S22','Screenshots')
move('Idano','Screenshots')
```

```
#-----------------------------------------------

# ***************************************************************************
# SCRIPT: SCREENSHOTS_EDITAR.PY
# Autor: Enrique Paloma Castro
# ***************************************************************************
from abaqus import *
from abaqusConstants import *
session.Viewport(name='Viewport: 1', origin=(0.0, 0.0), width=243.821884155273,height=165.322494506836)
session.viewports['Viewport: 1'].makeCurrent()
session.viewports['Viewport: 1'].maximize()
from caeModules import *
from driverUtils import executeOnCaeStartup
executeOnCaeStartup()
session.viewports['Viewport: 1'].partDisplay.geometryOptions.setValues(referenceRepresentation=ON)

#=============================================================================================================================
# ||||||||||||||||||                                    MODIFICAR                                    ||||||||||||||||||
#=============================================================================================================================
# Metodo range segun el numero de archivos odb
for k in range(1,201):
# Ruta de los archivos odb. Numero que varia cambiarlo por '+ str(k) +'
path = 'C:/scriptsabaqus/placas_UMAT/placasUMAT_200steps_1paso_1e-3tolerancia/placasUMAT_200steps_' + str(k)
              + '_1e-3tol.odb'
#=============================================================================================================================

# Apertura ODB
o1 = session.openOdb(name=path)
session.viewports['Viewport: 1'].setValues(displayedObject=o1)
# Factor de escala
session.viewports['Viewport: 1'].odbDisplay.commonOptions.setValues(
```

```
deformationScaling=UNIFORM, uniformScaleFactor=10)

# Fuente de la leyenda
session.viewports['Viewport: 1'].viewportAnnotationOptions.setValues(
legendFont='-*-verdana-medium-r-normal-*-*-130-*-*-p-*-*-*',
titleFont='-*-verdana-medium-r-normal-*-*-110-*-*-p-*-*-*',
stateFont='-*-verdana-medium-r-normal-*-*-110-*-*-p-*-*-*')

# Coordenadas trial y titulos
session.viewports['Viewport: 1'].viewportAnnotationOptions.setValues(triadPosition=(1, 9), titlePosition=(10, 20))
session.viewports['Viewport: 1'].viewportAnnotationOptions.setValues(statePosition=(10, 12))

# Seleccion variable SDV1
session.viewports['Viewport: 1'].odbDisplay.setPrimaryVariable(
variableLabel='SDV1', outputPosition=INTEGRATION_POINT, )

# Dibujar deformada
session.viewports['Viewport: 1'].odbDisplay.display.setValues(plotState=CONTOURS_ON_DEF)
```

REFERENCIAS

[1] Abaqus-Inc. Abaqus user manual, Version 6.9. Dassault Systems Simulia Corp, Providence, RI, USA; 2010.

[2] Airbus. Carbon Fibre Reinforced Plastics. Determination of fracture toughness energy of bonded joints. Mode I. G1C. Issue 1. AITM 1-0053, 2006.

[3] GI. Barenblatt. The formation of equilibrium cracks during brittle fracture. General ideas and hypotheses. Axially-symmetric cracks. Journal of Applied Mathematics and Mechanics, 23:622-636, 1959.

[4] A. Caporale, F. Luciano, E. Sacco. Micromechanical analysis of interfacial debonding in unidirectional fiber-reinforced composites. Computers and Structures, 84:2200-2211, 2006.

[5] A. Carpinteri. Post-peak and post-bifucartion analysis on cohesive carck propagation. Engineering Fracture Mechanics, 32:265-278, 1989.

[6] D. Castillo. Implementación de un modelo de interfase en el programa de elementos finitos Abaqus. Aplicación a materiales compuestos. PFC; Universidad de Sevilla, 2014.

[7] F. Erdogan. Fracture mechanics of interfases, In: Damage and Failure of Interfases. Balkema Publishers: Rotterdam, 1997.

[8] G. Geymonat, F. Krasucki, S.Lenci. Mathematical analysis of a bonded joint with a soft thin adhesive. Mathematics and Mechanics of Solids, 4:201-225, 1999.

[9] A. Hilleborg, M. Modeer, P.E. Petersson. Analysis of a crack formation and crack growth in concrete by fracture mechanics and finite elements. Cement and Concrete Research, 6:773-782, 1976.

[10] ISO. Fibre-Reinforced plastic composites - Determination of model I interlaminar fracture toughness, Gic, for unidirectionally reinforced materials. ISO 15024, 2001.

[11] A. Klarbring. Derivation of a model of adhesively bonded joints by the asymptotic expansion method. International Journal of Engineering Science, 29:493-512, 1991.

[12] V.I. Kushch, S.V. Shmegea, L. Mishnaevsky Jr. Explicit modeling the progressive interfase damage in fibrous composite: Analytical vs. numerical approach. Composite Science and Technology, 71:989-997, 2011.

[13] V.I. Kushch, S.V. Shmegea, L. Mishnaevsky Jr. Meso cell model of fiber reinfoced composite: Interfase stress statistics and debonding paths. International Journal of Solids and Structures, 45:2758-2784, 2008.

[14] V.I. Kushch, S.V. Shmegea, P. Brondsted, L. Mishnaevsky Jr. Numerical simulation of progressive debonding in fiber reinforced composite under transverse loading. International Journal of Engineering Science, 49:17-29, 2011.

[15] F. Lebon and F. Zaittouni. Asymptotic modelling of interfases taking contact conditions into account: Asymptotic expansions and numerical implementation. International Journal of Engineering Science, 48:111-127, 2010.

[16] S. Lenci. Analysis of a crack at a weak interface. International Journal of Fracture, 108:275-290, 2001.

[17] V. Mantič, L. Távara, A. Blázquez, E. Graciani, F. París. A linear elastic - brittle interface model: Application to the onset and propagation of a fibre-matrix interface crack under biaxial transverse loads, International Journal of Fracture 195 (2015) 15–38.

[18] A. Needleman. A continuum model for void nucleation by inclusion debonding. Journal of Applied Mechanics, 54:525-532, 1987.

[19] G. Puri. Python scripts for Abaqus. Learn by example; 2011.

[20] Python 2.7.11 documentation.

[21] M. Romanowicz. Progressive failure analysis of unidirectional fiber-reinforced polymers with inhomogeneous interphase and randomly distribute fibers under transverse tensile loading. Composite Part A: appalied science and manufacturing, 41:1829-1838, 2010.

[22] L. Távara. Damage initiation and propagation in composite materials. Boundary element analysis using weak interfase and cohesive zone models. PhD Thesis. Universidad de Sevilla: Sevilla 2010.

[23] L. Távara, V. Mantič, E. Graciani, F. París. BEM analysis of crack onset and propagation along fibre-matrix interfase under transverse tension using a linear elastic-brittle interfase model. Engineering Analysis with Boundary Elements, 35: 207-202, 2011.

[24] L. Távara, V. Mantič, E. Graciani, F. París. BEM modelling of interfase cracks in a group of fibres under biaxial transverse loads. Advances in Boundary Element Techniques XIV. 311-316, 2013.

[25] L. Távara, V. Mantič, E. Graciani, J. Cañas, and F. París. Analysis of a crack in a thin adhesive layer between orthotropic materials. An application to composite interlaminar fracture toughness test. Computer Modeling in Engineering and Sciences, 58:247-270, 2010.

[26] L. Távara. J. Reinoso, D. Castillo, V. Mantič. Mixed-mode failure of interfaces studied by the 2D Linear Elastic-Brittle Interface Model: macro- and micro-mechanical finite element applications to composites.