Diferenciación automática de fuerzas en la integración implícita de sistemas multicuerpo

A. Callejo, J. García de Jalón, A. F. Hidalgo

Instituto Universitario de Investigación del Automóvil (INSIA)
Universidad Politécnica de Madrid (UPM)
a.callejo@upm.es

Resumen

La diferenciación automática es una herramienta informático-matemática muy potente para calcular cualquier tipo de derivadas de funciones. Entre sus ventajas respecto a otras formas de calcular derivadas están su precisión, su eficiencia y su sencillez de implementación. Este artículo trata sobre la implementación de esta técnica en un algoritmo de simulación dinámica de sistemas multicuerpo. Concretamente, se aplica a un integrador implícito que resuelve las ecuaciones diferenciales del movimiento planteadas de forma semi-recursiva. Para ello, se ha elegido la librería ADOL-C, que trabaja por sobrecarga de operadores. Los fundamentos de la diferenciación automática, así como la base matemática de la formulación y del integrador implícito, son explicados con cierto detalle. Finalmente, se comparan la diferenciación automática y la diferenciación numérica desde el punto de vista de la eficiencia, y se estudia la influencia que tiene el tamaño del sistema multicuerpo en los tiempos de cálculo de la simulación. En suma, se ofrece una interesante perspectiva para la optimización del cálculo de sistemas multicuerpo y el desarrollo de algoritmos eficientes, así como una posible implementación de la diferenciación automática en el campo de los sistemas multicuerpo.

INTRODUCCIÓN

Desde el punto de vista de la topología, las dos grandes familias de sistemas multicuerpo son los mecanismos de cadena abierta y los mecanismos de cadena cerrada. Existe una gran variedad de métodos para el cálculo dinámico de ambas familias, que generalmente se clasifican en métodos globales y métodos topológicos. Los segundos son mucho más eficientes y también más complejos, porque aprovechan la topología del sistema para reducir el tamaño de los problemas matemáticos.

El fin de este artículo es mejorar el integrador implícito de una formulación topológica semi-recursiva, válida tanto para mecanismos de cadena abierta como de cadena cerrada. En la citada formulación, la etapa más costosa en términos de CPU es el cálculo numérico de la matriz Jacobiana de las fuerzas, que se lleva más del 95% del tiempo de integración [1]. Merece la pena, por tanto, explorar formas alternativas de calcular esta Jacobiana de forma más precisa y eficiente, y sin disminuir la versatilidad de la implementación.

Existen diversas formas de obtener la derivada de una función. La primera de ellas es obtener la derivada analíticamente, de acuerdo con la definición de la función y empleando las fórmulas del cálculo diferencial. En computación, esta técnica se aplica manualmente o por medio de las herramientas de diferenciación simbólica, que generan código a partir de las expresiones simbólicas. Así pues, en el desarrollo de una aplicación para la solución de un problema determinado, se debe generar el código que calcula la derivada y compilarlo cada vez que se introduce una modificación.

Otra forma de calcular las derivadas es utilizar técnicas de diferenciación numérica como las diferencias finitas (avanzadas, centradas, retrasadas). En el caso de una función escalar f que sólo depende de una variable x, se puede aproximar la derivada llevando a cabo un desarrollo en serie de Taylor en torno al punto x_0, truncando la serie más allá del término lineal y despejando el valor de la derivada:

$$f'(x_0) = \frac{df}{dx}(x_0) \approx \frac{f(x_0 + h) - f(x_0)}{h} \quad (1)$$

Utilizando los valores de la función en más puntos se pueden obtener fórmulas mucho más precisas. Estos métodos tienen la ventaja de que sólo necesitan el programa o función original. La aproximación de una derivada mediante un cociente de diferencias como en la expresión (1) exige que h sea muy pequeño. Sin embargo, cuando h es pequeño, en el numerador de (1) se están restando números muy parecidos, y, debido a la limitada precisión de los ordenadores, las derivadas se obtienen con mucha menos precisión que la función original. Estos errores numéricos son inevitables. Además, con funciones vectoriales (m funciones de n variables), el coste del cálculo de las Jacobianas se incrementa rápidamente con las dimensiones del problema.

La diferenciación automática (*automatic differentiation*, en adelante AD) permite obtener la derivada de una función definida mediante un archivo de código (en Fortran, C/C++, MATLAB, etc.), por compleja o larga que ésta sea, sin pérdida de precisión y en un tiempo de computación razonable [2]. Además, no sólo permite calcular la primera derivada, sino también derivadas de orden superior, gradientes, Jacobianas y Hessianas. La AD tiene un tiempo de desarrollo menor que la derivación manual y una precisión mayor que la numérica.

FUNDAMENTOS DE LA DIFERENCIACIÓN AUTOMÁTICA

La AD se basa en la descomposición de la función informática en operaciones aritméticas (suma, resta, producto, división, etc.) y llamadas a funciones matemáticas elementales (seno, exponencial, etc.), y en la aplicación sistemática de la regla de diferenciación de la cadena. Gracias a esta regla, cada función puede operarse por separado y manejar sus derivadas de forma independiente.

Concepto

Una forma muy común de representar la secuencia de cálculos que realiza el ordenador para evaluar una función son los *árboles computacionales*. Para entender la forma en que trabaja la AD, es útil representar el árbol computacional de la función que se desea derivar. Considérese como ejemplo la siguiente función escalar:

$$y \equiv f(x_1, x_2) = \text{sen}(x_1 \cdot x_2) + 3x_2 \quad (2)$$

Esta función se puede reescribir de forma que el resultado de cada operación aritmética o función elemental se almacene en una nueva variable intermedia x_j, con un índice j superior al de las variables de las que depende:

$$\begin{aligned} x_3 &= x_1 \cdot x_2 \\ x_4 &= \text{sen}(x_3) \\ x_5 &= 3x_2 \\ y &= x_4 + x_5 \end{aligned} \quad (3)$$

El criterio adoptado para nombrar las variables es el siguiente:

- $x_1, x_2, ..., x_n$ son las variables independientes.
- $x_{n+1}, x_{n+2}, ..., x_N$ son las variables intermedias utilizadas para calcular las variables dependientes.
- $y_1 = x_{N+1}, y_2 = x_{N+2}, ..., y_m = x_{N+m}$ son las variables dependientes que devuelve la función.
- $x_{j,\, j>n}$ sólo depende de variables con un índice inferior a j.

La Fig. (1) muestra una posible representación del árbol computacional del ejemplo presentado. Los vértices inferiores están asociados con las variables independientes, el vértice superior con la variable dependiente, y los vértices intermedios con las variables intermedias. Los arcos que unen los vértices representan la dirección del flujo de datos. Se adjunta a cada arco el valor de la derivada parcial del vértice al final del arco respecto al vértice al comienzo del arco.

Se puede calcular la derivada de la variable dependiente y respecto a x_1 aplicando la regla de la cadena sistemáticamente desde los vértices inferiores hacia el vértice superior, como muestra la Ec. (4). El operador $d(.)/d(.)$ denota derivadas absolutas, mientras que el operador $\partial(.)/\partial(.)$ denota derivadas parciales.

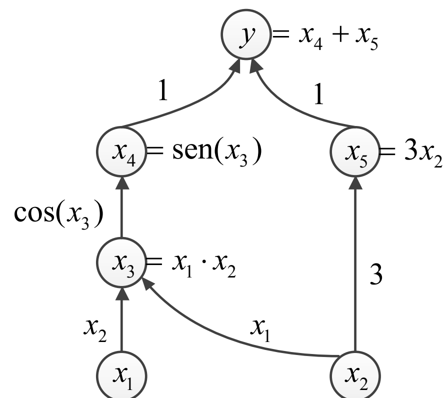

Fig. 1. Árbol computacional de $y = \mathrm{sen}(x_1 \cdot x_2) + 3x_2$.

En cada vértice, la derivada de su correspondiente variable es la suma de las contribuciones de los arcos entrantes. Cada arco aporta la derivada total del vértice en el inicio del arco por la derivada parcial asociada al arco. El cálculo de las derivadas puede realizarse, por tanto, a la vez que la evaluación de la función.

$$\begin{aligned}
&\frac{dx_3}{dx_1} = x_2 \\
&\frac{dx_4}{dx_1} = \frac{\partial x_4}{\partial x_3}\frac{dx_3}{dx_1} = \cos(x_3)\cdot x_2 = \cos(x_1 \cdot x_2)\cdot x_2 \\
&\frac{dy}{dx_1} = \frac{\partial y}{\partial x_4}\frac{dx_4}{dx_1} + \frac{\partial y}{\partial x_5}\frac{dx_5}{dx_1} + \frac{\partial y}{\partial x_1} = 1\cdot\frac{dx_4}{dx_1} + 0 + 0 = \cos(x_1 \cdot x_2)\cdot x_2
\end{aligned} \tag{4}$$

Todas las técnicas de AD operan con funciones que se definen mediante código informático. En forma general, la función informática **f** puede contener todos los elementos habituales de cualquier lenguaje de programación: expresiones matemáticas, bifurcaciones, bucles, llamadas a funciones de librería y a funciones de usuario, llamadas recursivas a la propia función, etc. Sea cual sea la forma en que esté programada, siempre parte de unas variables independientes $\mathbf{x}$ y termina calculando otras variables dependientes $\mathbf{y} \equiv \mathbf{f}(\mathbf{x})$, pasando por ciertas operaciones intermedias. Así pues, la función se puede descomponer en una secuencia discreta de operaciones aritméticas elementales (suma, resta, producto, división, etc.) y llamadas a funciones de librería (seno, exponencial, etc.). De cualquier forma, sus valores y los de sus derivadas pueden propagarse hasta obtener el resultado final. Esto es precisamente lo que hace la AD.

De lo anterior se desprende que la AD es muy versátil, ya que es capaz de trabajar con funciones complicadas, funciones definidas a trozos, sentencias condicionales, bucles, llamadas a sub-funciones, etc. Este tipo de sentencias no suponen ninguna dificultad para la AD porque el algoritmo evalúa la función en cada punto y a la vez calcula la derivada en él.

Tipos

Existen dos formas principales de implementar la AD, ambas con el mismo fundamento teórico. El método de la "sobrecarga de operadores" (*operator overloading*) transforma las variables originales de la función (por ejemplo de tipo *double*) en una estructura cuyas variables miembro son el valor original de la variable y el valor de su derivada o derivadas respecto a las variables independientes. Las operaciones aritméticas elementales y las funciones de librería se sobrecargan para que trabajen con estas estructuras y calculen tanto el resultado de la operación como la derivada de dicho resultado, siguiendo las reglas de derivación. El programa original permanece sin cambios, ya que todas estas sustituciones de variables, operadores y funciones se hacen en tiempo de ejecución. Por ello la implementación de la AD por esta vía es muy versátil y es válida prácticamente para cualquier tipo de función. Como desventaja, la ejecución puede ser lenta por el coste de la sobrecarga de operadores y de funciones.

Por otra parte, el método de la "transformación de código" (*source transformation*) funciona generando una nueva función de código que ya contiene las nuevas variables correspondientes a las derivadas y las sentencias para calcularlas. Esta nueva función debe ser llamada por el programa que necesita las derivadas, y para ello hay

que compilarla y crear un ejecutable. La ventaja frente a la sobrecarga de operadores es que la ejecución es mucho más rápida. La desventaja es que supone una mayor transformación de los programas, que pueden llegar a ser extremadamente complejos, y que muchas herramientas de AD no están lo suficientemente desarrolladas para generar código correcto a partir de funciones informáticas de gran complejidad.

En este artículo se ha considerado solamente la AD por sobrecarga de operadores (por medio de la librería ADOL-C) porque la función a derivar es complicada y se deseaba versatilidad en el desarrollo y en las posteriores modificaciones de la función. Por tanto, cabe considerar que los resultados sobre eficiencia presentados en los últimos apartados constituyen tan sólo un límite inferior de las mejoras esperables con otras herramientas de transformación de código tales como TAPENADE o ADIC.

A continuación se muestra un pequeño ejemplo de cómo procede la AD mediante sobrecarga de operadores. Supóngase que la función a derivar es, una vez más, $y = \mathrm{sen}(x_1 \cdot x_2) + 3x_2$. La función contiene cuatro operaciones o funciones elementales: dos productos, una llamada a la función seno y una suma. Para que el operador producto calcule, además del valor, la derivada de dicha operación en el punto, es necesario sobrecargarlo de la siguiente forma:

```
function g = mtimes(a,b);
g.x  = a.x * b.x;
g.dx = a.x * b.dx + b.x * a.dx;
```

Las variables 'g', 'a' y 'b' son estructuras que contienen dos campos: 'x' y 'dx'. El campo 'x' corresponde al valor de la variable, mientras que el campo 'dx' corresponde al valor de la derivada. Lo mismo habría que hacer con el operador suma y con la función de librería seno. De este modo se consigue que, de modo transparente para el usuario, al evaluar la función original $y(x_1,x_2)$, se evalúen simultáneamente dy/dx_1 y dy/dx_2.

Modos, precisión y eficiencia

Dependiendo de cómo se calcula la derivada en relación al árbol computacional, aparecen dos modos de AD: *forward* (hacia adelante) y *reverse* (hacia atrás). En el modo *forward*, la regla de la cadena se evalúa de las entradas a las salidas, y la carga computacional crece con el número de entradas (variables independientes). En el modo *reverse*, la regla de la cadena se aplica de las salidas a las entradas, y la carga computacional crece con el número de salidas (variables dependientes). Lo explicado en el primer apartado de esta sección corresponde al modo *forward*.

Con ambos modos, que tienen un desarrollo matemático algo distinto, se llega al mismo resultado. Sin embargo, al finalizar el modo *reverse*, se dispone además de las derivadas parciales de las variables dependientes respecto a las variables intermedias. Esto es de gran ayuda cuando se tienen varias variables independientes y se desean hallar las derivadas parciales, ya que sólo se tiene que ejecutar el método una vez. Sin embargo, el modo *reverse* necesita mucha más memoria, ya que ha de ir guardando todos los resultados parciales del árbol computacional. Debido a que en la aplicación que se presenta en este artículo no se necesitan derivadas parciales respecto a las variables intermedias y a que los problemas numéricos considerados requieren mucha memoria por ser de gran tamaño, se ha utilizado el modo *forward*. Para más información a este respecto, véase la referencia [2].

En cuanto a la precisión, dado que la AD está basada en simples operaciones aritméticas, su precisión básica es igual a la precisión de la máquina y del algoritmo considerado, y los métodos de AD no introducen error por sí mismos. El coste computacional del modo *forward* es, a priori, del mismo orden de magnitud que con diferencias finitas: evaluar una función y su derivada es por lo general entre dos y dos veces y media más costoso que evaluar la función únicamente.

$$COST(f, f') \leq [2, 2.5]\, COST(f) \tag{5}$$

En el modo *reverse*, sin embargo, evaluar la función y su derivada tiene un coste computacional entre tres y cinco veces mayor que evaluar la función únicamente.

$$COST(f, f') \leq [3, 5]\, COST(f) \tag{6}$$

En general, las características del modo *forward* lo hacen preferible si en la aplicación hay muchas más variables dependientes que independientes. También hay que tener en cuenta que los requerimientos de memoria en el modo *forward* son aproximadamente lineales con el número de variables independientes, mientras que en el modo *reverse* los requerimientos de memoria son mucho mayores.

En el campo de la dinámica de sistemas multicuerpo, la AD se ha utilizado sobre todo en optimización y análisis de sensibilidad. Los autores coinciden en su versatilidad, es decir, en la capacidad de generar las derivadas de funciones arbitrariamente complejas. También coinciden en resaltar que la precisión obtenida es igual a la precisión de la máquina, por contraste con el método menos exacto de la diferenciación numérica. De igual forma, están de acuerdo en que el tiempo de desarrollo del código es menor que con derivación manual y simbólica, aunque mayor que con la diferenciación numérica.

Sin embargo, hay dos aspectos sobre los que no hay acuerdo en la literatura. El primero es la eficiencia numérica de la función que calcula la derivada. Aunque está generalmente aceptado que la AD es más rápida que la diferenciación numérica, en determinadas aplicaciones puede no ser así [3]. El segundo es la fiabilidad del código en el caso de programas muy complejos [4]. Desde el punto de vista de los autores de este artículo, parte de esos desacuerdos provienen de tomar las características de una determinada herramienta de AD como generales o, al menos, de no considerar la naturaleza de esas herramientas (transformación de código, sobrecarga de operadores, modos *forward* o *reverse*) para valorar los resultados. Por este motivo, en este artículo se han intentado contextualizar con claridad las conclusiones sobre la AD.

FORMULACIÓN TOPOLÓGICA

La formulación multicuerpo en la que se basa este artículo es una formulación topológica semi-recursiva de conocida eficiencia [5]-[6]. Una buena combinación de características que ofrece esta formulación es la de integrar las ecuaciones diferenciales del movimiento de forma explícita y en coordenadas independientes. Sin embargo, como ya se ha dicho, en este artículo se desea combinar la formulación topológica con un integrador implícito en coordenadas dependientes. El uso de integradores implícitos hace que la integración numérica sea más estable, con lo que se pueden utilizar pasos de integración más grandes.

La citada formulación plantea en primer lugar las ecuaciones diferenciales de cadena abierta y considera el sistema como una estructura de árbol, según muestra la Fig. (2). Como se trata de resolver sistemas de cadena cerrada (en el caso más general), previamente es necesario retirar determinados pares y barras biarticuladas (*rods*) para que el sistema pase a ser de cadena abierta. Dichos pares y barras biarticuladas se incluirán más tarde en la integración implícita en forma de ecuaciones de restricción definidas mediante el vector $\mathbf{\Phi}$.

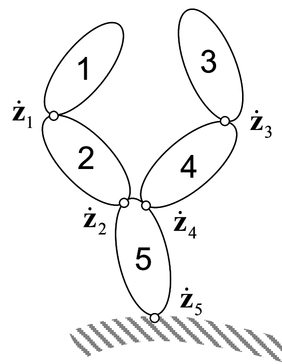

Fig. 2. Sistema multicuerpo en forma de árbol.

Las velocidades y aceleraciones cartesianas de los sólidos se pueden calcular recursivamente a través del árbol de forma muy eficiente si se utiliza como punto de referencia de los sólidos el punto que instantáneamente coincide con el origen de coordenadas global [5]-[6]. De este modo, la matriz de masas $\bar{\mathbf{M}}$ y el vector de fuerzas externas $\bar{\mathbf{Q}}$ están expresados considerando ese punto de referencia que es común para todos los elementos, lo que permite sumarlos o acumularlos sin tener que transformarlos. Además, las ecuaciones del movimiento se pueden expresar en función de las coordenadas relativas $\mathbf{z}$ de los pares, gracias a la introducción de una matriz de transformación de velocidades $\mathbf{R}_d$, que se construye a partir de la topología del sistema (definida en la matriz de accesibilidad $\mathbf{T}$) y de los diferentes tipos de pares presentes en él (pares prismáticos o de revolución y, combinando ambos, cualquier otro tipo de par). Con estas características, la expresión de las ecuaciones diferenciales del movimiento para el sistema de cadena abierta tiene la siguiente forma:

$$\mathbf{R}_d^T\left(\mathbf{T}^T\bar{\mathbf{M}}\mathbf{T}\right)\mathbf{R}_d\ddot{\mathbf{z}} = \mathbf{R}_d^T\mathbf{T}^T\left(\bar{\mathbf{Q}} - \bar{\mathbf{M}}\mathbf{T}\dot{\mathbf{R}}_d\dot{\mathbf{z}}\right) \tag{7}$$

Obsérvese que todavía no ha aparecido el vector de ecuaciones de restricción $\mathbf{\Phi}(\mathbf{z})$, dado que será misión del integrador implícito el hacer cumplir las restricciones de cierre de lazos. Para simplificar la notación de cara al próximo apartado, se pueden agrupar ciertos términos de la ecuación de la siguiente manera:

$$\mathbf{M} \equiv \mathbf{R}_d^T \left(\mathbf{T}^T \bar{\mathbf{M}} \mathbf{T}\right) \mathbf{R}_d, \quad \mathbf{Q} \equiv \mathbf{R}_d^T \mathbf{T}^T \left(\bar{\mathbf{Q}} - \bar{\mathbf{M}} \mathbf{T} \dot{\mathbf{R}}_d \dot{\mathbf{z}}\right) \tag{8}$$

De este modo las ecuaciones diferenciales del movimiento quedan en la forma:

$$\mathbf{M}\ddot{\mathbf{z}} = \mathbf{Q} \tag{9}$$

INTEGRADOR IMPLÍCITO

El integrador implícito que se va a utilizar para integrar las ecuaciones diferenciales del movimiento está basado en la referencia [7], aunque con algunas diferencias [1]. Utiliza penalizadores sólo en posiciones (no en velocidades ni en aceleraciones), así como la regla trapezoidal y proyecciones de velocidades y aceleraciones. Estas últimas son correcciones mediante técnicas algebraicas de proyección sobre subespacios, que hacen cumplir las restricciones de velocidades y aceleraciones que no se han impuesto mediante penalizadores.

Ecuaciones diferenciales y regla trapezoidal

De entre las formas clásicas de escribir las ecuaciones diferenciales del movimiento de sistemas de cadena cerrada, es decir, las ecuaciones diferenciales de un sistema multicuerpo con restricciones [8], se ha escogido un esquema basado en penalizadores en posiciones pero sin multiplicadores de Lagrange, es decir:

$$\mathbf{M}\ddot{\mathbf{z}} + \mathbf{\Phi}_{\mathbf{z}}^T \alpha \mathbf{\Phi} = \mathbf{Q} \tag{10}$$

donde **M** y **Q** son respectivamente las matrices de masas y de fuerzas generalizadas definidas en (8), **z** son las coordenadas relativas dependientes, $\mathbf{\Phi}$ es el vector de ecuaciones de restricción y α es el factor de penalización. Las ecuaciones (10) forman un sistema de n ecuaciones diferenciales con n incógnitas. Para la integración de dichas ecuaciones se plantea el uso de la regla trapezoidal, que muestra un comportamiento adecuado para sistemas de 2º orden como éste. Sus ecuaciones, planteadas para un instante particular $n+1$, son las siguientes:

$$\mathbf{z}_{n+1} = \mathbf{z}_n + \frac{h}{2}(\dot{\mathbf{z}}_n + \dot{\mathbf{z}}_{n+1}) \tag{11}$$

$$\dot{\mathbf{z}}_{n+1} = \dot{\mathbf{z}}_n + \frac{h}{2}(\ddot{\mathbf{z}}_n + \ddot{\mathbf{z}}_{n+1}) \tag{12}$$

Despejando las velocidades y aceleraciones en el instante $n+1$ en función de las posiciones se obtiene:

$$\dot{\mathbf{z}}_{n+1} = \frac{2}{h}\mathbf{z}_{n+1} + \hat{\dot{\mathbf{z}}}_n; \qquad \hat{\dot{\mathbf{z}}}_n = -\left(\frac{2}{h}\mathbf{z}_n + \dot{\mathbf{z}}_n\right) \tag{13}$$

$$\ddot{\mathbf{z}}_{n+1} = \frac{4}{h^2}\mathbf{z}_{n+1} + \hat{\ddot{\mathbf{z}}}_n; \qquad \hat{\ddot{\mathbf{z}}}_n = -\left(\frac{4}{h^2}\mathbf{z}_n + \frac{4}{h}\dot{\mathbf{z}}_n + \ddot{\mathbf{z}}_n\right) \tag{14}$$

donde $\hat{\dot{\mathbf{z}}}_n$ y $\hat{\ddot{\mathbf{z}}}_n$ son los llamados términos "históricos", que dependen sólo de los instantes anteriores. Combinando estas ecuaciones con las ecuaciones del movimiento (10) evaluadas en el instante $n+1$, se llega al siguiente sistema de ecuaciones no lineales:

$$\mathbf{f}(\mathbf{z}_{n+1}) = \mathbf{M}_{n+1}\mathbf{z}_{n+1} + \frac{h^2}{4}\mathbf{\Phi}_{\mathbf{z}_{n+1}}^T \alpha \mathbf{\Phi}_{n+1} - \frac{h^2}{4}\mathbf{Q}_{n+1} + \frac{h^2}{4}\mathbf{M}_{n+1}\hat{\ddot{\mathbf{z}}}_n = \mathbf{0} \tag{15}$$

donde $\mathbf{M} = \mathbf{M}(\mathbf{z})$, $\mathbf{Q} = \mathbf{Q}(\mathbf{z}, \dot{\mathbf{z}})$, $\mathbf{\Phi} = \mathbf{\Phi}(\mathbf{z})$ y $\mathbf{\Phi}_{\mathbf{z}} = \mathbf{\Phi}_{\mathbf{z}}(\mathbf{z})$.

Obtención de posiciones

Para calcular $\mathbf{z}_{n+1}$ a partir de las ecuaciones no lineales (15), se aplica el método de Newton-Raphson, cuya expresión detallada se omite por ser ampliamente conocida. Para ello es necesario calcular la matriz tangente (Jacobiana) del sistema de ecuaciones no lineales, que en el paso n y en la iteración k tiene la siguiente forma:

$$\left[\frac{\partial \mathbf{f}(\mathbf{z})}{\partial \mathbf{z}}\right]_n^k = \left(\mathbf{M}(\mathbf{z}) + \frac{h}{2}\mathbf{C} + \frac{h^2}{4}\left(\mathbf{\Phi}_{\mathbf{z}}^T \alpha \mathbf{\Phi}_{\mathbf{z}} + \mathbf{K} + \mathbf{\Phi}_{\mathbf{zz}}^T \alpha \mathbf{\Phi}(\mathbf{z}) + \mathbf{M}_{\mathbf{z}}\ddot{\mathbf{z}}\right)\right)_n^k \tag{16}$$

Según [7], esta expresión tiene algunos términos que no merece la pena calcular. El término $\mathbf{\Phi}_{\mathbf{zz}}^T$ es un tensor de tercer orden muy disperso. El término $\mathbf{\Phi}_{\mathbf{zz}}^T \alpha \mathbf{\Phi}(\mathbf{z})$ es casi siempre despreciable frente a $\mathbf{\Phi}_{\mathbf{z}}^T \alpha \mathbf{\Phi}_{\mathbf{z}}$, con una diferencia de varios órdenes de magnitud. Eliminando todos estos términos poco relevantes, queda una matriz tangente aproximada que en la práctica se comporta muy bien:

$$\left[\frac{\partial \mathbf{f}(\mathbf{z})}{\partial \mathbf{z}}\right]_n^k \approx \left(\mathbf{M}(\mathbf{z}) + \frac{h}{2}\mathbf{C} + \frac{h^2}{4}\left(\mathbf{\Phi}_{\mathbf{z}}^T \alpha \mathbf{\Phi}_{\mathbf{z}} + \mathbf{K}\right)\right)_n^k \tag{17}$$

donde:

$$\mathbf{K} = -\frac{\partial \mathbf{Q}}{\partial \mathbf{z}}, \quad \mathbf{C} = -\frac{\partial \mathbf{Q}}{\partial \dot{\mathbf{z}}} \tag{18}$$

La Jacobiana de las ecuaciones no lineales queda expresada, por tanto, en función de otras dos matrices Jacobianas: $\mathbf{K}$ y $\mathbf{C}$. Estas son las Jacobianas que se desea calcular por medio de AD. Contienen las derivadas de las fuerzas respecto a las posiciones relativas y a las velocidades relativas, respectivamente. Las fuerzas **Q** son un conjunto de *n* funciones de *n* variables, o lo que es lo mismo, una aplicación $\mathbb{R}^n \rightarrow \mathbb{R}^n$, siendo por tanto las entradas y las salidas vectores. La primera derivada de una función de este tipo es la matriz Jacobiana, donde cada elemento (*i,j*) es la derivada parcial de la salida Q_i con respecto a las entradas z_j o $\dot{z}_j$:

$$\mathbf{K}_{i,j} = -\frac{\partial Q_i}{\partial z_j}, \quad \mathbf{C}_{i,j} = -\frac{\partial Q_i}{\partial \dot{z}_j} \tag{19}$$

El residuo, para la iteración k del método de Newton-Raphson y un instante de tiempo $(n+1)$ cualquiera, se obtiene directamente de las ecuaciones del movimiento (15):

$$\mathbf{f}(\mathbf{z})\Big|_n^k = \frac{h^2}{4}\left(\mathbf{M}\ddot{\mathbf{z}} + \frac{h^2}{4}\mathbf{\Phi}_{\mathbf{z}}^T \alpha \mathbf{\Phi} - \frac{h^2}{4}\mathbf{Q}(\mathbf{z}, \dot{\mathbf{z}})\right)_n^k \tag{20}$$

En cada paso de tiempo, resolviendo las ecuaciones dinámicas (15) y aplicando las ecuaciones del integrador (13) y (14), se obtienen unas velocidades y aceleraciones "sucias" $\dot{\mathbf{z}}_{n+1}^*$ y $\ddot{\mathbf{z}}_{n+1}^*$, calculadas a partir de la regla trapezoidal y no derivando las posiciones. Como ya se ha hecho notar, las ecuaciones anteriores imponen el equilibrio dinámico y el cumplimiento de las restricciones en posición $\mathbf{\Phi}(\mathbf{z})$, pero no el cumplimiento de las restricciones de velocidad $\dot{\mathbf{\Phi}}(\mathbf{z})$ y aceleración $\ddot{\mathbf{\Phi}}(\mathbf{z})$. Para corregir esta deficiencia se utilizan las proyecciones de velocidades y aceleraciones, que corrigen sus valores de modo que se satisfagan dichas restricciones.

Proyección de velocidades y aceleraciones

Para proyectar las velocidades y aceleraciones, es decir, para hallar $\dot{\mathbf{z}}$ y $\ddot{\mathbf{z}}$ a partir de $\dot{\mathbf{z}}^*$ y $\ddot{\mathbf{z}}^*$ (velocidades y aceleraciones "sucias"), se plantea un esquema de penalización expresado por la siguiente ecuación:

$$\left(\mathbf{M} + \mathbf{\Phi}_{\mathbf{z}}^T \alpha \mathbf{\Phi}_{\mathbf{z}}\right)\dot{\mathbf{z}} = \mathbf{M}\dot{\mathbf{z}}^* - \mathbf{\Phi}_{\mathbf{z}}^T \alpha \mathbf{\Phi}_t \tag{21}$$

En adelante no se desarrollan con profundidad las expresiones por cuestiones de espacio. En los problemas de minimización con restricciones de los métodos de proyección aparece una matriz de peso. Con el objeto de ahorrar tiempo de cálculo, Cuadrado y otros [9] propusieron utilizar una matriz de peso tal que permitiera utilizar la última factorización de la matriz tangente (17) para resolver las proyecciones. Para ello, la matriz de peso es:

$$\mathbf{P} = \mathbf{M}(\mathbf{z}) + \frac{h}{2}\mathbf{C} + \frac{h^2}{4}\mathbf{K} \tag{22}$$

Definiendo la función a minimizar y las restricciones a las que está sujeta, imponiendo la condición de mínimo, igualando a cero y operando, se llega al siguiente sistema de ecuaciones:

$$\left(\mathbf{P} + \frac{h^2}{4}\mathbf{\Phi}_{\mathbf{z}}^T \alpha \mathbf{\Phi}_{\mathbf{z}}\right)\dot{\mathbf{z}} = \mathbf{P}\dot{\mathbf{z}}^* - \frac{h^2}{4}\mathbf{\Phi}_{\mathbf{z}}^T \alpha \mathbf{\Phi}_t \tag{23}$$

Como se ve, la matriz del sistema (23) es la matriz tangente expresada por la ecuación (17) empleada en la resolución de las ecuaciones del movimiento. De esta manera, se puede utilizar su última factorización para obtener las velocidades proyectadas con un bajo coste computacional. Considerando por último que el sistema no tiene restricciones reónomas, se llega a la expresión que permite obtener las velocidades proyectadas:

$$\dot{\mathbf{z}} = \left(\mathbf{P} + \frac{h^2}{4} \mathbf{\Phi}_{\mathbf{z}}^T \alpha \mathbf{\Phi}_{\mathbf{z}} \right)^{-1} \mathbf{P}\dot{\mathbf{z}}^* \tag{24}$$

Con las aceleraciones se procede de la misma manera y se obtiene, mediante las mismas deducciones que con las velocidades, la siguiente expresión de proyección modificada de aceleraciones:

$$\left(\mathbf{P} + \frac{h^2}{4} \mathbf{\Phi}_{\mathbf{z}}^T \alpha \mathbf{\Phi}_{\mathbf{z}} \right) \ddot{\mathbf{z}} = \mathbf{P}\ddot{\mathbf{z}}^* - \frac{h^2}{4} \mathbf{\Phi}_{\mathbf{z}}^T \alpha \dot{\mathbf{\Phi}}_{\mathbf{z}} \dot{\mathbf{z}} - \frac{h^2}{4} \mathbf{\Phi}_{\mathbf{z}}^T \alpha \dot{\mathbf{\Phi}}_t \tag{25}$$

donde, en ausencia de restricciones reónomas, se llega finalmente a la siguiente expresión que permite obtener las aceleraciones proyectadas:

$$\ddot{\mathbf{z}} = \left(\mathbf{P} + \frac{h^2}{4} \mathbf{\Phi}_{\mathbf{z}}^T \alpha \mathbf{\Phi}_{\mathbf{z}} \right)^{-1} \left(\mathbf{P}\ddot{\mathbf{z}}^* - \frac{h^2}{4} \mathbf{\Phi}_{\mathbf{z}}^T \alpha \dot{\mathbf{\Phi}}_{\mathbf{z}} \dot{\mathbf{z}} \right) \tag{26}$$

RESULTADOS

Para poner a prueba la formulación descrita y poder sacar conclusiones sobre su eficiencia, se han llevado a cabo una serie de simulaciones con sistemas de diferentes tamaños (número de sólidos y grados de libertad), realizando el cálculo de la matriz Jacobiana (17) mediante AD y mediante diferenciación numérica.

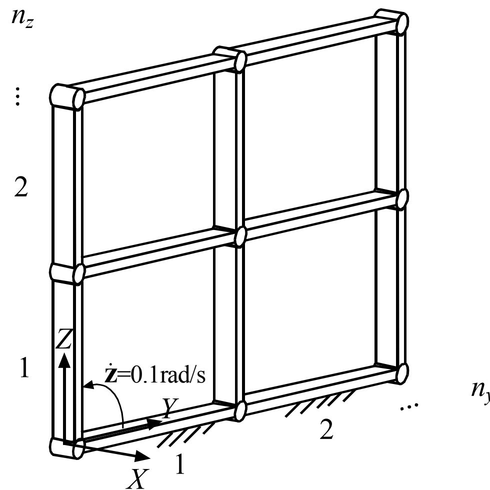

Fig. 3. Cuadrilátero articulado múltiple.

El sistema considerado es un cuadrilátero articulado múltiple, compuesto por cuadriláteros articulados en serie en dirección *Y* y en paralelo en dirección *Z* (ver Fig. (3)). El número de cuadriláteros en dirección *Y* se denota como n_y, y el número de cuadriláteros en *Z* se denota como n_z. Este último número coincide con el número de grados de libertad del sistema. En cuanto al número de sólidos, es igual a $(1+2n_y)n_z+1$, incluyendo el sólido fijo. Las barras tienen un peso de 1 kg y una longitud de 1 m. Se da una velocidad inicial de 0,1 rad/s al primer grado de libertad del sistema y nula a todos los demás.

Para un tiempo de simulación fijo de 1 s con 200 pasos de integración, el tiempo de cálculo depende del número de grados de libertad (que es igual a n_z) y del número de sólidos rígidos (que depende de n_y). Se han simulado sistemas con diferentes valores de n_z, y para cada uno de ellos se ha variado el valor de n_y, con lo que se ha variado el número de sólidos del sistema. Todas las simulaciones se han llevado a cabo en una plataforma Intel® Core™ i7 870, 2,93 GHz, 6 GB RAM. La Tabla (1) muestra los tiempos de computación obtenidos para las distintas variantes del sistema. La columna 'gdl' especifica el número de grados de libertad, la columna 'sólidos'

el número de sólidos rígidos, la columna ‘n_{jacs}’ el número de Jacobianas calculadas, la columna ‘t_{num}’ el tiempo de simulación utilizando la Jacobiana numérica (con diferencias finitas avanzadas y h=10^{-8}), la columna ‘t_{AD}’ el tiempo de simulación utilizando la Jacobiana con AD y la última columna el cociente entre ambos tiempos. Los tiempos han sido ponderados con el cociente 400/n_{jacs} para que el aumento de iteraciones en el método de Newton-Raphson (y por tanto en el número de Jacobianas calculadas), debido al aumento del tamaño del sistema, no distorsione el cómputo de tiempos.

Tabla 1. Tiempos de simulación del cuadrilátero articulado múltiple.

gdl	sólidos	n_y	n_z	n_{jacs}	t_{num} (s)	t_{AD} (s)	t_{AD}/t_{num}
10	31	1	10	427	0,979	1,505	**1,54**
	51	2		425	3,216	5,153	**1,60**
	71	3		425	7,592	10,28	**1,35**
	91	4		425	14,50	17,41	**1,20**
20	61	1	20	430	4,891	7,415	**1,52**
	101	2		428	18,91	22,53	**1,19**
	141	3		428	47,27	50,44	**1,07**
	181	4		427	97,20	95,82	**0,99**
30	91	1	30	459	13,68	18,15	**1,33**
	151	2		456	56,68	59,90	**1,06**
	211	3		455	150,2	143,9	**0,96**
	271	4		454	307,8	275,2	**0,89**
40	121	1	40	468	29,88	35,43	**1,19**
	201	2		465	130,1	127,3	**0,98**
	281	3		464	343,6	308,8	**0,90**
	361	4		463	861,1	791,8	**0,92**

Los datos de la Tabla (1) pueden representarse según muestra la Fig. (4). Se observa que cuanto mayores son el número de grados de libertad del sistema y el número de sólidos, más eficiente es la utilización de la AD para calcular la matriz Jacobiana, llegando a superar a las diferencias finitas como método de diferenciación.

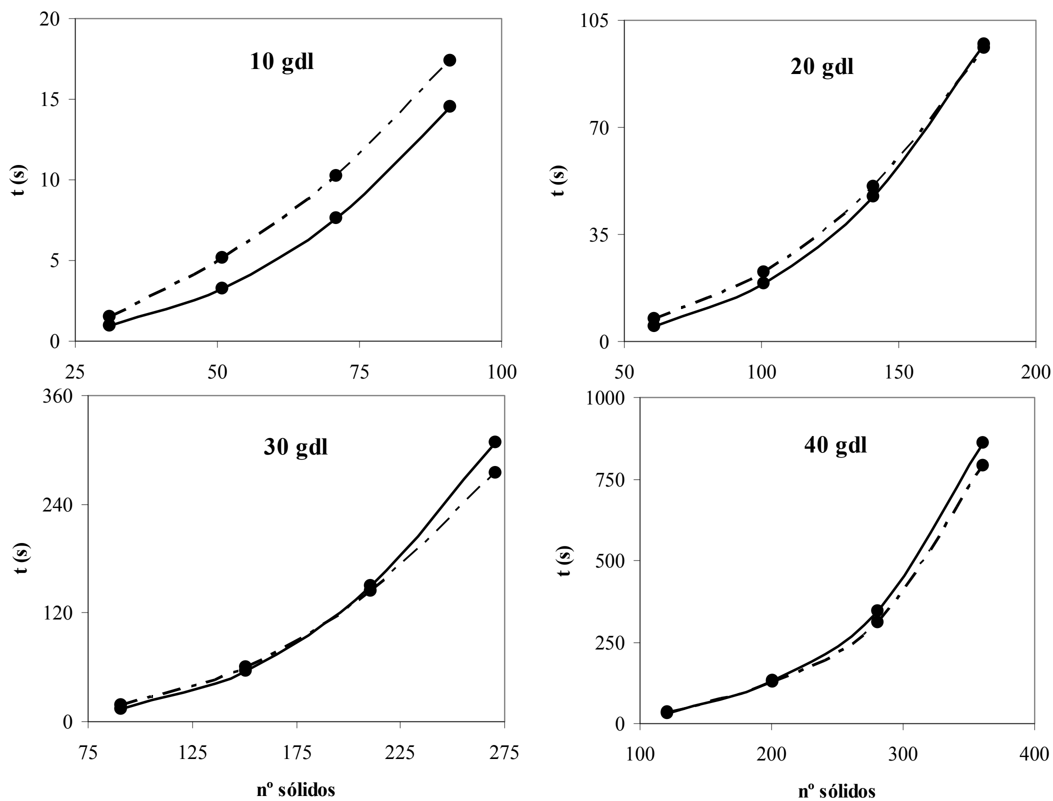

Fig. 4. Representación de los resultados. Línea continua: Jacobiana numérica; línea discontinua: AD.

CONCLUSIONES

La diferenciación automática es una herramienta muy potente y versátil para calcular derivadas de funciones definidas informáticamente. En el campo concreto de la dinámica de sistemas multicuerpo, han sido varios los intentos de implementación de esta técnica, por ejemplo en problemas de optimización y análisis de sensibilidad, pero sin conclusiones claras sobre la conveniencia de su uso.

En este artículo se ha presentado una implementación concreta y original de la diferenciación automática en un integrador implícito que resuelve las ecuaciones diferenciales del movimiento planteadas topológicamente. Para ello se ha utilizado la librería ADOL-C (que trabaja por sobrecarga de operadores) en modo *forward*.

El análisis de tiempos llevado a cabo al final del artículo pone de manifiesto que la eficiencia de la AD crece con el número de grados de libertad y con el número de sólidos rígidos del sistema multicuerpo. También demuestra que a la precisión y versatilidad de la AD no se opone su velocidad de cálculo, ya que para determinados tamaños del sistema es superior a la velocidad de cálculo de la diferenciación numérica. Teniendo en cuenta que la sobrecarga de operadores es más lenta que la transformación de código y por tanto los tiempos presentados son un límite superior de lo que se puede conseguir, la AD se perfila como una herramienta muy útil y eficaz para calcular derivadas en simulación de sistemas multicuerpo.

AGRADECIMIENTOS

Este trabajo ha sido financiado con ayuda de los proyectos SOMAVE (TRA2006-13942), CABINTEC y OPTIVIRTEST (TRA2009-14513) del Ministerio de Ciencia e Innovación; del Departamento de Educación del Gobierno de Navarra; y de la Universidad Politécnica de Madrid.

REFERENCIAS

[1] A. F. Hidalgo, A. Callejo and J. García de Jalón, *Using implicit integrators and automatic differentiation to compute large and complex MBS in real-time*, 1st Joint International Conference on Multibody System Dynamics, Lappeenranta, (2010).

[2] A. Griewank, *Evaluating derivatives - Principles and Techniques of Algorithmic Differentiation*, SIAM, Frontiers in Applied Mathematics, (2000).

[3] P. Eberhard, W. Schiehlen and J. Sierts, *Sensitivity analysis of inertia parameters in Multibody Dynamics simulations*, 12th IFToMM World Congress, Besançon, (2007).

[4] K. S. Anderson and Y. Hsu, *Analytical fully-recursive sensitivity analysis for multibody dynamic chain systems*, Multibody System Dynamics, 8 (2002), 1-27.

[5] J. García de Jalón, E. Álvarez, F. A. de Ribera, I. Rodríguez and F. J. Funes, *A Fast and Simple Semi-Recursive Dynamic Formulation for Multi-Rigid-Body Systems*, in Advances in Computational Multibody Systems, ed. by J. Ambrósio, Springer-Verlag, 1-24, (2005).

[6] J. García de Jalón, A. F. Hidalgo and A. Callejo, *MBS software development with MATLAB for teaching and industrial usages*, ASME International Design Engineering Technical Conferences & Computers and Information in Engineering Conference IDETC/CIE 2009, San Diego, (2009).

[7] D. Dopico, *Formulaciones semi-recursivas y de penalización para la dinámica en tiempo real de sistemas multicuerpo*, Tesis doctoral, Universidade da Coruña, (2004).

[8] J. García de Jalón and E. Bayo, *Kinematic and Dynamic Simulation of Multi-Body Systems – The Real-Time Challenge*, Springer-Verlag, New-York, (1993).

[9] J. Cuadrado, J. Cardenal, P. Morer and E. Bayo, *Intelligent Simulation of Multibody Dynamics: Space-State and Descriptor Methods in Sequential and Parallel Computing Environments*, Mutibody System Dynamics, 4 (2000), 55-73.