Universidad Complutense de Madrid

Universidad Nacional de Educación a Distancia

Máster en Ingeniería de Sistemas y Control

PARTICIÓN, REDUCCIÓN DE ÍNDICE Y TEARING DE MODELOS DE ECUACIONES ALGEBRAICAS DIFERENCIALES

Memoria presentada por

MªÁngeles González Domínguez

Bajo la dirección de

Alfonso Urquía Moraleda

Curso Académico 2014/15

Septiembre 2015

Proyecto Fin de Máster

Máster en Ingeniería de Sistemas y Control

PARTICIÓN, REDUCCIÓN DE ÍNDICE Y TEARING DE MODELOS DE ECUACIONES ALGEBRAICAS DIFERENCIALES

Proyecto tipo B

Proyecto específico propuesto por el alumno

Memoria presentada por

MªÁngeles González Domínguez

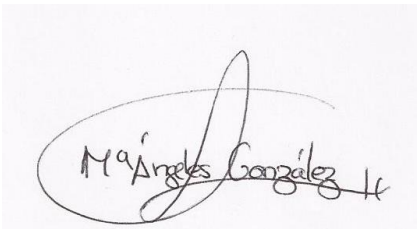
Bajo la dirección de

Alfonso Urquía Moraleda

Autorización

Autorizamos a la Universidad Complutense y a la UNED a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria de este Trabajo Fin de Máster, como el código, la documentación y/o el prototipo desarrollado.

Firmado: Mª Ángeles González Domínguez



Firma del alumno

RESUMEN

En este trabajo nos basaremos en el proceso de análisis y manipulaciones automáticas sobre los modelos físicos que realiza el entorno de modelado OpenModelica, explicado en las diapositivas de Francesco Casella, *Symbolic Manipulation for the Simulation of Object-Oriented Models*. Nos centraremos en la parte referente a la resolución de los DAE y que posibles casuísticas nos podremos encontrar. Más específicamente,

- Partición o emparejamiento de las ecuaciones y variables del modelo. Veremos el algoritmo de eliminación que aparece en el libro *Continuous System Simulation* de F.E. Cellier y E. Kofman página 255; el algoritmo de *score* del artículo de Volpi, *Note of Numeric Calculus*; y el algoritmo de ordenación BLT, explicado también en el libro de F.E. Cellier y E. Kofman, en la Sección 7.2.

- Lazos algebraicos, deshacemos los ciclos dentro del grafo del modelo mediante técnicas de *tearing*. En nuestro caso concreto, se verá un método basado en el algoritmo de ramificación y poda, expuesto en el artículo de T. Gundersen y T. Hertzberg, *Partitionig and tearing of networks - aplied to process flowsheeting, Modeling, Identification and Control.*

- Singularidades estructurales, donde veremos la versión del algoritmo de Pantelides expuesta en el artículo de Pantelides, *The consistent initialization of differencial-algebraic systems.*

Cada uno de los algoritmos ha sido implementado en C++ y el IDE utilizado será *Code::Blocks*, cuyo enlace se proporciona en el mismo trabajo.

PALABRAS CLAVE.

Lazos algebraicos, singularidades estructurales, índice, DAE, Modelica, teoría de grafos, sistemas de ecuaciones algebraicas diferenciales, modelo físico, programación orientada a objetos, C++

ABSTRACT

In this project we will build on the automatic process of analysis and manipulations of physical models performing by OpenModelica modeling environment, explained in Francesco Casella slides, Symbolic Manipulation for the Simulation of Object-Oriented Models. In particular we will focus on the part referring to the resolution of the DAE and possible casuistic we can find in there. More specifically,

- Partition or match of the model equations and variables. We will see the elimination algorithm in the book Continuous System Simulation FE Cellier and E. Kofman page 255; score algorithm in the article Volpi, Note of Numeric Calculus; and the sorting algorithm BLT, also explained in the book of FE Cellier and E. Kofman, Section 7.2.

- Algebraic loops, rid cycles in the graph model by tearing techniques. It will be based on the algorithm branch and bound method, discussed in the article by T. Gundersen and T. Hertzberg, Partitionig and tearing of networks in our case - aplied to process flowsheeting, Modeling, Identification and Control.

- Structural singularities, where we see the version of the Pantelides algorithm set out in the Pantelides' article, The consistent initialization of differencial-algebraic systems.

Each of the algorithms has been implemented in C ++ and the IDE will be used Code :: Blocks, whose link is provided in the same project.

KEYWORDS

Algebraic loops, structural singularities, index, DAE, Modelica, graph theory, algebraic system of differential equations, physical model, object-oriented programming, C++

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE ALGORITMOS

ÍNDICE DE TABLAS

CAPÍTULO 1.
INTRODUCCIÓN, OBJETIVOS Y ESTRUCTURA

1.1. INTRODUCCIÓN.

En este trabajo nos basaremos en el proceso de análisis y manipulaciones automáticas sobre los modelos físicos que realiza el entorno de modelado OpenModelica, explicado en las diapositivas de Francesco Casella, *Symbolic Manipulation for the Simulation of Object-Oriented Models*, y que podemos ver resumido en la Figura 1.1.

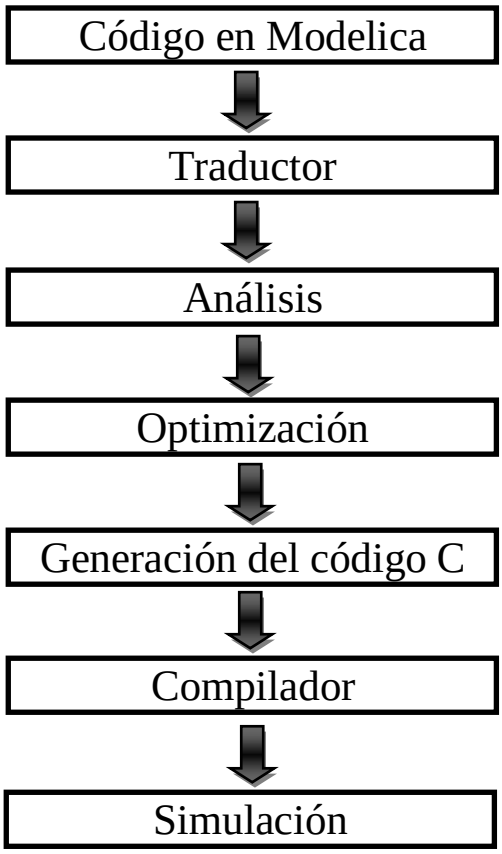


Figura 1.1. Proceso de resolución y simulación de modelos físicos

Inicialmente se transforma el modelo orientado a objetos descrito en lenguaje Modelica, en un modelo plano. En este proceso, denominado *flattening* y que corresponde al paso de traducción de la Figura 1.1, se deshace la composición, jerarquía, herencia y parametrización del modelo descrito en Modelica. El modelo plano está compuesto por declaraciones de variables, parámetros y constantes, ecuaciones, algoritmos y funciones.

En la fase de análisis se realiza la asignación de la causalidad computacional, que consiste en decidir qué ecuación o algoritmo debe usarse para evaluar cada incógnita del modelo. Si la estructura computacional del modelo se describe mediante su matriz de incidencia estructural, este paso consiste en obtener la matriz de incidencia estructural en forma de matriz triangular inferior por bloques (matriz BLT). Además se eliminan del modelo las variables alias y las ecuaciones triviales generadas al traducir las conexiones entre componentes.

En la fase de optimización se realizan análisis y manipulaciones adicionales sobre el modelo si éste es singular, o bien si hay bloques diagonales de dimensión mayor que uno en la matriz de incidencia BLT. En el primer caso, se analiza si el modelo esta sobredeterminado o infradeterminado, y si procede se aplica un algoritmo para la reducción del índice del DAE. En el segundo caso, se aplican algoritmos para la resolución de los lazos algebraicos, entre los que se encuentran los algoritmos de *tearing*.

Finalmente, las fases de creación y compilación del código C, se traducen a código todas las transformaciones que se ha hecho sobre el modelo en las fases anteriores para su posterior ejecución durante la simulación.

1.2. OBJETIVOS.

El principal objetivo del presente trabajo es implementar algunos algoritmos que podrían usarse en las fases de análisis y optimización descritas en la Figura 1.1 de la Sección 1.1. Más concretamente, cuando se quieren obtener la forma BLT de la matriz de incidencia, calcular particiones, resolver modelos con singularidades estructurales, y cortar lazos algebraicos.

Para hallar la partición o emparejamiento de las ecuaciones y variables del modelo, veremos el algoritmo de eliminación que aparece en el libro *Continuous System Simulation* (F.E. Cellier y E. Kofman 2006) y que sigue una estrategia de criba según se vayan asignando los emparejamientos; y dos algoritmos para el cálculo de la forma BLT de la matriz de incidencia.

El algoritmo de *score* del artículo *Note of Numeric Calculus* (Volpi 2004), en el cual propone un método de permutación de filas y columnas basándonos en la búsqueda de los elementos nulos de la matriz; y el algoritmo de ordenación BLT, explicado también en el libro de F.E. Cellier y E. Kofman, en la Sección 7.2, donde se apoya en el algoritmo de Tarjan de componentes fuertes para encotrar la permutación de la matriz.

En cuanto al caso de resolución de singularidades estructurales, veremos la versión del algoritmo de Pantelides expuesta en el artículo *The consistent initialization of differencial-algebraic systems* (Pantelides 1988), en la que se verá cómo introducir nodos derivados para suplir la falta de ecuaciones para poder hallar todas las incógnitas del modelo.

Por último, para lazos algebraicos, veremos un ejemplo de algoritmo de *tearing*, basado en el algoritmo de ramificación y poda de teoría de grafos, expuesto en el artículo *Partitioning and tearing of networks - applied to process flowsheeting* (T. Gundersen y T. Hertzberg 1983)

Todos los algoritmos estarán implementados en C++, de los cuales se hará un estudio de su complejidad y una comparativa de su rendimiento cuando proceda frente a la complejidad teórica obtenida previamente. Además se hará una documentación con las principales variables y funciones utilizadas en cada uno de ellos. La herramienta utilizada para compilar los códigos será *Code::Blocks*. También al final de cada capítulo habrá un ejemplo de ejecución y una discusión de lospros y contras de cada uno de los algoritmos.

1.3. ESTRUCTURA

Aparte del capítulo introductorio, el trabajo está compuesto por otros cinco más en los que se desarrollan los casos con los que nos podemos encontrar a la hora de resolver los sistemas de ecuaciones algebraicas diferenciales, y uno final que recoge las conclusiones y trabajos futuros.

En el Capítulo 2, se hace una revisión general del problema de resolución de los sistemas modelos físicos desde un punto de vista teórico, presentando los principales conceptos necesarios para el desarrollo de los algoritmos en los apartados posteriores.

Los Capítulos 3 al 5, recogen algoritmos para resolución de los problemas de partición. reducción del índice y lazos algebraicos. Dentro de cada una de las sección de los algoritmos habrá una breve descripción de éstos, un comentario de su implementación y su aplicación a varios ejemplos.

En *Conclusiones y Trabajos Futuros*, se hará un resumen general de todo lo visto en apartados anteriores así como una discusión de posibles nuevas líneas de estudio referentes al tema del que trata este trabajo.

Por último indicar que al final del trabajo se encuentran los *Anexos A*, *B* y *C*, donde se incluirán los códigos en C++ de los algoritmos vistos.

.

CAPÍTULO 2

RESOLUCION DE MODELOS DE ECUACIONES ALGEBRAICAS DIFERENCIALES

2.1. INTRODUCCIÓN.

En este capítulo daremos una visión general de los conceptos teóricos necesarios para entender los procesos que se realizan en las fases de análisis y optimización de la Figura 1.1. Comenzaremos con la definición de causalidad computacional y matriz de incidencia, explicando cómo puede ésta transformarse en forma de grafo bipartito.

En la Sección 2.3, definiremos el concepto de partición y de cómo se apoya en la forma BLT de la matriz de incidencia. También se hablará de uno de los métodos utilizado para obtener dicha forma, el algoritmo de Tarjan, en el cual tenemos que construir el grafo de dependencia del modelo, y se apoya en las propiedades de los grafos fuertemente conexos.

En la Sección 2.4, hablaremos de los lazos algebraicos y de cómo pueden ser fácilmente identificados con los bloques de la forma BLT de la matriz de incidencia o con los ciclos del grafo de dependencia del modelo.

Por último, en la Sección 2.5 veremos cuándo un modelo presenta una singularidad estructural, definiremos cómo se calcula su índice, y un ejemplo de algoritmo que se puede utilizar en el caso de problemas de índice superior.

2.2. CAUSALIDAD Y MATRIZ DE INCIDENCIA.

Basándonos en el libro *Modelado orientado a objetos y simulación de sistemas físicos* (A. Urquía y C. Martín 2011), demos una serie de conceptos iniciales fundamentales para la comprensión del resto del trabajo.

Dado un conjunto de relaciones matemáticas o modelo, definimos la asignación de la *causalidad computacional* a la decisión de qué relación debe emplearse para calcular cada incógnita, basándose en una clasificación previa de todas las variables del modelo. Dicha clasificación consiste en dividirlas en dos grandes grupos:

- Conocidas: la variable *tiempo*, las *constantes* y *parámetros*, entradas globales al modelo, y aquellas variables que aparecen derivadas en el modelo, al ser consideradas *variables de estado.*
- Desconocidas: Variables auxiliares que sustituyen las derivadas de las variables de estado, y el resto de incógnitas que aparecen en el modelo.

Atendiendo esta clasificación, se puede representar el conjunto de ecuaciones del modelo en forma de matriz $F(x, y) = 0$ donde el vector x son las variables conocidas y el y las desconocidas.

A partir de esto podremos definir un nuevo tipo de matriz que nos será útil a la hora de ver la causalidad de un modelo (particiones), la *matriz de incidencia*.

Llamamos *matriz de incidencia* o *matriz Jacobiana estructural* a una matriz de elementos booleanos, que indica qué incógnita aparece en cada ecuación. Su construcción se realiza de la siguiente manera: sea A la matriz de incidencia, $x_j \in X$ el conjunto de variables desconocidas y $f_i \in F$ el conjunto de ecuaciones, definimos:

$$A(i,j) = \begin{cases} 0 & si\ x_j \notin f_i \\ 1 & si\ x_j \in f_i \end{cases} \tag{2.1}$$

A partir de la matriz de incidencia también podemos representar los datos mediante un grafo, cuyos nodos están distribuidos en dos columnas. En la primera habrá tantos nodos como ecuaciones en el modelo, y en la segunda, tantos como variables. Las aristas del grafo vendrán dadas por los unos de la matriz de incidencia, es decir, si una variable *v* aparece en la ecuación *f*, entonces existirá una arista que una los nodos *v* y *f*.

Veamos todo esto mediante un ejemplo. Sean las ecuaciones simplificadas que definen el movimiento de un péndulo:

$$\begin{aligned} &\ddot{x} = \lambda * x \\ &\ddot{y} = \lambda * y - g \\ &x^2 + y^2 = L \end{aligned} \tag{2.2}$$

Las escribimos en primer orden mediante la introducción de dos variables auxiliares *u* y *v*:

$$\begin{aligned} \dot{x} &= u \\ \dot{y} &= v \\ \dot{u} &= \lambda * x \\ \dot{v} &= \lambda * y - g \\ x^2 &+ y^2 = L \end{aligned} \tag{2.3}$$

Entonces su matriz de incidencia será:

$$\begin{array}{c} \\ f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \end{array} \begin{array}{c} \begin{array}{ccccccccc} x & y & u & v & \dot{x} & \dot{y} & \dot{u} & \dot{v} & \lambda \end{array} \\ \begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

Figura 2.1. Matriz de incidencia

Nótese que en este caso hemos querido incluir todas las variables para obtener después un grafo los más completo posible.

Y ahora si tenemos en cuenta lo que hemos dicho anteriormente, cada elemento no nulo de la matriz corresponderá con una arista del grafo, y las ecuaciones e incógnitas del modelo con los nodos. Al final obtenemos el grafo:

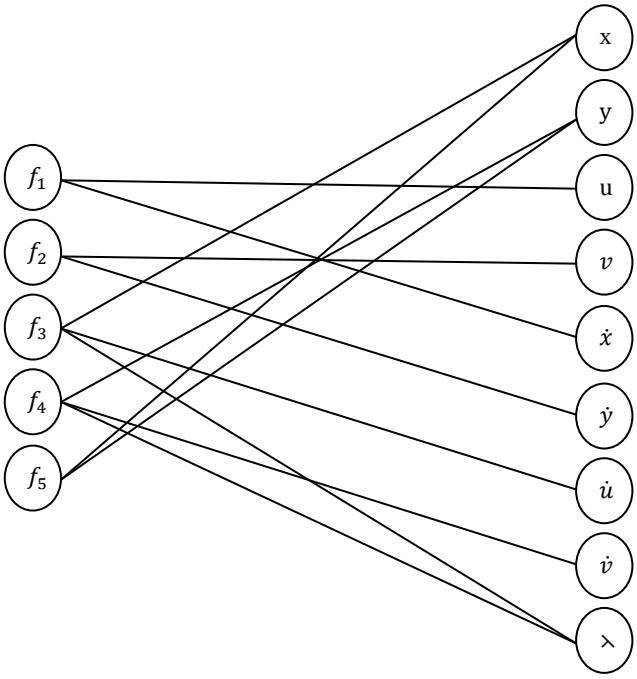


Figura 2.2. Grafo de la matriz de incidencia

2.3. PARTICIONES.

Comencemos primero suponiendo que en el modelo que queremos resolver y simular tiene el mismo número de variables y ecuaciones. En principio, cada incógnita tendría su correspondiente ecuación, pero no solo basta con eso, sino que el modelo no debe contener *singularidades estructurales*. Diremos que un modelo es *no singular* si coinciden el número de ecuaciones y de incógnitas y además:

1. Cada incógnita puede emparejarse con una ecuación en la que aparezca y que no haya sido asignada a otra incógnita previamente.

2. A través de permutaciones de las columnas (variables) y filas (ecuaciones) de la matriz de incidencia, se consigue que todos os elementos de la diagonal sean distintos de cero.

Dadas estas condiciones podremos asignar qué ecuaciones corresponden para resolver qué incógnitas, lo que es igual, hallar la partición del modelo. Además de la definición de no singular, se obtienen las directrices básicas del algoritmo de partición o asignación de la causalidad computacional:

- El objetivo es calcular las incógnitas en base a las variables conocidas: *variables de estado* (variables cuya deriva es una incógnita), parámetros y constantes.
- Las ecuaciones con una única incógnita se asignan directamente a dicha incógnita.
- Las incógnitas que aparecen en una única ecuación, deben ser asignadas a ella.
- Al final cada incógnita debe quedar emparejada de forma única con una ecuación, y viceversa.

Existe una estrategia inicial para abordar este tipo de problemas mediante un sistema de cribas, tachando aquellas filas / columnas ya asignadas. El problema es que sólo sirve para los casos ideales como comentaremos en el Capítulo 3.

Otro método se basa en la segunda condición de los modelos no singulares. Se busca encontrar permutaciones de filas y columnas de la matriz de incidencia hasta convertirla una matriz triangular inferior por bloques o *BLT* (*Block Lower Triangular*) Entonces, definimos *matriz triangular inferior por bloques* como aquella matriz en la que los elementos que están por encima de la diagonal principal son 0 y en la diagonal los elementos no nulos son cuadrados y del menor tamaño posible.

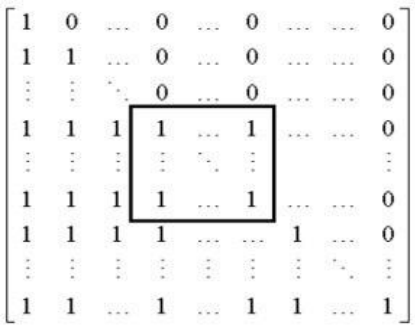


Figura 2.3. Matriz BLT

El caso ideal sería que todos los elementos de la diagonal sean de dimensión 1, pero como se puede observar en la Figura 2.3 , no siempre está garantizado este hecho, sino que aparecen bloques o lazos algebraicos que deben ser resueltos por otros métodos. Una manera fácil de detectar estos bloques es mediante el algoritmo de Tarjan, el cual interpreta la matriz de incidencia como un grafo bipartito y halla los conjuntos de dimensión mínima que habrá en la diagonal. Dichos conjuntos también determinarán las permutaciones necesarias para formar la BLT.

2.3.1. ALGORITMO DE TARJAN Y TEORÍA DE GRAFOS

Un manera diferente de aproximar el problema de hallar la forma BLT de la matriz de incidencia, es transformar ésta en un grafo bipartito que hemos visto en la Sección 2.2, y aplicar sobre él algoritmos propios de la teoría de grafos, aprovechando las propiedades de *conexión* y *fuertemente conexo*, por ejemplo:

$$\begin{array}{c} \\ f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \end{array} \begin{array}{c} \begin{array}{ccccccccc} x & y & u & v & \dot{x} & \dot{y} & \dot{u} & \dot{v} & \lambda \end{array} \\ \begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

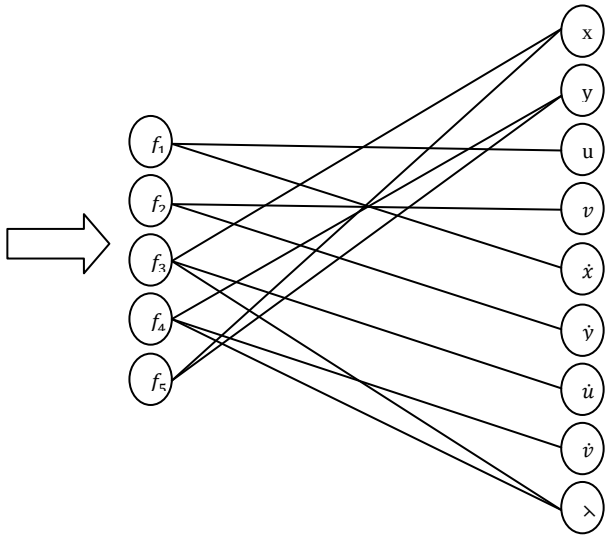


Figura 2.4. Transformación en grafo bipartito

Llamamos *grafo conexo* a aquel en el que para cada par de vértices, *u* y *v*, de G existe al menos una trayectoria, es decir una sucesión de vértices adyacentes en la que no haya repeticiones entre *u* y *v*. En el caso de grafos dirigidos será conexo si su versión sin direcciones lo es.

Un segundo concepto perteneciente a la teoría de grafos, manejado por el algoritmo de Tarjan, es el de *grafo de dependencia* de un modelo. En el algoritmo de Tarjan no se trabaja directamente con el grafo bipartito, sino que se realiza inicialmente un emparejamiento por criba de las ecuaciones y variables (coloración inicial), y después construye un grafo de dependencia con los nodos correspondientes a las ecuaciones. La regla que determina si una función depende de otra es la siguiente:

Sea (f, v) una arista del grafo bipartito, tal que *v* es un nodo correspondiente una variable y *f* a una ecuación, sea f' la ecuación asignada a *v* en el emparejamiento del algoritmo de eliminación entonces: si $f \neq f'$, la arista (f', f) pertenecerá al grafo de dependencia. Este grafo es dirigido y fuertemente conexo.

Un grafo conexo se le llamará *fuertemente conexo*, si para cada par de vértices, *u* y *v*, existe un camino de *u* a *v* y de *v* a *u*, es decir, para cada par de vértices existe un ciclo que le contiene. Además los grafos fuertemente conexos tienen la propiedad de poder dividirse en *componentes fuertes* o ciclos disjuntos que siguen conservando las mismas propiedades del grafo madre.

Ahora bien sabemos que podemos dividir el grafo de nuestro modelo en ciclos disjuntos de dimensión mayor o igual a 1 (un nodo independiente se le considera un ciclo de tamaño 1). Si permutamos las filas (ecuaciones) según los ciclos y luego las columnas gracias al emparejamiento inicial, obtendremos los bloques de la forma BLT. Además, podremos decir que estas componentes fuertes corresponden con cada uno de los bloques que obtenemos en la matriz BLT.

2.4. LAZOS ALGEBRAICOS.

Como hemos visto en la sección anterior, a la hora de calcular la forma BLT de nuestra matriz de incidencia, pueden aparecer bloques de dimensión mayor a 1, a los que llamamos lazos algebraicos. Formalmente definiremos *lazo algebraico* como un subconjunto de ecuaciones del modelo que tienen total dependencia entre sí (ciclo dentro del grafo de dependencia)

Existen varios modos de resolver estos bloques de ecuaciones, principalmente numéricos, algoritmos de *tearing* y algoritmos de relajación. Entre los métodos numéricos tenemos la *eliminación gaussiana*, si las ecuaciones son lineales, y la *iteración de Newton*, en caso contrario.

El método de Gauss consiste en transformar un sistema de ecuaciones en otro equivalente de forma que éste sea escalonado y utilizar el método de reducción de ecuaciones dependientes de manera que en cada ecuación tengamos una incógnita menos que en la ecuación precedente.

En cuanto a la iteración de Newton, consiste en aproximar el valor de las incógnitas del bucle mediante la fórmula iterativa presentada en la Figura 2.3. Sea *x* nuestra incógnita, *f(x) = 0* nuestra ecuación, y sea $x_0 = a$ el valor inicial, entonces definiremos:

$$x_{j+1} = x_j + \frac{f(x_j)}{f'(x_j)} \tag{2.4}$$

Ya volviendo a la perspectiva de la teoría de grafos, nos encontramos con los algoritmos de *tearing*. La idea básica de estos algoritmos es "romper" los ciclos incluyendo nodos nuevos,

siendo éstos versiones relajadas de las incógnitas por las que hemos cortado. En estos algoritmos el objetivo principal es la elección del nodo por donde se va a cortar, también conocido como *variable de tearing*, y no tiene por qué ser única. Un ejemplo de criterio de corte sería aquel nodo del ciclo con mayor número de conexiones.

Por último, se encuentran los algoritmos de relajación. En una idea similar a los métodos numéricos, aproximando el valor de las incógnitas mediante la ecuación Gauss-Seidel:

$$x_n = (1-w)x_n + \frac{b_i - \sum_{j=1}^{i-1}\left(a_{ij}x_j^{(k)}\right) - \sum_{j=i+1}^{i-1}\left(a_{ij}x_j^{(k-1)}\right)}{a_{ii}} \qquad (2.5)$$

2.5. SISTEMAS SINGULARES.

En las secciones anteriores sólo nos hemos centrado en los casos en que el modelo que queremos resolver es no singular. Pero cuando es singular, pueden darse dos situaciones: la primera es que el número de ecuaciones difiere del de incógnitas dando lugar a *sistemas sobredeterminados*, si el número de ecuaciones es mayor al de incógnitas; o *infradeterminados*, si es menor. La segunda es que el número de ecuaciones e incógnitas es el mismo pero a la hora de realizar el algoritmo de partición, una de las variables no tiene ecuación para calcularla, es decir, una ecuación es redundante volviendo a un caso de sistema infradeterminado. Esto se puede deber a un error en el modelo, o bien que el sistema tiene más variables que aparecen derivadas que grados de libertad. En este último caso, se dice que es un problema *estructuralmente singular* o con *índice superior.*

2.5.1. ÍNDICE.

Dado un sistema de ecuaciones DAE, $F(t, x, \dot{x})$, decimos que tiene *índice m*, si *m* es el número mínimo de veces que ha tenido que ser diferenciado el sistema DAE, para poder expresar $\dot{x}$ como una función continua de x y del tiempo.

Generalmente los problemas de índice uno o cero, son bastante fáciles de resolver, en términos de complejidad computacional, a través de algoritmos numéricos. Pero cuando llegamos a índices superiores estos métodos no funcionan adecuadamente debido a que la operación de derivar tiene malas propiedades, y para evitarlas se procura realizar las derivadas

simbólicamente y añadirlas al modelo. Además, el algoritmo de *Pantelides* permite decidir qué ecuaciones deben ser derivadas.

2.6. CONCLUSIONES.

A la hora de querer resolver un sistema de ecuaciones, hemos visto que se siguen una serie de pautas. La primera es obtener la forma BLT de la matriz de incidencia, para calcular la partición del modelo, aunque no siempre es posible. Por otro lado los métodos de ordenación BLT, también nos sirven para detectar problemas con lazos algebraicos.

Otro caso diferente a considerar es cuando el número de ecuaciones es insuficiente para poder causalizar nuestro modelo, teniendo que recurrir a crear "nuevas" ecuaciones y variables mediante derivadas.

Por último, comentar cómo nos hemos apoyado en términos propios de la teoría de grafos, como los grafos de dependencia y sus propiedades, por ejemplo en el algoritmo de Tarjan, aprovechamos que son fuertemente conexos y pueden cortarse en ciclos disjuntos.

CAPÍTULO 3
PARTICIÓN

3.1. INTRODUCCIÓN

Tal como se comentó en los capítulos anteriores a la hora de querer resolver un sistema de ecuaciones algebraicas diferenciales, el primer y más sencillo recurso es encontrar la partición del modelo, buscando despejar las incógnitas del modelo en base a las variables conocidas y constantes. Por supuesto debe cumplirse que el modelo sea no singular, en caso contrario se tendrían que pasar a algoritmos que comentaremos en capítulos posteriores.

En este capítulo veremos varias estrategias para realizar los emparejamientos, o bien seleccionando pares de ecuaciones y variables e ir "tachándolas" de la lista de no asignadas (algoritmo de eliminación); o bien, permutando filas y columnas de las matriz de incidencia hasta obtener la forma BLT, como en el algoritmo de *score* y el de *ordenación BLT*. Además, en este último se verá cómo se pueden aplicar nociones de la teoría de grafos para la localización de los bloques de la diagonal de BLT.

Por último, se hará una comparativa de cada uno de los algoritmos señalando los pros y contras de cada uno.

3.2. ALGORITMOS DE PARTICIÓN

En esta sección vamos a ver dos tipos de estrategias de partición, una mediante criba, con el algoritmo de eliminación; y otra en la permutaremos la matriz de incidencia en forma de triangular inferior, en concreto el algoritmo de *score* y el de ordenación BLT.

3.2.1. ALGORITMO DE ELIMINACIÓN

Antes de centrarnos en algoritmos basados en la triangulación de matrices, veamos el algoritmo propuesto en el libro *Continuous System Simulation* página 255 (F.E. Cellier y E. Kofman 2006), el cual se basa en las reglas 2 y 3 de los algoritmos de partición y se apoya en las siguientes directrices:

- Si una fila contiene una única columna con valor distinto de 0, se asigna dicha ecuación a la incógnita, y se eliminan tanto la fila como la columna.
- Análogamente procedemos por columnas.
- El algoritmo parará cuando no sea posible eliminar más filas o columnas.

Por lo que al final se obtendrán un vector con los pares de ecuación y variable asignada correspondiente.

Entonces el algoritmo de eliminación en pseudocódigo sería en el que se muestra en Código 3.1.

PASO 0. *Inicialización.* Inicializamos *sumasc* y *sumasf* como las sumas por filas/ columnas de la matriz de incidencia.

PASO 1. *Bucle principal.* Mientras haya variables/ecuaciones sin emparejar vamos al PASO 2. Si no PASO 6.

PASO 2. *Elegimos minimo.* Dentro de las ecuaciones y variables no asignadas buscamos aquella cuya suma sea mínima dentro de los vectores *sumasc* y *sumasf*. En caso de empate se dará preferencia a las variables. Si el mínimo está en las variables (columnas) vamos al PASO 3. En caso contrario, al PASO 4.

PASO 3. *Asignamos ecuación a la variable*. Buscamos dentro del vector *sumasf* aquel índice *e* que contenga al mínimo y ese será el número de ecuación asignada. Ir a PASO 5.

PASO 4. *Asignamos variable a la ecuación*. Buscamos dentro del vector *sumasf* aquel índice *v* que contenga al mínimo y ese será el número de ecuación asignada. Ir a PASO 5.

PASO 5. *Marcamos.* Apuntamos la nueva pareja *(e,v)* encontrada y la eliminamos de las variables/ecuaciones sin emparejar. Actualizamos *sumasc* y *sumasf* con la nueva matriz de incidencia.

PASO 6. *Fin.* Sacamos el conjunto de parejas encontrado.

Algoritmo 3.1. Algoritmo de Eliminación

Como se puede observar en el PASO 2, la fila o columna elegida depende de que su suma sea mínima dentro de la matriz de incidencia formada por aquellas filas/columnas que no hayan sido ya emparejadas, (en caso de empate por preferencia será la columna), es decir, se le dan prioridad a aquellas variables/funciones que tienen una única relación y que se verían desparejadas en caso de no seleccionarse en ese momento. Hay que tener en cuenta que las sumas por filas y columnas almacenadas en *sumasc* y *sumasf* se van actualizando según va reduciéndose el tamaño de la matriz de incidencia

IMPLEMENTACIÓN

El algoritmo corresponde con el archivo *.../Códigos/Capitulo 3/Eliminacion/ej.cpp* y se encuentra en Anexo A, Apartado 1.

Los datos de entrada que deberá proporcionar el usuario serán la matriz de incidencia (sin las variables de estado), y el número de filas y columnas que tiene dicha matriz.

Las variables utilizadas por el algoritmo son:

- *var, ecua*: Vector que almacenan qué variables / ecuaciones todavía no han sido asignadas.
- *sumasc, sumasf*: Vector que almacenan las sumas de los valores por filas/columnas de la matriz de incidencia, teniendo en cuenta las filas/columnas eliminadas (no se suman). Son imprescindibles a la hora de elegir la fila/columna que vamos a intentar emparejar.
- *parvar, parecua*: Apuntan en orden que variable/ecuaciones hemos emparejado.
- *datos*: Vector auxiliar que copia la fila/columna que estamos intentando emparejar.

Los vectores *var, ecua, sumasc* y *sumasf* tienen dimensión variable a lo largo del código ya que en cada vuelta su dimensión será inferior debida a que eliminamos las variables y ecuaciones ya asignadas.

De la parte explicada en el PASO 2 se encargará la función *mínimo*, y lo que devolverá es la posición dentro del vector de sumas dónde se encuentre el mínimo, además que dicha posición también coincide con el número de fila / columna de la matriz de incidencia original. La complejidad de la función *minimo* será de $O(n)$ (lineal), n dimensión de la matriz.

A continuación, conocida ya la fila/columna que tiene prioridad, buscamos dentro de ella quién tiene prioridad mediante un sistema similar a la de función *minimo*, con función *minimoreduc* y correspondería a los PASOS 3 y 4. Da igual si se ha escogido una fila o una columna primero, basta con cambiar el orden de los datos de entrada. La complejidad de la función *minimoreduc* será de $O(2n)$ (lineal), *n* dimensión de la matriz.

Por último, actualizamos todos los vectores de control *sumasc*, *sumasf*, *parvar* y *parecua*, y apuntamos el emparejamiento encontrado.

La complejidad del bucle principal será de $O(3n^2)$ (cuadrático), *n* dimensión de la matriz, ya que tiene que lanzar las funciones *minimo* y *minimoreduc n* veces para realizar todos los emparejamientos.

APLICACIÓN

Veamos ahora un pequeño ejemplo aplicado sacado del libro *Continous System Simulation* (F.E. Cellier y E. Kofman 2006). Se la matriz de incidencia 10 x 10:

$$\begin{array}{c} \\ e_1 \\ e_2 \\ e_3 \\ e_4 \\ e_5 \\ e_6 \\ e_7 \\ e_8 \\ e_9 \\ e_{10} \end{array} \begin{array}{c} \begin{array}{cccccccccc} v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & v_7 & v_8 & v_9 & v_{10} \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

PASO 0

Calculamos las sumas por filas y columnas iniciales:

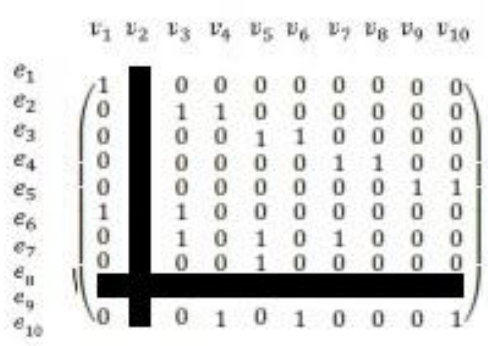
sumasc = (2,1,3,3,3,2,2,1,1,2), *sumasf* = (1,2,2,2,2,2,3,1,2,3)

El mínimo está en la variable v_2 y las que conectan con ella son e_9. El mínimo está en e_9.

Ya tenemos el primer emparejamiento: Parejas $= \{(v_2, e_9)\}$

PASO 1.

Tenemos:



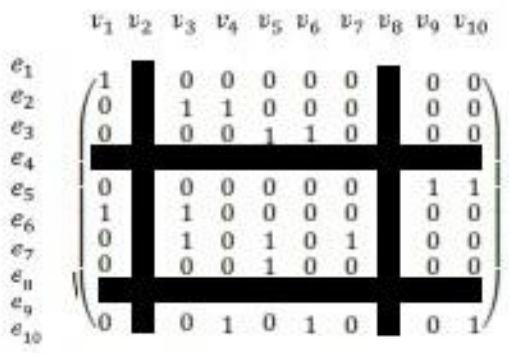
Calculamos de nuevo los vectores de sumas:

sumasc = (2, X,3,3,3,2,2,1,1,2), *sumasf* = (1,2,2,2,2,2,3,1,X,3)

El mínimo está en la variable v_8 y las que conectan con ella son e_4. El mínimo está en e_4.

Entonces nuestros emparejamientos: Parejas = $\{(v_2, e_9), (v_8, e_4)\}$

Tenemos:



Calculamos de nuevo los vectores de sumas:

sumasc = (2,X,3,3,3,2,1,X,1,2), *sumasf* = (1,2,2,X,2,2,3,1,X,3)

PASO N.

Repetimos el proceso hasta llegar a

$$Parejas = \begin{Bmatrix} (v_2, e_9), (v_8, e_4), (v_7, e_7), (v_9, e_5), (v_{10}, e_{10}), \\ (v_4, e_2), (v_3, e_6), (v_1, e_1), (v_5, e_8), (v_6, e_3) \end{Bmatrix}$$

3.2.2. ALGORITMO SCORE

Este algoritmo propuesto en el artículo *Note of Numeric Calculus* (Volpi 2004) busca trasladar a los ceros de la matriz hacia la esquina superior derecha mediante permutaciones de filas y columnas.

Dada una matriz de incidencia *A* con *m* filas y *n* columnas, sea a_{ii} el elemento de la diagonal perteneciente a la fila *i* columna *i*. Vamos recorriendo los elementos de la primera fila de derecha a izquierda hasta el elemento a_{1i+1}, siguiendo las pautas:

- Si elemento $a_{1j} = 0$, cumple con nuestro objetivo, por lo que pasamos al siguiente a_{1j-1}
- Si el elemento $a_{1j} \neq 0$, Buscamos por columnas de izquierda a derecha el primer elemento b_{ik} que cumpla: $b_{ik} = 0, i \geq 1, k < j$.
- La búsqueda será lineal por filas recorriendo las columnas de izquierda a derecha hasta llegar al elemento correspondiente diagonal, y después saltando a la siguiente columna.
- Hacemos la permutación por filas y columnas.
- El algoritmo parará cuando no sea posible hacer más permutaciones con los elementos de la primera fila.

PASO 0. *Inicialización.* Dado *ce* $\leftarrow j$, j = *dim(M)* - 1. *M* matriz de incidencias.

PASO 1. *Bucle principal.* Mientras *j* no sea la primera columna y haya opciones para explorar vamos al PASO 2. Si no PASO 5.

PASO 2. *Casuística.* Si $M_{1j} = 0$, no es necesario hacer permutación. Volvemos al PASO 1. En caso contrario vamos a PASO 3.

PASO 3. *Búsqueda.* Buscamos por columnas de izquierda a derecha el primer elemento b_{ik} que cumpla: $b_{ik} = 0, i \geq 1, k < j$ Ir a PASO 4.

PASO 4. *Permutación*. Intercambiamos las columnas *j* y *k*, y las filas *j* y *k* .Ir a PASO 5.

PASO 5. *Fin.* Obtendremos una matriz BLT.

Algoritmo 3.2. Algoritmo de Score

Según el autor en casos en los que la cantidad de ceros en esta zona sea alta, basta con sustituir la búsqueda del PASO 3 con una función ponderada que elegirá la columna más adecuada. Dicha función se basa en restar la cantidad de ceros que se encuentran en la parte triangula superior de la matriz antes y después de la permutación y si es mayor, la permutación será válida y podrá realizarse.

IMPLEMENTACIÓN

El algoritmo corresponde con el archivo *.../Códigos/Capitulo 3/Score/Ej.cpp* y se encuentra en Anexo A, Apartado 2.1.

Los datos de entrada que deberá proporcionar el usuario serán la matriz de incidencia (sin las variables de estado), y el número de filas y columnas que tiene dicha matriz.

El programa no tiene una inicialización explícita, sino que pasa directamente a la implementación de los pasos comentados anteriormente.

La complejidad de la función será de $O(n)$ (lineal), *n* dimensión de la matriz.

En cuanto a la modificación con la función ponderada la complejidad pasa a ser $O(n^2)$ (cuadrático).

El código perteneciente a esta última parte está en *.../Códigos/Capitulo 2/Score/Ej con peso nf.cpp* y se encuentra en Anexo A, Apartado 2.2.

APLICACIÓN

Veamos un ejemplo. Sea la matriz 6 x 6:

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 1 & 2 & 0 \\ 1 & -1 & 1 & 2 & -3 & 2 \\ -6 & 1 & 1 & 3 & 5 & 2 \\ 1 & 0 & 0 & 3 & -1 & 0 \\ 2 & 0 & 0 & 5 & 1 & 0 \\ -9 & 2 & 1 & 1 & 7 & 1 \end{pmatrix} \end{array}$$

PASO 1.

El primer elemento por la derecha distinto de 0 es a_{15} .

Buscamos desde la izquierda y el primer elemento distinto de 0 es a_{12} .

Entonces intercambiamos las columnas 2 y 5, y las filas 2 y 5 obteniendo:

$$\begin{array}{c} \\ 1 \\ 5 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1 & 5 & 3 & 4 & 2 & 6 \end{array} \\ \begin{pmatrix} 1 & 2 & 0 & 1 & 0 & 0 \\ 2 & 1 & 0 & 5 & 0 & 0 \\ -6 & 5 & 1 & 3 & 1 & 2 \\ 1 & -1 & 0 & 3 & 0 & 0 \\ -9 & -3 & 1 & 2 & -1 & -2 \\ & 7 & 1 & 1 & 2 & 1 \end{pmatrix} \end{array}$$

PASO 2.

Repetimos el proceso y obtenemos que hay que intercambiar 3 y 4 consiguiendo nuestra matriz BLT final:

$$\begin{pmatrix} 1 & 2 & 1 & 0 & 0 & 0 \\ 3 & 1 & 5 & 0 & 0 & 0 \\ 1 & -1 & 3 & 0 & 0 & 0 \\ -6 & 5 & 3 & 1 & 1 & 2 \\ 1 & -3 & 2 & 1 & -1 & -2 \\ -9 & 7 & 1 & 1 & 2 & 1 \end{pmatrix}$$

3.2.3. ALGORITMO DE ORDENACIÓN BLT

Ahora veamos el algoritmo de ordenación BLT propuesto en la Sección 7.2 del libro *Continuous System Simulation* (F.E. Cellier y E. Kofman 2006), el cual no ignora los bloques de la diagonal, al contrario de lo que hace el de eliminación, donde sólo se preocupa de encontrar emparejamiento de todas las incógnitas y las funciones.

Las directrices del algoritmo son las siguientes:

1. Hallamos un emparejamiento mediante el algoritmo de eliminación del apartado 3.2.1, y mediante este emparejamiento, construimos el grafo de dependencia de las ecuaciones del modelo.

2. Lanzamos el algoritmo de Tarjan para encontrar las componentes fuertes del grafo de dependencia. Esta parte la desarrollaremos en el apartado 3.2.4.

3. Hacemos la permutación de la matriz de la siguiente manera: colocamos las filas según el orden en que se han obtenido las componentes fuertes, y las columnas, en base a ese orden, según las asignaciones se hayan hecho en el paso 1, es decir, si una función f según las componentes fuertes se obtiene que ir a la posición 3 (pasa a ser la tercera fila de la matriz BLT), entonces la columna de la variable v asignada a f pasará también a la posición 3.

Al final se obtendrá la BLT ordenada que buscamos.

PASO 0. *Emparejamiento*. Lanzamos el algoritmo de eliminación para encontrar una emparejamiento inicial. Ir a PASO 1.

PASO 1. *Grafo de dependencia*. Construimos el grafo de dependencia de modo sea $(f, v) \in$ G grafo sea f' la ecuación asignada a *v* entonces: si $f \neq f'$, la arista $(f', f) \in$ Gdep. Ir a PASO 2.

PASO 2. *Tarjan. Lanzamos* el algoritmo de Tarjan para localizar los bloques o componentes fuertes de la futura matriz BLT. Ir a PASO 3.

PASO 3. *Permutación*. Permutamos las filas según el orden de las componentes fuertes del PASO 2, y luego las columnas siguiendo el mismo orden pero basándonos le emparejamiento del PASO 0 .Ir a PASO 4.

PASO 4. *Fin.* Obtendremos una matriz BLT.

Algoritmo 3.3. Algoritmo de ordenación BLT

El ejemplo de aplicación de este algoritmo lo veremos al final del siguiente apartado dado que no hemos explicado todavía el algoritmo de *Tarjan*.

IMPLEMENTACIÓN

El código aquí mostrado corresponde con al proyecto *.../Códigos/Capitulo 2/BLT y Tarjan/BLT.cpp* y se encuentra en Anexo A, Apartado 3.

En este sólo comentaremos la parte del programa correspondiente a los puntos 2 y 4 antes comentados (para el primer punto ir al apartado 3.2.1). En cuanto al tercer punto lo veremos en 3.2.4.

Los datos de entrada que deberá proporcionar el usuario serán la matriz de incidencia (sin las variables de estado), y el número de filas y columnas que tiene dicha matriz.

Como hemos visto antes, tras conseguir un emparejamiento inicial con la función *emparejar*, debemos construir el grafo de dependencia cuyas las aristas las almacenaremos en el vector *conexiones* de modo que dada la ecuación *i*, meteremos en *conexiones[i]* todas aquellos índices de funciones con las que está conectado.

Después de obtener el grafo de dependencia, lanzaremos el algoritmo de Tarjan, para así obtener el orden en que debemos permutar la matriz para así ponerla en forma BLT (Para más detalle ir a la Sección 3.2.4).

Por último realizamos, la permutación de la matriz respetando el orden en que han salido los índices de las ecuaciones en el vector de componentes fuertes *cfuertes*.

Primero permutamos las filas según *cfuertes* y luego las columnas (variables), con la ayuda de un vector auxiliar que permuta el vector *assignv* según *cfuertes*.

Su complejidad es de de $O(n^2 + n * e)$ (cuadrático), n dimensión de la matriz y e el número de aristas del grafo . La complejidad del $O(n * e)$ corresponde al algoritmo de Tarjan como veremos en el apartado siguiente.

3.2.4. ALGORITMO DE TARJAN

Como se ha dicho en el apartado anterior dado un grafo dirigido, el *algoritmo de Tarjan* busca la componentes fuertes del grafo de dependencia, el cual es fuertemente conexo. La versión que vamos a implementar en este trabajo será la propuesta en el libro *Técnicas de cálculo para sistemas de ecuaciones, programación lineal y programación entera* (O'Connor 1997)

PASO 0. *Inicialización.* Pila $\leftarrow \emptyset$, $i \leftarrow 1$. Ir a PASO 1.

PASO 1. *Tarjan.* Para todo $v \in \mathrm{G(V)}$ y $num(v) \neq -1$, ir a PASO 2.

PASO 2. *Empilamos v* $.num(v) = i,\ lowlink(v) = i$ y $Pila = Pila \cup \{v\}$. Ir a PASO 3.

PASO 3. *Exploración.* Miramos la casuística de aquellos nodos conectados a *v*. $\forall w \in G(V), (v,w) \in G(A)$, ir a PASO 4. Si no existe ningún *w* que cumpla condición, vamos al PASO 6.

PASO 4. *Estudio de w.* Si $num(w) = -1$, no se encuentra en la pila, vamos al PASO 2 con *v = w.* En caso contrario ir a PASO 5.

PASO 5. *Actualizamos lowlink.* Hacemos $lowlink(v) =$ min $(lowlink(v), lowlink(w))$ para marcar posibles ciclos. Ir a PASO 6.

PASO 6. *Componentes fuertes.* Si $num(v) = lowlink(v)$, hemos localizado un ciclo o componente fuerte. Sacamos vértices *u* de la *Pila* hasta que *u = v*. Guardamos el conjunto de vértices extraídos como una nueva componente fuerte.

Algoritmo 3.4. Algoritmo de Tarjan

Los pasos principales que siguen el algoritmo son los siguientes:

1. Vamos explorando uno a uno los nodos siguiendo las direcciones (grafo dirigido), apuntando cuáles forman ciclos y con quién.

2. Definimos dos vectores de apoyo *lowlink*, que indica con qué nodo del grafo forman un ciclo, y *num*, indica la posición de un nodo una vez reenumerado.

3. Según vamos explorando los nodos los almacenamos en una pila, sólo los sacaremos cuando hayamos encontrado la componente fuerte o ciclo al que pertenezcan.

   A la hora de explorar un nodo, *v*, miramos el resto de nodos, *w*, con los que está conectado. Para cada uno de ellos nos podemos encontrar con varios casos:

   - *w* todavía no está apilado es decir, su valor en *num* es nulo (en nuestra implementación por convención será -1) entonces lo apilamos, actualizamos su *lowlink* y continuamos explorando a continuación de él.

- *w* está en la pila, entonces comprobamos si puede ser el final de un ciclo (está más abajo en la pila), es decir, $num(w) < num(v)$. En ese caso actualizamos *lowlink* para dejar marcada la componente fuerte.

Una vez hecha la clasificación de todos los nodos en el paso 4, buscamos los nodos en la pila aquellos que cumplan que *lowlink* = *num*. Estos indican las diferentes componentes fuertes.

IMPLEMENTACIÓN

El código aquí mostrado corresponde con el proyecto *.../Códigos/Capitulo 2/BLT y Tarjan/BLT.cpp*, concretamente en archivo *main.cpp*, función *f_connect*.

Las principales variables utilizadas en este algoritmo son:

- *num*, y *lowlink* con la definición comentada arriba.
- *enpila*: marca si el nodo está en la pila de explorados del algoritmo.
- *conexiones:* vector en el que se apuntan los índices de los nodos que están conectados a cada uno de los nodos función.
- *pila*: la pila en la que se van metiendo los nodos ya explorados.
- *ind*: controla el número de nodo por el que nos encontramos en la exploración.
- *cfuertes*: almacena las componentes fuertes que se hayan ido encontrando.

La llamada al algoritmo se encuentra en nuestra función *main()* y se lanza para cada uno de los nodos del grafo de dependencia que no hayan sido metidos en la pila previamente (ya explorados).

El algoritmo en sí se desarrolla en la función *fconnect* cuyos parámetros de entrada o e/s (en caso de que tengan un & delante) son los comentados arriba.

Primero inicializamos el nodo que estemos explorando en este momento, lo incluimos en la pila y asignamos sus valores de num y lowlink.

En segundo lugar, exploramos cada uno de los nodos (w) conectados al actual y vemos la casuística. Si w no pertenece a la pila, eso quiere decir está en mitad de un ciclo y debemos

seguir explorando por ese camino, por lo que lanzamos recursivamente la función poniendo como nodo a explorar a w y cuando regresamos apuntamos el ciclo encontrado a través de lowlink. En caso de que w ya perteneciera a la pila, miramos si con el nodo actual forma parte de un ciclo.

Por último, nos queda ver si el nodo actual es el final del ciclo (sus *lowlink* y *num*) y sacar la componentes fuerte de la pila. Se seguirán sacando nodos de la pila hasta que el nodo que se encuentre en la cima sea el actual, y entonces guardamos la componente fuerte encontrada en *cfuertes*.

Su complejidad es de de $O(n * e)$, n dimensión de la matriz y e el número de aristas del grafo.

APLICACIÓN

Sea la matriz de incidencia inicial:

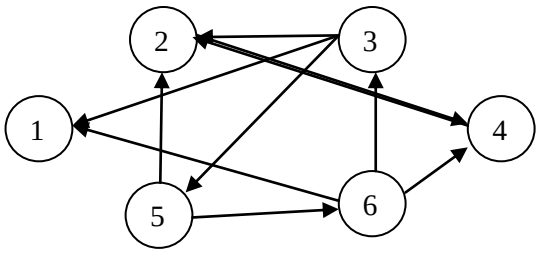
$$\begin{pmatrix} 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}$$

PASO 1.

Hallamos el emparejamiento inicial : $(f_2, v_1), (f_3, v_2), (f_6, v_3), (f_1, v_4), (f_4, v_5), (f_5, v_6)$

PASO 2

Construimos según su definición el grafo de dependencia:



PASO 3 . Algoritmo de Tarjan

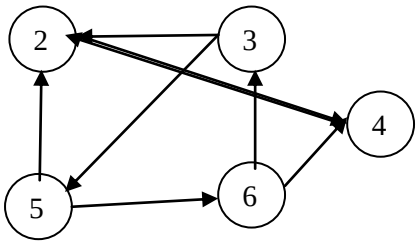
ITERACIÓN 1

Pila = {1}, ind = (1, -1, -1,-1,-1,-1), lowlink = (1,-1,-1,-1,-1,-1)

No salen conexiones del nodo 1 → Es una componente fuerte. La sacamos de la pila y actualizamos nuestro conjunto de componentes fuertes:

Pila = ∅, Comp. fuertes = {{1}}

Y nuestro grafo de dependencia quedará



ITERACIÓN 2

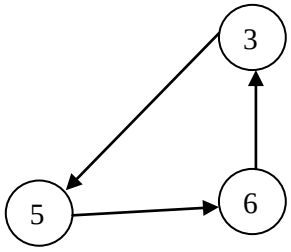
Exploramos por el nodo 2 y obtenemos el camino 2 ---> 4, con

Pila = {2,4}, ind= (1,2,3,-1,-1,-1), lowlink = (1,2,3,-1,-1,-1)

Como no hay mas camino lowlink (4) = lowlink (2) y obtenemos:

Pila = ∅, Comp.fuertes = {{2,4},{1}}

El grafo de dependencia quedará:



Claramente un ciclo que será la última componente fuerte, por lo tanto:

Comp. fuertes = {{3,5,6,}, {2,4}, {1}}

PASO 4 . Permutación

Realizamos la permutación de las filas según el orden 3,5,6,2,4,1

Y las columnas mapeando los valores con el emparejamiento inicial : 2,6,3,1,5,4

Obtenemos

$$\begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}$$

3.3. RESULTADOS

Antes de ver tiempos de ejecución veamos en profundidad algunas cuestiones referentes a los algoritmos:

El algoritmo de eliminación (Sección 3.2.1), en caso de no presentar singularidades, garantiza que se encuentran los emparejamientos, pero si la matriz de incidencia presenta bloques en la diagonal (lazos algebraicos), los ignora completamente dando soluciones no correctas, es decir, el algoritmo termina aunque sigue habiendo dependencia entre variables no conocidas.

Un ejemplo de esto lo vemos si aplicamos el algoritmo de eliminación al ejemplo de la Sección 3.2.4. Al lanzar el algoritmo sólo obtendremos la solución que tomamos como inicial en el de ordenación BLT: $(f_2, v_1), (f_3, v_2), (f_6, v_3), (f_1, v_4), (f_4, v_5), (f_5, v_6)$, pero vemos claramente que en ese ejemplo lo que resulta es una matriz con bloques de dimensión mayor a uno en la diagonal.

$$\begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}$$

Ahora bien, en referente a nuestro segundo algoritmo, el de *score* (Sección 3.2.2), no ignora los bloques y debido a su complejidad se podría decir que es un excelente algoritmo, pero no siempre funciona. Los ejemplos propuestos por el autor siempre tienen una característica común la cantidad de ceros en la parte triangular superior.

¿Dónde está la trampa? Muy sencillo, en ejemplos en los que la triangular superior prácticamente nula, como la del ejemplo del apartado anterior, no deja realizar ninguna permutación, aun sabiendo que se puede obtener la forma BLT.

Debido todo la anterior comentado, la mejor opción sería el algoritmo de *ordenación BLT*.

Ahora comparemos el tamaño frente a tiempo.

		n = 6	n = 10	n = 15	n = 20	Complejidad
Eliminación	Tiempo (seg)	0.035	0.071	0.222	0.312	$O(3n^2)$
	Iteraciones	6	10	15	20	
Score con función ponderada	Tiempo (seg)	0.7	0.227	0.606	0.713	$O(n^2)$
	Iteraciones	6	16	66	23	
Tarjan	Tiempo (seg)	0.041	0.053	0.058	0.065	$O(n^2 + n * e)$
	Iteraciones	6	10	15	20	

Tabla 3.1. Algoritmos de partición

Claramente podemos observar que según el tamaño de nuestra matriz crece, el algoritmo más efectivo es el de ordenación BLT con Tarjan.

Aunque en apariencia todos tiene una complejidad similar, el n^2 que aparece en el algoritmo de Tarjan se debe a las salidas por pantalla de la matriz y la inicialización de los vectores, quitando eso es prácticamente lineal. En cambio, en los otros casos hay se recorre una y otra vez la matriz dando lugar a los cuadrados.

3.4. CONCLUSIONES

Hemos visto tres ejemplos de algoritmos para resolver particiones, uno por sistema de criba y los otros dos por permutaciones. El peor de los tres ha resultado es de *score* debido a que sólo sirve para determinados casos muy preparados, en el resto ni siquiera cumple con el objetivo de crear la BLT.

El algoritmo de eliminación está bien para casos en los que se sepa de antemano que no va a haber lazos algebraicos, o para hallar emparejamientos iniciales, muy útiles para métodos que veremos en capítulos posteriores.

Por último, comentar la potencia que tienen los algoritmos de grafos en cuestión de tiempo. La exploración inteligente que se realiza en el algoritmo de Tarjan reduce la complejidad, ya que se evita volver a pasar por zonas ya exploradas, como sí hace el de eliminación.

CAPÍTULO 4
CÁLCULO Y REDUCCIÓN DEL ÍNDICE

4.1. INTRODUCCIÓN.

En el capítulo anterior sólo nos hemos centrado en los casos en que el modelo que queremos resolver es no singular. Pero hay casos en los que pese a que el número de ecuaciones e incógnitas es el mismo, a la hora de realizar el algoritmo de partición, una de las variables no tiene ecuación para calcularla, es decir, una ecuación es redundante. Esto se puede deber a un error en el modelo, o bien a que el sistema tiene más variables que aparecen derivadas que grados de libertad, encontrándonos ante un problema *estructuralmente singular* o con *índice superior*. En este capítulo precisamente veremos un ejemplo de cómo podemos resolver esta situación.

Generalmente los problemas de índice uno o cero, son bastante fáciles de resolver en términos de complejidad computacional a través de algoritmos numéricos. Pero cuando llegamos a índices superiores estos métodos no funcionan adecuadamente, y necesitamos un mecanismo para decidir que ecuaciones son necesarias derivar. Dicho mecanismos son los *algoritmos de reducción de índice*, más concretamente vamos a ver aquí una versión del algoritmo de *Pantelides* para problemas de índice superior expuesta en el artículo *The consistent initialization of differencial-algebraic systems* (Pantelides 1988).

4.2. ALGORITMO DE PANTELIDES

Como vimos en el Capítulo 2, a través de la matriz de incidencia, podemos formar un grafo a partir del modelo, en el que los nodos de las ecuaciones se encuentran en una columna y los de las variables en la otra, conectados entre sí allí donde haya un 1 en la matriz de incidencia. Precisamente en esta estructura es en la que nuestro algoritmo se va a apoyar.

Basándonos en la idea expuesta en el artículo, con nuestro modelo distribuido en forma de grafo buscamos hallar un emparejamiento para todas las variables, por lo que se sigue una estrategia similar a la del algoritmo de eliminación, pero en el caso de modelos con singularidades

estructurales, tenemos ecuaciones singulares, es decir, todas variables a las que están conectadas han sido ya marcada, obligándonos a crear nuevos nodos a través de derivadas e incluirlos en el grafo del modelo para volver a intentar realizar el emparejamiento.

PASO 0. *Inicialización.* $C = C \cup \{f\}$. *match*← *false.* Ir a PASO 1.

PASO 1. *Emparejamiento.* Si $\exists v \in \mathrm{G(V)}\ tal\ que\ (f, v) \in G(A)$, *assign[v]= NIL* y *vmap[v]= NIL*, ir a PASO 2. En caso contrario ir a PASO 3.

PASO 2. Asignamos emparejamiento: $\mathrm{assign}(v) = f\ y\ match = true$. FIN

PASO 3. *Exploración.* Miramos la casuística de aquellos nodos conectados a *f* que cumplan $v \in G(V), (f, v) \in G(A)\ ,\ v \notin C\ y\ vmap[v] = NIL$, ir a PASO 4.

PASO 4. *Rellamada.* . $C = C \cup \{v\}$. Volvemos al PASO 0 con $f = assian(v)$. Si

Algoritmo 4.1. Función de emparejamiento

Los pasos que sigue el algoritmo son los siguientes:

1. Como se ha dicho antes buscamos hacer un emparejamiento de cada ecuación como se hace en el algoritmo de eliminación, por lo que vamos para cada nodo ecuación, lanzamos la función de emparejamiento que determinará si podemos, modificando o no otras parejas, o bien no hay ningún nodo disponible para asignarle, explorando las diferentes aristas del grafo. El algoritmo de emparejamiento modificado se encuentra en el Algoritmo 4.1.

   Además podemos observar que vamos apuntando en la pila *C* aquellos nodos por los que se ha hecho la exploración en busca del emparejamiento. Dicha pila *C* será útil para el punto 3.

2. Si se ha encontrado una pareja para la ecuación entre sus adyacentes, pasaríamos a la siguiente ecuación.

3. Si no se ha encontrado pareja, incluso intentando distribuir el resto, dichas cadenas de nodos que deberían modificarse, se guardan en una pila *C* y para cada uno de ellos deberemos calcular la derivada, es decir, ampliamos las opciones de emparejamiento en aquellas cadenas donde encontramos problemas incluyendo nuevos nodos.

4. En el caso de ser nodos ecuación, además se deberán asignar las conexiones a ese nuevo nodo que se introduce en el grafo, y vendrán determinadas por las que tenía el nodo ecuación del que derivan. Sea f el nodo sin pareja y f' el nuevo nodo derivado, entonces cada nodo variable *v* que estaba conectado con f lo estará con f', y también v'.

5. Por último repetiremos el proceso evaluando el emparejamiento desde el nodo que representa la ecuación derivada de la ecuación que se ha evaluado en el punto uno.

6. El algoritmo terminará cuando se haya encontrado emparejamiento para todas las variables que no sean de estado.

PASO 0. *Inicialización.* $C \leftarrow \emptyset$, *assignv*$\leftarrow \emptyset$,.*vmap* y *eqmap* Ir a PASO 1.

PASO 1. *Bucle principal.* Si $\forall f \in \mathrm{G(F)}$, ir a PASO 2

PASO 2. *Emparejamiento.* Calcular *match* de *f* con la función de emparejamiento. Si existe ir al PASO 1, en caso contrario ir a PASO 3

PASO 3. *Nuevas derivadas:* Para todo elemento de la pila C, seguimos la casuística:

a) Si es un nodo variable vamos al PASO 4.
b) Si es un nodo función vamos al PASO 5.

PASO 3. *Variable derivada*. Dado *v* nuestro nodo calculamos *v'*, *vmap[v]* = *v'* (apuntamos que v' es derivada de v), incluimos v' en el grafo. Ir a PASO 6.

PASO 4. *Función derivada*. Dado *g* nuestro nodo calculamos *g'*, *eqmap[g]* = g' (apuntamos que f' es derivada de g), incluimos g' en el grafo. Ir a PASO 5.

PASO 5. *Generación de las nuevas aristas.* Conectamos los nuevos nodos función derivada con las variables que contengan y sus respectivas derivadas: $\forall\, v \in \mathrm{G(V)}, (g, v) \in \mathrm{G(E)}, G(E) \leftarrow G(E) \cup \{(g', v), (g', v')\}$

PASO 6. *Asignación.* Asignamos a las nuevas variables derivadas su función derivada correspondiente: $\forall\, v \in C, assign[v'] = eqmap[assign[v]]$

PASO 7. *Reinicialización*. Volver a PASO 2, con *f* = *f'*.

Algoritmo 4.2. Algoritmo de Pantelides

Existen casos en los que el algoritmo de Pantelides aquí expuesto puede darse una situación de bucle infinito debido a que el sistema sea inconsistente, tal como se comenta en el artículo *The*

consistent initialization of differencial-algebraic systems (Pantelides 1988). Más concretamente, enuncia el siguiente Teorema 4.2:

"El algoritmo termina si y sólo si el sistema extendido es no singular. En caso contrario el DAE en estructuralmente inconsistente"

Por ejemplo, si al realizar la creación de los nodos derivados y posterior búsqueda de una nueva coloración, nos encontramos en la pila *C* que la estructura se "repite" respecto a la iteración anterior, puede ser que nos encontremos ante una situación en la que el nuevo grafo tenga la misma singularidad estructural sólo que formada con los nuevos nodos.

IMPLEMENTACIÓN

El código aquí mostrado corresponde con al proyecto *.../Códigos/Capitulo 3/Pantelides/ Pantelides.cbp* y el Anexo B.

Los datos de entrada que deberá proporcionar el usuario serán la matriz de incidencia (incluyendo las variables de estado), el número de filas y columnas que tiene dicha matriz, y por último el vector de enteros *mapini* en el que se indica, habiendo numerado las variables del modelo (incluido de estado), el número de variable que corresponde su derivada, y en caso de no tenerla se pondría -1. Por ejemplo si nuestro modelo tiene *x* con posición 0 y *x'* con posición 4, entonces $mapini[0] = 4$. Sin perder generalidad se tomará -1 como NULL, ya que no se consideran los vectores circulares.

Como apoyo para poder identificar el tipo de nodo (variable o función) que se va metiendo en la pila se utiliza la clase *Nodo* que tiene dos características:

- *tipo*: dice el tipo de nodo que es. Sus dos valores posibles son 'v' de variable, o 'f' de función o ecuación.
- *numtipo:* indica el número de columna o fila dentro de la matriz al que corresponde el nodo.

El uso de este objeto nos permite ahorrar el tener que crear al menos dos vectores más necesarios para poder de qué tipo son los elemento añadidos a la pila *C*. Por otro lado las

operaciones que tendrá la clase *Nodo* son aparte de los constructores, las dos funciones *get* para tipo y numtipo.

Las variables utilizadas en el algoritmo son:

- *variables* y *funciones*: Almacenan los nodos variable y ecuación, incluidos los nuevos que se van creando a lo largo del algoritmo.
- *fconexiones*: por cada ecuación, almacena los *numtipo* de las variables que se encuentra en ella.
- *vpila*: Marca si un nodo variable está en la pila.
- *vmap* y *eqmap*: Siguiendo una idea de *mapini*, almacenan la correspondencia de índices entre las variables/ecuaciones y sus derivadas. Además el *vmap* inicial es igual a *mapini*, y *eqmap* es -1 en todas sus componentes.
- *assignv*: Indica el *numtipo* de ecuación con la que se ha emparejado la variable.
- *match*: indica si hemos encontrado emparejamiento para la ecuación que estemos considerando en ese momento.
- *C* pila donde se almacenan los nodos de los que se deben calcular su derivada.

Lo primero que se realiza es inicialización de todos los vectores de apoyo del algoritmo y creamos también *fconexiones* a partir de la matriz de incidencia.

A continuación, para cada ecuación se evalúa si puede emparejar con una variable. De esto se encarga la función *match_eq*. Dada *f* la ecuación que queremos evaluar en este momento, los pasos que se siguen para determinar si dicho emparejamiento existe son:

PASO 1. Añadimos *f* a la pila para marcarlo. Ahora, dados los *numtipo* de todos los nodos variable que se encuentran conectados a *f*, miramos si alguno cumple que no es variable de estado (no tiene derivada dentro del modelo) y no se le ha emparejado a otra ecuación.

PASO 2. Si hemos encontrado una variable que cumpla las condiciones anteriores, apuntaremos el emparejamiento y devolveremos *match = true*.

PASO 3. En caso contrario para aquellos nodos variable *v* que no hayan sido metido en la pila previamente y que no tengan derivada, los apuntamos en *C* para ser

para candidatos a ser derivados más adelante en caso de no encontrar emparejamiento

Por otro lado seguiremos explorando el grafo a través del nodo función que está asignado a *v*, por si es posible encontrar un emparejamiento para nuestro *f* actual, cambiando las asignaciones. Esta última parte se realiza mediante una llamada recursiva a la propia *match_eq*.

PASO 4. Si se consigue encontrar emparejamiento, se devolverá *match=true*, sino se devolverá la pila *C* con todos los nodos (ecuación o variable) que serán candidatos a ser derivados, y *match=false*.

Si hemos conseguido encontrado un emparejamiento, pasaremos a la siguiente ecuación, en caso contrario deberemos crear nuevos nodo a partir de las derivadas de los nodos apuntados en la pila *C*.

Exploramos uno a uno los nodo de *C* y según su tipo realizamos una serie de operaciones:

- Si el del tipo 'v' (variable), creamos un nuevo nodo *noprima*, el cual lo añadimos nuestros vectores *variables*, *vmap*, *vpila* y *assignv*. Además apuntamos *noprima* como la derivada dl nodo que consideramos en este momento.

- Si es del tipo 'f', hacemos algo similar al caso de las variables con los vectores *eqmap*, y *funciones* al crear el nodo derivado.

  Además al ser un nodo función también hay que considerar cuáles van a ser sus conexiones dentro del grafo. Por cada nodo variable *v* conectado con el nodo *f* de la pila *C*, anotamos como conexión para *f'*, *v* y *v'* (regla de derivación)

A continuación sólo nos quedará asignar a las nuevas variables derivadas *v'* la derivada de la función que tenían las variables *v* de la que proceden, para completar el emparejamiento inicial necesario en *eq_match*.

Por último como no se ha encontrado un emparejamiento para la función inicial *f* deberemos seguir a partir de su derivada, ya que ésta ha pasado a ser una ecuación exclusivamente compuesta por variables de estado.

Como se ha dicho el algoritmo terminará cuando se hayan podido emparejar todas las variables que no sean de estado. Pero también se puede sacar otra información de resultado gracias al

vector de apoyo *eqmap*: las ecuaciones que es necesario derivar para reducir el índice del modelo.

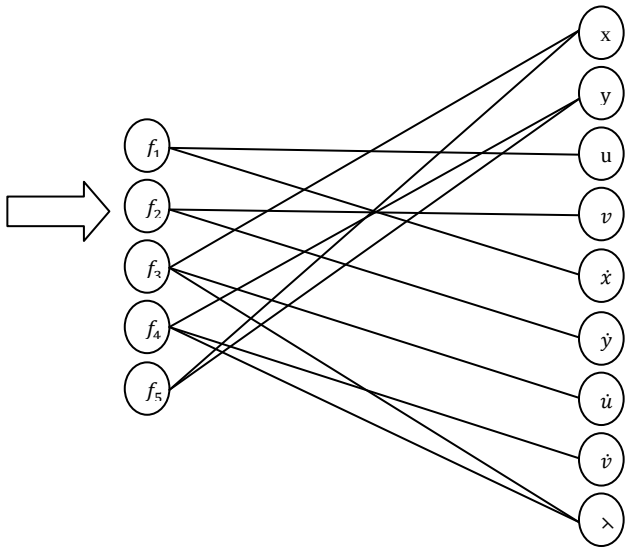
La complejidad del algoritmo será *O(m x (m x n))* (Podemos llegara a explorar en el caso peor todos los nodos) + *O(m x e(m x n))* (En el peor de los caos podemos volver a crear tantos nodos como el grafo tenga) ≈ *O(2m x e(m x n))*, *m* x *n* las dimensiones de la matriz, y *e* el número de nodos nuevos creados

El ejemplo utilizado para la implementación es el que se encuentra las diapositivas *Lecture in 12b* (Broman 2013), el cual también se ha expuesto en el primer capítulo de este trabajo: las ecuaciones del péndulo.

APLICACIÓN

Veamos un ejemplo. Sea la matriz de incidencia y el grafo de las ecuaciones del péndulo:

$$\begin{array}{c} \\ f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \end{array} \begin{array}{c} \begin{array}{ccccccccc} x & y & u & v & \dot{x} & \dot{y} & \dot{u} & \dot{v} & \lambda \end{array} \\ \begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$



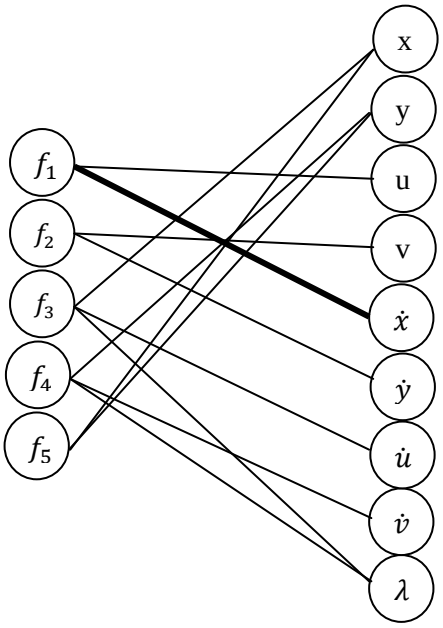
Como puede observarse el número de ecuaciones es insuficiente para poder despejar todas las incógnitas, por lo que tenemos que aplicar el algoritmo de Pantelides.

Inicialmente nuestro vectores de asignación y derivadas, y la pila serán:

vmap = $\{x \rightarrow \dot{x}, y \rightarrow \dot{y}, u \rightarrow \dot{u}, v \rightarrow \dot{v}\}$
eqmap = { }
assign = { }
C = { }

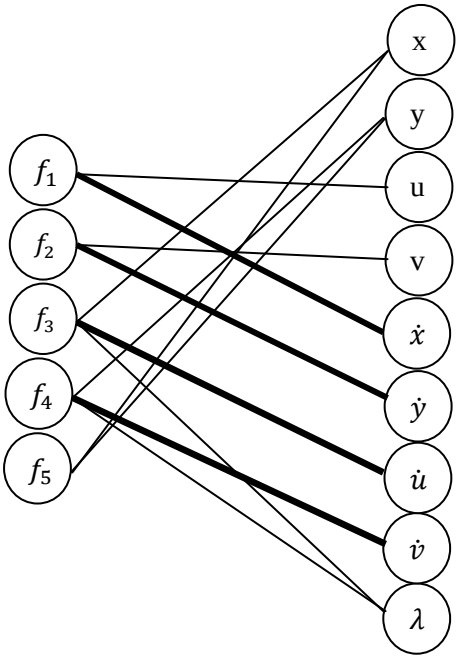
Empezamos a buscar emparejamiento para f_1, nuestra función *eq_match* nos devuelve que ha encontrado el nodo $\dot{x}$ como pareja, obteniendo:

vmap = $\{x \rightarrow \dot{x}, y \rightarrow \dot{y}, u \rightarrow \dot{u}, v \rightarrow \dot{v}\}$
eqmap = { }
assign = $\{\dot{x} \rightarrow f_1\}$
C = $\{f_1\}$



Este proceso, encontrar directamente los emparejamientos, se repite para los nodos f_2, f_3 y f_4, quedando nuestro grafo y vectores:

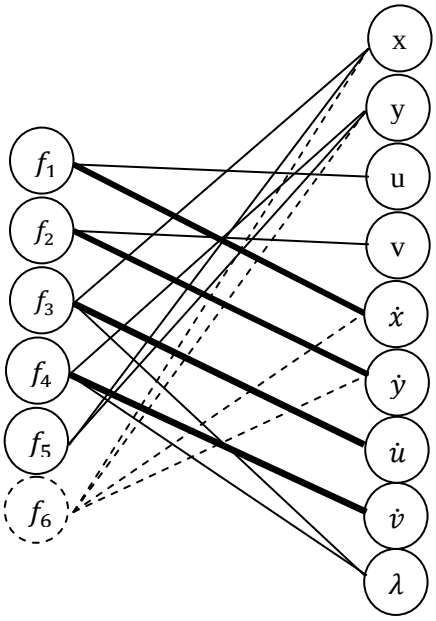
vmap = $\{x \rightarrow \dot{x}, y \rightarrow \dot{y}, u \rightarrow \dot{u}, v \rightarrow \dot{v}\}$
eqmap = { }
assign = $\{\dot{x} \rightarrow f_1, \dot{y} \rightarrow f_2, \dot{u} \rightarrow f_3, \dot{v} \rightarrow f_4\}$
C = $\{f_4\}$



Pero cuando llegamos a f_5, vemos que no está conectado a ninguna incógnita, sino a variables de estado. Nuestra función *eq_match* nos dirá que no hay emparejamiento y tendremos que crear tantos nodos nuevos como elementos halla en la pila *C*.

vmap = $\{x \rightarrow \dot{x}, y \rightarrow \dot{y}, u \rightarrow \dot{u}, v \rightarrow \dot{v}\}$
eqmap = $\{f_5 \rightarrow f_6\}$
assign = $\{\dot{x} \rightarrow f_1, \dot{y} \rightarrow f_2, \dot{u} \rightarrow f_3, \dot{v} \rightarrow f_4\}$
C = $\{f_5\}$

Tenemos que crear un nodo nuevo f_6 derivado de f_5 y establecer todas las nuevas conexiones dentro del grafo:

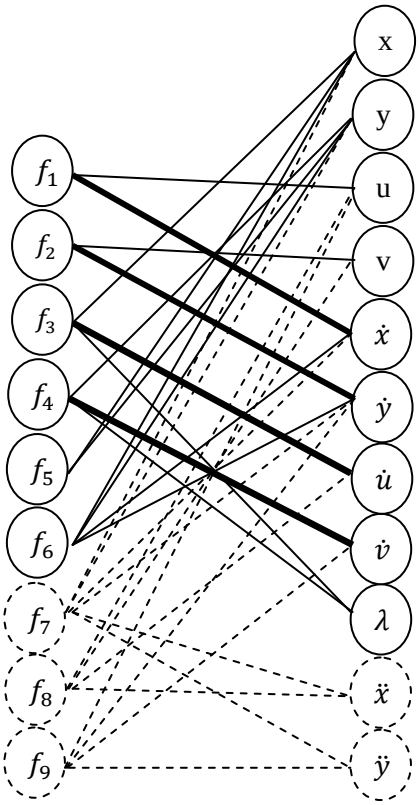


Continuamos explorando por f_6 y volvemos a encontrarnos en la misma situación de f_5 aunque en este caso la pila *C* indica que hemos tenido que recorrer un mayor camino dentro del grafo para encontrar el emparejamiento, por lo que tenemos que derivar varios nodos:

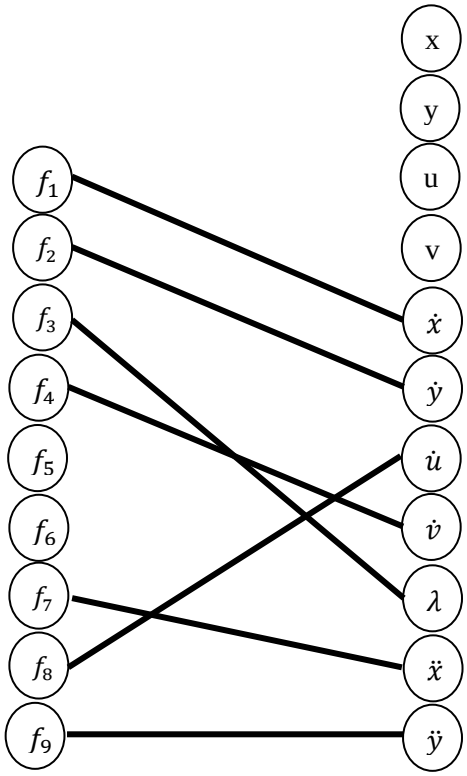
vmap = $\{x \rightarrow \dot{x}, y \rightarrow \dot{y}, u \rightarrow \dot{u}, v \rightarrow \dot{v}\}$
eqmap = $\{f_5 \rightarrow f_6\}$
assign = $\{\dot{x} \rightarrow f_1, \dot{y} \rightarrow f_2, \dot{u} \rightarrow f_3, \dot{v} \rightarrow f_4\}$
C = $\{f_6, \dot{x}, f_1, \dot{y}, f_2\}$

Creamos los nuevos nodos distinguiendo que sean tipo función o tipo variable, y establecemos las nuevas conexiones en nuestro grafo. También actualizaremos nuestros vectores, obteniendo:

vmap = $\{x \rightarrow \dot{x}, y \rightarrow \dot{y}, u \rightarrow \dot{u}, v \rightarrow \dot{v}, \dot{x} \rightarrow \ddot{x}, \dot{y} \rightarrow \ddot{y}\}$
eqmap = $\{f_5 \rightarrow f_6, f_6 \rightarrow f_7, f_1 \rightarrow f_8, f_2 \rightarrow f_9\}$
assign = $\{\dot{x} \rightarrow f_1, \dot{y} \rightarrow f_2, \dot{u} \rightarrow f_3, \dot{v} \rightarrow f_4, \ddot{x} \rightarrow f_8, \ddot{y} \rightarrow f_9\}$
C = $\{f_6, \dot{x}, f_1, \dot{y}, f_2\}$



Continuaremos por la derivada de f_6, f_7, y en este caso obtenemos que hemos encontrado el emparejamiento recolocando algunas de las parejas en *eq_match*, dejando cubiertas todas las incógnitas. Por tanto nuestro grafo de emparejamiento y vectores finales serían:



vmap = $\{x \to \dot{x}, y \to \dot{y}, u \to \dot{u}, v \to \dot{v}, \dot{x} \to \ddot{x}, \dot{y} \to \ddot{y}\}$
eqmap = $\{f_5 \to f_6, f_6 \to f_7, f_1 \to f_8, f_2 \to f_9\}$
assign = $\{\dot{x} \to f_1, \dot{y} \to f_2, \dot{u} \to f_8, \dot{v} \to f_4, \ddot{x} \to f_7, \ddot{y} \to f_9, \lambda \to f_3\}$
C = $\{f_7, \dot{x}, f_8, \dot{u}, f_3\}$

4.3. RESULTADOS

Veamos el comportamiento del algoritmo con varios problemas de diferente tamaño:

	Algoritmo de Panteides		
	m x n = 5 x 9	m x n = 10 x 20	m x n = 18 x 18
Tiempo (seg)	0.102	0.598	0.745
Nodos nuevos creados	6	8	7
Iteraciones	7	11	12

Tabla 4.1. Algoritmo de Pantelides

En la ejecución hemos usado dos ejemplos en los que el número de ecuaciones es inferior al de incógnitas y el tercero algunas de las ecuaciones eran redundantes obligando a crear nuevas.

Como se puede observar la creación de nodo no suele superar el la dimensión mayor de la matriz, ya que el algoritmo prioritiza la recolocación de los emparejamientos antes de crear ningún nodo nuevo, yéndose la mayor parte del tiempo de ejecución en la parte de exploración.

4.4. CONCLUSIONES.

En este capítulo hemos visto un ejemplo de cómo podernos enfrentar al caso en que tengamos insuficientes ecuaciones para poder definir adecuadamente nuestras incógnitas.

El algoritmo de Pantelides se apoya en la teoría de grafos, principalmente en la parte de exploración o coloración del grafo, intentando "recolocar" los emparejamientos para evitar la creación de nuevas derivadas.

Por otro lado, todas las operaciones que realiza son simbólicas, es decir, en ningún momento el algoritmo calcula la derivadas de las funciones o las variables, sino que nos dice que nos esperemos hasta el emparejamiento final, donde ya nos indica qué operaciones son las que

tenemos que realizar realmente. Lo que supone un gran ahorro en tiempo, pese a que lo tengamos que sacrificar el espacio al tener que almacenar la estructura del grafo.

CAPÍTULO 5
LAZOS ALGEBRAICOS

5.1. INTRODUCCIÓN

A veces cuando obtenemos la BLT de la matriz de incidencia de nuestro modelo, la diagonal está compuesta por bloques de dimensión mayor a 1. Dichos bloques, son lo que conocemos como lazos algebraicos, y en ellos para hallar el valor de las variables desconocidas, dependemos de ellas mismas.

En este capítulo vamos a explicar dos métodos de *tearing*, uno de ellos apoyándose en el algoritmo de Tarjan de búsqueda de componentes fuertes. En este último caso desharemos los bloques formados en la matriz BLT, expresada en forma de grafo, desechando aquellas aristas de los ciclos que supongan menor peso a la hora de resolver, es decir, seguimos una estrategia de ramificación y poda.

5.2. ALGORITMOS DE RASGADURA (TEARING)

Los algoritmos numéricos, como el de eliminación *gaussiana*, calculan los valores de las incógnitas a través del método de dependencia de ecuaciones, obteniendo el valor exacto de cada una de ellas.

En cambio, los algoritmos de rasgadura buscan mediante la relajación de alguna de las variables del modelo (*variables de rasgadura*), mejorar las condiciones impuestas por el modelo de manera que se deshagan los lazos algebraicos, o lo que es lo mismo los ciclos formados dentro del grafo del modelo.

5.2.1. ALGORITMO DE SUSTITUCIÓN

En el artículo *Methods for tearing systems of equations in object-oriented modeling* (H. Elmqvist y M. Otter June 1994) nos propone la forma estándar de los algoritmos de rasgadura. La idea es crear copias de los nodos variable para relajar las condiciones, emparejando la función que estaba asignada al nodo copia y así deshacer el ciclo.

Dado un grafo bipartito, en cada una de las componentes fuertes, dada su dimensión *n*, seleccionamos *n-1* de los nodos variable, *x*, y realizamos una copia x' a la que asignamos la ecuación *f* que estaba previamente emparejada con *x*. Por otro lado, el cálculo de cada variable *x* será mediante un algoritmo de sustitución. En el ejemplo anterior dado el modelo:

$$\begin{aligned} f_1(x_1, x_2, x_3) &= 0 \\ f_2(x_1, x_2, x_3) &= 0 \\ f_3(x_1, x_2, x_3) &= 0 \end{aligned} \qquad (5.1)$$

Dado *initxi* el valor inicial dado para el problema para la nueva variable *newxi*, el algoritmo de sustitución en pseudocódigo será:

PASO 0. *Inicialización de las nuevas variables:* newx2 = initx2 y newx3 = initx3. Ir a PASO 1.

PASO 1. *Asignación* de los valores relajados a x2 y x3: x2 = newx2 y x3 = newx3. Ir a PASO 2.

PASO 2. Calculamos x1 (El único sin nodo copia) a partir de los valores iniciales en la funcion con la que está emparejado**:** x1 = f1inv1(x2, x3). Ir a PASO 3.

PASO 3. Calculamos tambien newx2 y newx3, ya conocido x1, con las funciones que tienen emparejadas**:** newx2 = f2inv2(x1, x3) y newx3 = f3inv3(x1, x2). Ir a PASO 4.

PASO 4. Si converge(x2,newx2) y converge(x3,newx3), entonces FIN; En caso contrario ir a PASO 1.

Algoritmo 5.1. Algoritmo de sustitución

Si nos fijamos en la Figura 5.1, vemos como se introducen en el grafo original las nuevas variables creadas NEWx2 y NEWx3.

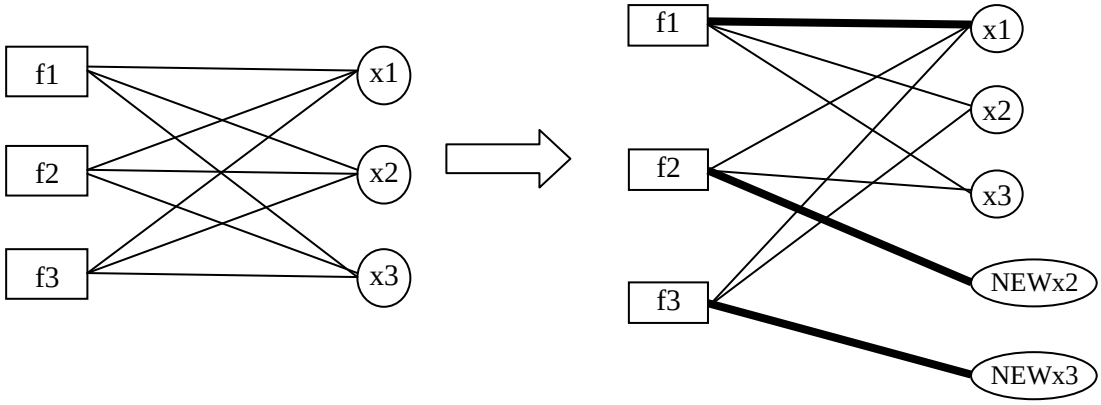


Figura 5.1. Introducción de los nuevos nodos relajados

En cuanto al cálculo de sus valores, vendrá dado por la función:

$NEWxi_0 = x_{i0}$ (Solución inicial) (5.1)

$NEWxi_j = f_j(y1, \dots, yn, NEWx1_{j-1}, \dots NEWxm_{j-1})$

La función f_j que aparece en la Ec. (5.1) será aquella ecuación asignada a *xi* mediante el emparejamiento inicial del modelo. En cada iteración calcularemos la variable sustituyendo el valor del resto en f_j, tanto las que han necesitado relajarse como las que no, y después compararemos si el valor obtenido tanto en esta iteración como en la anterior converge por debajo de una criterio dado.

La solución encontrada en cada ciclo mediante este proceso será óptima y más precisa en cuanto el criterio de convergencia sea más pequeño. Pero a costa de esto también hay un gran sacrificio en tiempo computacional. La convergencia no siempre puede darse de forma rápida y puede tardar muchas vueltas hasta que se cumplan los criterios.

Otro problema que presenta este método, es el hecho de que en caso con una gran cantidad de variables, la complejidad se multiplica exponencialmente, es decir, puede darse el caso de que haya que crear tantas funciones de convergencia como variables haya en el modelo, haciendo poco eficiente este algoritmo. A continuación mostraremos una alternativa, que aunque no garantiza una solución óptima en todos los casos, es más eficiente para casos grandes.

5.2.2. ALGORITMO DE CORTE

En este caso vamos a ver un algoritmo propuesto en el artículo *Partitioning and tearing of networks - applied to process flowsheeting Identification and Control Modeling* (T. Gundersen y T. Hertzberg 1983), que la estrategia que sigue se basa en algoritmos de ramificación y poda de árboles, en donde "cortamos" aquellas conexiones del grafo donde la solución no sea optima. En nuestro caso concreto, cortamos las conexiones que den lugar a ciclos y así deshacer los lazos algebraicos. Los nodos por donde se realizarán los cortes, serán nuestras *variables de rasgadura*. Este algoritmo no es óptimo, pero resulta de gran utilidad en problemas de gran tamaño donde puede llegar a ser bastante eficiente.

Para la detección de los ciclos nos apoyaremos en el algoritmo de Tarjan visto en la Sección 3.2.4 del Capítulo 3, aplicando el criterio de corte en aquellas componentes fuertes con dimensión mayor a 1.

El algoritmo primero lanzará el algoritmo de eliminación para hallar un emparejamiento inicial y así formar el grafo de dependencia y obtener las componentes fuertes del grafo inicial.. A continuación apuntamos predecesores y sucesores de cada nodo dentro de nuestro grafo de dependencia. Como el objetivo de nuestro algoritmo es deshacer los ciclos, o lo que es equivalente, hacer todas las componentes fuertes sean de dimensión 1, realizaremos los siguientes pasos hasta conseguirlo:

(i). Cogemos una de las componentes fuertes con dimensión superior a uno, y calculamos las ponderaciones de sus conexiones.

Sea *C* el conjunto de nodos de la componente fuerte que hemos escogido. Entonces para cada nodo $k \in C$, calculamos las sumas de los pesos de las sus predecesores y sucesores:

$$pre_k = \sum_{\substack{i \in C \\ i \neq k}} w_{ik}^{+} \; y \; suc_k = \sum_{\substack{i \in C \\ i \neq k}} w_{ki}^{-} \qquad (5.2)$$

(ii). Cogemos aquel nodo *k* que cumpla $min\left({}^{pre_i}/_{suc_i}\right), i \in C$.

(iii). Cortamos cualquier conexión que llegue al nodo *k* dentro del grafo, incluyendo la que forma el ciclo actual.

(iv). Volvemos a lanzar el algoritmo de Tarjan y si no cumplimos que el número de componentes fuertes sea igual al de nodos volvemos a (i).

Con las componentes fuertes halladas en la ejecución final del algoritmo de Tarjan, utilizamos el sistema de permutación explicado en el algoritmo de triangulación BLT, visto en la Sección 3.2.3 del Capítulo 3.

PASO 0. *Inicialización.* Cálculo emparejamiento inicial y el grafo de dependencia. Ir a PASO 1

PASO 1. *Relaciones.* Se construyen los vectores de predecesores y sucesores de cada nodo dentro del grafo de dependencia. Ir a PASO 2

PASO 2. *Tarjan.* Hallamos las componentes fuertes del grafo de dependencia inicial. Ir a PASO 3

PASO 3. *Bucle principal.* Si el número de nodos no sea igual a la dimensión de las componentes fuertes, entonces vamos al PASO 4. Si no al PASO 7.

PASO 4. *Ponderaciones.* Para cada componente fuerte con dimensión superior a uno, y calculamos las ponderaciones de sus conexiones. Sea *C* el conjunto de nodos de la componente Entonces para cada nodo $k \in C$, calculamos:

$$pre_k = \sum_{\substack{i \in C \\ i \neq k}} w_{ik}^{+} \; y \; suc_k = \sum_{\substack{i \in C \\ i \neq k}} w_{ki}^{-}$$

y cogemos aquel nodo *k* que cumpla $min\left({}^{pre_i}/_{suc_i}\right), i \in C$. Ir a PASO 5

PASO 5. *Corte por k:* Dentro del vector de sucesores eliminamos *k* de todos los nodos.

PASO 6. *Tarjan.* Hallamos las componentes fuertes del nuevo grafo de dependencia. Ir a PASO 3.

PASO 7. Apuntamos los nodos que pertenecen a las componentes fuertes de dimensión 1

Algoritmo 5.2. Algoritmo de corte

IMPLEMENTACIÓN

El código corresponde con el proyecto *.../Códigos/Capitulo 4/RamCorte/RamCorte.cbp* y se encuentra en el Anexo C.

Por convenio vamos a considerar que el peso de todas las aristas del grafo de dependencia sean $w_{ij} = 1$, aunque perfectamente podrían generalizarse a cualquier valor.

Los datos de entrada que deberá proporcionar el usuario serán la matriz de incidencia (sin las variables de estado), y el número de filas y columnas que tiene dicha matriz.

El paso 0 consiste simplemente en lanzar el algoritmo de eliminación y apuntar los emparejamientos que nos servirán para crear los vectores de antecesores y predecesores de cada nodo del grafo de dependencia, y con este último realizamos una llamada inicial al algoritmo de Tarjan, para obtener las componentes fuertes del modelo original.

A continuación nos encontramos con el bucle principal del algoritmo en el que se desarrolla el proceso de ramificación y poda: localizamos una componente fuerte con dimensión mayor a 1, calculamos los alfas, podamos por aquel que sea mínimo y volvemos a lanzar el algoritmo de Tarjan para comprobar si el número de componentes fuertes es inferior al número de ecuaciones.

Para el cálculo de las alfas, utilizamos la función *calc_alfa* en la que se implementan los sumatorios vistos en el punto 3(ii) y devolvemos la división controlando que en el caso de que el sumatorio de los sucesores sea 0, entonces devolvemos que el alfa de ese nodo es 0.

La complejidad del algoritmo vendrá dada por la suma de las complejidades de las diferentes partes del mismo. Sabemos que el algoritmo de Tarjan es de $O(n * e)$, n dimensión de la matriz y e el número de aristas del grafo; el algoritmo de emparejamiento del $O(n^2)$. Lo que nos queda es ver el bucle principal.

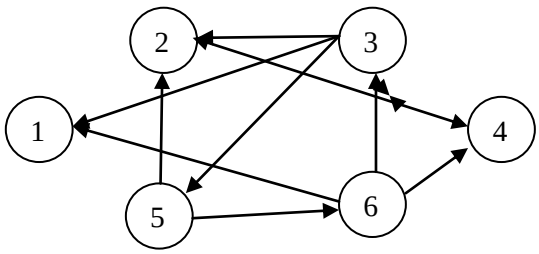
Tanto como la búsqueda de la componente fuerte, como el marcado de las ecuaciones y al cálculo del mínimo alfa son lineales $O(n)$, pero el cálculo de cada alfa (*calc_alfa*) supone recorrer todos los elementos de predecesores y sucesores que en casos extremos puede ser *n-1* veces, y hacemos la llamada a la función *n* veces, por lo que será de $O(n^2)$.

Así mismo, en el peor de los casos el bucle principal se puede repetir *n* veces, lo que implica que su complejidad sería $O(n * (n^2 + n * e + 3n)) \approx O(n^3 + n^2 * e)$.

APLICACIÓN

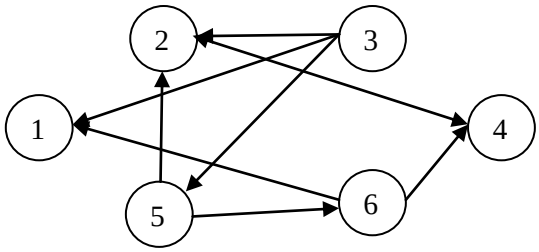
Tomemos el ejemplo que vimos en la Sección 3.2.4, y deshagamos los bloques de la matriz de incidencia. Para ello, recordemos cual era el emparejamiento inicial y el grafo de dependencia:

Emparejamiento inicial : $(f_2, v_1), (f_3, v_2), (f_6, v_3), (f_1, v_4), (f_4, v_5), (f_5, v_6)$



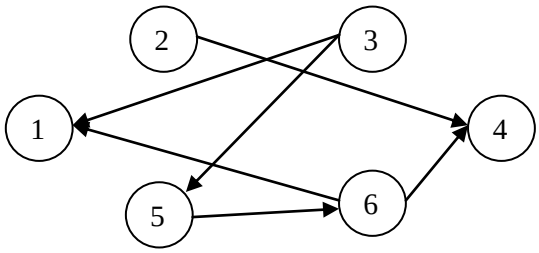
Lanzamos el algoritmo de *Tarjan* y obtenemos las componentes fuertes {{3,5,6},{2,4},{1}}. Como podemos observar no todas son de dimensión 1, por lo que tenemos que podar.

Sea la componente fuerte{3,5,6}. Calculamos alfa y vemos que podemos podar por 3. Entonces podaremos la arista (6,3):



Volvemos a lanzar el algoritmo de *Tarjan* y obtenemos{{3}{5},{6},{2,4},{1}}

Repetimos el proceso para la componente fuerte {2,4}. El mínimo estará en 2 y podaremos: (3,2), (5,2), (4,2).



Volvemos a lanzar el algoritmo de *Tarjan* y las componentes fuertes son : {3},{5}, {6}, {2}, {4}, {6}

5.3. RESULTADOS

Veamos su rendimiento de tamaño frente al tiempo en la Tabla 5.1

	Algoritmo de corte			
	n = 6	n = 10	n = 15	n = 20
Tiempo (seg)	0.118	0.146	0.169	0.183
Número de cortes	4	10	41	65
Iteraciones	2	3	4	7

Tabla 5.1. Algoritmo de Ramificación y Poda

Con *n* pequeño observamos que el incremento del tiempo es normal, pero cuando ya nos fijamos en el caso de 20 vemos que en la relación tamaño/tiempo el algoritmo se va estancando, pese a realizar gran cantidad de cortes. Esto afirma lo que ya habíamos comentado, según el *n* es más grande el algoritmo es más eficiente y el número de vueltas que realiza no crece demasiado.

5.4. CONCLUSIONES

Todos los algoritmos de *tearing* se caracterizan por la búsqueda de nodos dentro de los ciclos que cumplan un criterio determinado y "romper" los lazos algebraicos, sea por la introducción de nuevos nodos con el valor relajado de la variable, o cortando literalmente las aristas del grafo como hemos visto en este último ejemplo.

Ya centrándonos en el algoritmo de la Sección 5.3.2, vemos que volvemos a apoyarnos en la teoría de grafos, más concretamente en el algoritmo de ramificación y poda, donde desechamos los caminos menos óptimos, lo que traduciría en nuestro caso, aquello que deshagan los ciclos afectando lo menos posible al modelo en general.

Por último comentar, el comportamiento del algoritmo observado en la tabla de resultados. Según se aumentaba la dimensión del problema, la velocidad de crecimiento de la función de tamaño frente al tiempo se relajaba.

CAPÍTULO 6
CONCLUSIONES Y TRABAJOS FUTUROS

6.1. INTRODUCCIÓN

En la Sección 6.2 de este capítulo haremos una compilación de todos los algoritmos vistos para las partes análisis y optimización del proceso realizado por OpenModelica a la hora de simular modelos físicos, descrito en la Sección 1.1 del Capítulo 1, así un pequeño comentario sobre el comportamiento de cada uno de ellos

En la Sección 6.3, haremos una serie de propuestas de futuras líneas de estudio que podrían ampliar las ideas comentadas en este trabajo, como por ejemplo la introducción de la implementación de métodos numéricos para comparar su rendimiento frente a los algoritmos aquí expuestos, o la creación de una implementación completa de los fases de análisis y optimización de la Sección 1.1

6.2. CONCLUSIONES

A lo largo del trabajo se han mostrado algunos ejemplos de algoritmos que podrían ser utilizados para simplificar y/o resolver sistemas de ecuaciones diferenciales en las fases de análisis y optimización del proceso de resolución de sistemas de ecuaciones diferenciales en OpenModelica. Concretamente, nos hemos centrado en algoritmos, no de cálculo numérico como la *eliminación gaussiana* o las ecuaciones de *Newton*, sino de la teoría de grafos.

A la hora de simular un modelo físico, inicialmente, se comprueba si nos encontramos en la situación ideal, mismo número de ecuaciones que variables y se pueden asignar una a una sin repeticiones (problema no singular). En este caso hemos visto los algoritmos de eliminación, *score* y el de ordenación BLT. El algoritmo más potente de los tres ha sido el de ordenación BLT en que nos hemos apoyado en la teoría de grafos, más concretamente en el algoritmo de Tarjan de búsqueda de componentes fuertes, donde se aprovechan las propiedades de los grafos fuertemente conexos y los algoritmos de coloración o marcado. En el caso del algoritmo de

eliminación, resulta útil para la búsqueda de emparejamiento iniciales usadas en los algoritmos de *tearing* para lazos algebraicos. Por último comentar que la sección de resultados, cuando se ha comparado complejidades y tiempos entre los algoritmos, el más eficiente ha resultado ser el de ordenación BLT.

Continuando con el proceso descrito en la Sección 1.1, puede darse el caso de que cuando aplicamos nuestro algoritmo de ordenación BLT, en la diagonal de la matriz de incidencia ya permutada nos encontremos bloques, o lo que es lo mismo lazos algebraicos. Aquí, hemos visto ejemplos de algoritmos de *tearing*, tomando la matriz de incidencia para construir el grafo de dependencia, e interpretar los lazos como ciclos dentro él que debemos romper. En concreto se ha visto en detalle un algoritmo basado en la estrategia de ramificación y poda según el número de predecesores y sucesores tenga cada nodo del ciclo, y que ha resultado ser más eficiente en cuanto mayor sea el tamaño del problema como ha podido observarse en la Tabla 5.1.

El último caso visto en este trabajo ha sido en el que el número de ecuaciones sea diferente al de incógnitas, encontrándonos ante una *singularidad estructural*. Para resolverla hemos visto una versión del algoritmo de reducción de índice de *Pantelides*, en el que se sigue la estrategia de introducción de nuevos nodos que "representan" la supuesta derivada candidata a ayudar a la resolución del modelo. Gracias al algoritmo de *Pantelides* se evita el cálculo de las derivadas de las ecuaciones hasta que se llega a obtener el conjunto de óptimo, no teniendo calcular todas sino las que quedan emparejadas al final del algoritmo.

Para finalizar, un último comentario referente a la relación entre las diferentes fases y los algoritmos expuestas en el trabajo se relacionan entre sí. Hemos visto que pese a que nos encontramos en situaciones más complejas (singularidades estructurales y lazos algebraicos), los algoritmos utilizados se basan, al menos en la estrategia, en ideas contadas en procesos anteriores. Por ejemplo, en los algoritmos de *tearing* vistos aquí, necesitamos del cálculo de una solución inicial, la cual se puede calcular mediante el algoritmo de eliminación de la Sección 3.2.1 del Capítulo 3.

6.3. TRABAJOS FUTUROS

Como se ha comentado antes, el trabajo se ha centrado en los algoritmos más simbólicos, muchos de ellos basados en la teoría de grafos, por lo que resultaría interesante estudiar la parte de algoritmos numéricos utilizados en las fases de análisis y optimización, y así comparar sus

rendimiento frente a los algoritmos expuestos en el presente trabajo. También en el caso de los lazos algebraicos haber realizado un ejemplo de algoritmo de relajación.

También sería interesante implementar otros algoritmos para la reducción del índice y compararlos con el algoritmo de Pantelides.

Otra línea futura de estudio, sería realizar un algoritmo que incluya todas las transformadas realizadas sobre el modelo físico en las fases de análisis y optimización de la Sección 1.1, incluyendo cálculo de particiones, eliminación de ecuaciones triviales y redundantes, resolución de problemas con singularidades estructurales y ruptura de lazos algebraicos.

ANEXO A. ALGORITMOS DE PARTICIÓN

En el presente anexo se mostrarán los códigos correspondientes a los algoritmo presentados en el Capítulo 3.

1. ALGORITMO DE ELIMINACIÓN.

```
#include <iostream>
#include <cstdlib>
#include <stdio.h>
#include <vector>

using namespace std;
void imprimir (vector<int> v, vector<int> w) {

    cout << "Los emparejamientos encontrados son (variable, ecuacion): ";
    for (unsigned k=0; k<v.size(); k++) {
        cout << "(" << v[k]<<","<<w[k]<<") ";
    }
    cout << '\n';
}

int minimo(vector<int> v) {
    int k,mini, posicion =0;
    mini = v[0];
    for (k=1; k< v.size(); k++) {
        if (v[k] < mini) {
            mini = v[k];
            posicion = k;
        }
    };
    return posicion;
}

int minimoreduc(vector<int> v, vector<int> s, vector<int> w) {
    vector<int> aux;
    int i, smin, posicion;

    for (i=0; i < v.size(); i++) {
        if (w[v[i]]==1) {
            aux.push_back(i);
        }
    }
    smin = s[aux[0]];
    posicion = aux[0];
    for (i=1; i < aux.size(); i++) {
        if (s[aux[i]]<smin) {
            smin = s[aux[i]];
            posicion = aux[i];
        }
    }
    return posicion;
}
main() {
    //DATOS
```

```
49      int f=10;
50      int c=10;
51      int m[10][10] = {{1, 0, 0, 0, 0, 0, 0, 0, 0, 0},
52          {0, 0, 1, 1, 0, 0, 0, 0, 0, 0},
53          {0, 0, 0, 0, 1, 1, 0, 0, 0, 0},
54          {0, 0, 0, 0, 0, 0, 1, 1, 0, 0},
55          {0, 0, 0, 0, 0, 0, 0, 0, 1, 1},
56          {1, 0, 1, 0, 0, 0, 0, 0, 0, 0},
57          {0, 0, 1, 0, 1, 0, 1, 0, 0, 0},
58          {0, 0, 0, 0, 1, 0, 0, 0, 0, 0},
59          {0, 1, 0, 1, 0, 0, 0, 0, 0, 0},
60          {0, 0, 0, 1, 0, 1, 0, 0, 0, 1}
61      };
62
63      //VARIABLES A UTILIZAR
64      vector<int> ecua (f);
65      vector<int> var (c);
66      vector<int> sumasc (c);
67      vector<int> sumasf (f);
68      vector<int> datos(f);
69      vector <int> parvar;
70      vector <int> parecua;
71      int i,j,k,e1,e2,resto,ind;
72      bool encontrado,fin;
73
74      //Inicializamos el vector que nos indicara que ecuaciones siguen disponibles
75
76      for (i=0; i <f; i++) {
77          ecua[i]=i;
78          var[i]=i;
79      }
80
81      //Inicializamos el array con las sumas que indicaran cual es el prioritario
82
83      for (j=0; j <c; j++) {
84          for (i=0; i <f; i++) {
85              sumasc[j] = sumasc[j] + m[i][j];
86              sumasf[j] = sumasf[j] + m[j][i];
87          }
88      }
89      //Estas sumas determinaran cual variable elegir a la hora de eliminar,
        siempre la de menor importe.
90
91      fin = false; //Marca si hemos hecho todos los emparejamientos
92      do {
93          //Seleciono la variable y columna que tienen mayor prioridad
94          e1 = minimo(sumasc);
95          e2 = minimo(sumasf);
96          ind = 0;
97          encontrado = false;
98          if (sumasc[e1] < sumasc[e2]) {
99              cout<< "El minimo esta por las variables: "<<sumasc[e1]<<" posicion
                "<< var[e1] <<endl;
100             //Extraemos la columna corresondiente a e1
101             for (k=0; k<f; k++) {
102                 datos[k] = m[k][var[e1]];
103             }
104             cout << "La columna extraida es la "<<var[e1]<<":"<<endl;
105             for (unsigned k=0; k<c; k++) {
106                 cout << ' ' << datos[k];
107             }
108             cout << '\n';
109             //Hallamos la fila de mayor prioridad para e1
```

```
110             e2 = minimoreduc(ecua, sumasf, datos);
111             cout<< "La fila obtenida es: "<< ecua[e2]<<endl;
112         } else {
113             cout<< "El minimo esta por las ecuaciones: "<<sumasc[e2]<<" posicion
                "<< ecua[e2] <<endl;
114             //Extraemos la fila correspondiente a e2
115             for (k=0; k<f; k++) {
116                 datos[k] = m[ecua[e2]][k];
117             }
118             cout << "La fila extraida es la "<<ecua[e2]<<":"<<endl;
119             for (unsigned k=0; k<f; k++) {
120                 cout << ' ' << datos[k];
121             }
122             cout << '\n';
123             //Hallamos la columna de mayor prioridad para e2
124             e1 = minimoreduc(var, sumasc, datos);
125             cout<< "La variable obtenida es: "<< var[e1] <<endl;
126         }
127 
128         //ACTUALIZACION DE DATOS
129 
130         //AÃ±adimos la nueva pareja encontrada
131         parvar.push_back(var[e1]);
132         parecua.push_back(ecua[e2]);
133         cout <<"Pareja encontrada (v,e): ("<<parvar [ind]<<","<<parecua[ind]<<")
                                    "<<endl;
134         //Actualizamos el vetor de sumas con los datos de la fila que quitamos
135         for (k=0; k<sumasc.size(); k++) {
136             sumasc[k] -=  m[ecua[e2]][var[k]];
137         }
138         //Actualizamos el vetor de sumas con los datos de la columna que
           quitamos
139         for (k=0; k<sumasf.size(); k++) {
140             sumasf[k] -=  m[ecua[k]][var[e1]];
141         }
142         //Borramos el numero de ecuaciÃ³n de nuestra lista
143         ecua.erase (ecua.begin()+e2);
144         //Borramos el numero de la variable de nuestra lista
145         var.erase (var.begin()+e1);
146         //Quitamos la suma de la variable que hemos asignado
147         sumasc.erase (sumasc.begin()+e1);
148         //Quitamos la suma de la ecuacion que hemos asignado
149         sumasf.erase (sumasf.begin()+e2);
150         ind +=1;
151         fin = (var.empty());
152     } while (!fin);
153     imprimir(parvar,parecua);
154     return 0;
155 }
```

2. ALGORITMO DE SCORE

En el Apartado 1 se expondrá el código completo del algoritmo, mientas que en el segundo solo aquella partes que se han visto modificadas al introducir la función ponderada.

2.1. ESTÁNDAR

```
#include <iostream>
#include <cstdlib>
#include <stdio.h>

using namespace std;
void imprimir (int a, int b, int m[6][6]) {

    int i,j;

    for (i=0; i<a; i++) {
        for (j=0; j<b; j++) {
            printf("\t");
            cout << m[i][j];
//      printf("%0.2f\t " ,m[i][j]);
        }
        printf("\n");
    }
    printf("============================================================");
    printf("\n");
}

int main() {
    //DATOS
    int f =6;
    int c = 6;
    int m[][6] = {{1, 0, 0, 1, 2, 0},
        {1, -1, 1, 2, -3, 2},
        {-6, 1, 1, 3, 5, 2},
        {1, 0, 0, 3, -1, 0},
        {2, 0, 0, 5, 1, 0},
        {-9, 2, 1, 1, 7, 1}
    };
    //NOTA: tambien hay que cambiar las dimensiones en la funcion imprimir.


    //VARIABLES DEL PROGRAMA
    int i,j,h,elem, aux, ce; //Variables del bucle
    bool fin = false, fin2 = false,encontrado = false, fin3 = false;

    cout << "La matriz inicial es: ";
    printf("\n");
    imprimir (f,c,m);

    //ALGORITMO

    ce = c-1; //InicializaciÃ³n del elemento de la columna actual
    while (!fin) {
        elem = m[0][ce];
        cout << "El elemento actual a considerar es "<< elem << " en la posicion
        ("<<1<<","<<ce+1<<")"<< endl;
        if (elem == 0) {
            ce -= 1;
            cout << "Es cero ya. No necesita permutar"<<endl;
        } else { //El elemento es distinto de 0. Procedemos a hacer la busqueda
            fin2 = false;
            fin3 = false;
            encontrado = false;
            i = 0;
```

```
58              j = 0;
59              while (!fin2) {
60                  if (m[i][j] == 0) {
61                      //Sale del bucle conservando el valor de la columna que
                        buscamos
62                      encontrado = true;
63                  } else {
64                      j += 1;
65                      if (j == ce) {
66                          j = 0;
67                          i +=1;
68                          if (i == f) {
69                              fin3 = true;
70                          }
71                      }
72                  }
73                  //Terminamos si hemos encontrado la columna a permutar o hemos
                    agotado todas las opciones
74                  fin2 = (encontrado) || (fin3);
75              }
76              if (encontrado) {
77                  cout << "Intercambiamos las columnas: " << j+1 << " y " << ce+1 <<
                     endl;
78                  //Intercambiamos las columnas
79                  for (h=0; h<f; h++) {
80                      aux = m[h][ce];
81                      m[h][ce] = m[h][j];
82                      m[h][j] = aux;
83
84                  }
85
86
87                  cout << "Intercambiamos las filas: " << j+1 << " y " << ce+1 <<
                    endl;
88                  //Intercambiamos las filas
89                  for (h=0; h<c; h++) {
90                      aux = m[ce][h];
91                      m[ce][h] = m[j][h];
92                      m[j][h] = aux;
93                  }
94              } else {
95                  cout<<"No existe columna con la que pueda permutar"<<endl;
96              }
97          }
98          imprimir (f,c,m);
99          fin = (fin3) || (ce == 0);
100     }
101     cout << "Nuestra matriz triangular por bloques es: " << endl;
102     imprimir (f,c,m);
103
104     return 0;
105 }
```

2.2. CON FUNCIÓN PONDERADA

Introducimos la función *f_peso* para calcular las sumas ponderdas:

```
22 bool f_peso (int a, int b, int m[6][6], int cel, int fel) {
23     int maux [a][b];
24     int aux,h,k, diag =1;
```

```
25      int summ=0, summaux=0;
26
27      //Hacemos la copia de la matriz
28      for (h = 0 ; h < a ; h ++ )
29          for (k = 0 ; k < b ; k ++ ) maux[h][k] = m[h][k];
30
31      //Intercambiamos las columnas
32      for (h=0; h<a; h++) {
33          aux = maux[h][cel];
34          maux[h][cel] = maux[h][fel];
35          maux[h][fel] = aux;
36      }
37      //Intercambiamos las filas
38      for (h=0; h<b; h++) {
39          aux = maux[cel][h];
40          maux[cel][h] = maux[fel][h];
41          maux[fel][h] = aux;
42      }
43
44      //Calculamos las sumas
45
46      for (h=1; h<b; h++) {
47          for (k=0; k <diag; k++) {
48              if (maux[k][h] ==0) {
49                  summaux += ((a-k+1)^2)*(h^2);
50              }
51              if (m[k][h] ==0) {
52                  summ += ((a-k+1)^2)*(h^2);
53              }
54          }
55      }
56      return (summaux > summ);
57  }
```

Intercambiamos las líneas 53-75 del algoritmo original por las siguientes:

```
98          } else { //El elemento es distinto de 0. Procdemos a hacer la busqueda
99              fin2 = false;
100              fin3 = false;
101              encontrado = false;
102              i = fe;
103              j = 0;
104              while (!fin2) {
105                  if (m[i][j] == 0) {
106                      cout<<"Fila "<< i<<endl;
107                      cout << "Intercambiamos las columnas: " << j+1 << " y " <
                        < ce+1 << endl;
108                      encontrado = f_peso (f,c,m,ce,j);
109                  };
110                  cout<<encontrado<<endl;
111                  if (!encontrado) {
112                      j += 1;
113                      if (j == ce) {
114                          j = 0;
115                          i +=1;
116                          if (i == f) {
117                              fin3 = true;
118                          }
119                      }
120                  }
```

```
121                 //Terminamos si hemos encontrado la columna a permutar o hemos
                    agotado todas las opciones
122                 fin2 = (encontrado) || (fin3);
123             }
```

3. ALGORITMO DE ORDENACIÓN BLT Y TARJAN

Inicialmente tenemos la función *f_connect* en la que se desarrolla en algoritmo de Tarjan:

```
01 #include <iostream>
02 #include "Parejas.h"
03 #include <cstdlib>
04 #include <stdio.h>
05 #include <vector>
06
07
08 using namespace std;
09
10 void f_connect(int v, vector<vector <int> > conexiones,
11                 vector<int> &pila, int &ind, vector<vector <int> > &cfuertes,
                    vector<int> &lowlink, vector<int> &num,
12                 vector<bool> &enpila) {
13     int i,u,w;
14
15     //Inicializamos los datos de v y lo aÃ±adimos a la pila
16     ind +=1;
17     lowlink[v] = ind;
18     num[v] = ind;
19     enpila[v] = true;
20     pila.push_back(v);
21
22     //Exploramos cada uno de los nodos que salen de v y los clasificamos en caso
       de que existan.
23     if (!conexiones[v].empty()) {
24         for (i=0; i < conexiones[v].size(); i++) {
25             w = conexiones[v][i];
26             if ((num[w]==-1)||(enpila[w])) {
27                 //No se encuentra en la pila, continuamos explorando por w
28                 if (num[w]==-1) {
29                     f_connect(w, conexiones, pila, ind, cfuertes, lowlink, num,
                                  enpila);
30                 }
31                 //Marcamos el ciclo
32                 lowlink[v] = min(lowlink[v],lowlink[w]);
33             }
34         }
35     }
36
37     //Identificamos si tenemos una componente fuerte en nuestra pila, mirando si
       hay ciclo formado
38     if (lowlink[v] == num[v]) {
39         vector<int> aux;
40         do {
41             u = pila[pila.size()-1];
42             aux.insert(aux.begin(),u);
```

```
            enpila[u]= false;
            pila.pop_back();
        } while (u!=v);
        cfuertes.insert(cfuertes.begin(),aux);
    }
    return;
}
```

Y después, la función principal con el algoritmo de ordenación BLT.

```
int main() {
    //DATOS
    int f=6;
    int c=6;
    int m[6][6] =  {{0, 0, 1, 1, 0, 1},
        {1, 1, 0, 0, 1, 1},
        {0, 1, 1, 0, 0, 0},
        {1, 0, 1, 0, 1, 0},
        {0, 1, 0, 0, 0, 1},
        {0, 0, 1, 0, 0, 1}
    };
    //Nota: Hay que cambiar los valores de las dimensiones de las funciones en
    las que se pasa m como valor
    //Eso incluye: "parejas" en Parejas.cpp y Parejas.h, y la funcion imprimir
    en este archivo

    //Variables del programa

    int i,j;
    vector<int> assignv; //Marca a que numero de ecuacion se ha asignado cada
    variable
    vector<int> pila;//Pila utilizada en el algoritmo de Tarjan
    vector<vector <int> > conexiones; //Vector donde apuntaremos por cada
    ecuacion con quien esta conectado en el grafo de dependencia
    vector<vector <int> > cfuertes; //Lista donde apuntaremos las componentes
    fuertes
    vector<int> lowlink; //En la pila, numero de Nodo (num) con el que forma c.
                                                            fuerte o ciclo
    vector<int> num; //Indica el numero asignado a un Nodo una vez renumerado
    por el algoritmo
    vector <bool> enpila; //Marca si un Nodo esta en la pila
    int ind = 0; //Inicializacion del indice de nodo en tarjan
    int maux[f][c], BLT[f][c];
    vector<int> orden, ordenv;

    //INICIALIZACIÃ“N
    //Inicializamos assignv, lowlink, num y enpila
    for (i=0; i<c; i++) {
        assignv.push_back(-1);
        num.push_back(-1);
        lowlink.push_back(-1);
        enpila.push_back(false);
    }

    //Inicializamos conexiones
    for (i=0; i<c; i++) {
        vector <int> w;
```

```
90          conexiones.push_back(w);
91      }
92      //Inicializamos maux y BLT
93      for (i=0; i<f; i++) {
94          for (j=0; j<c; j++) {
95              maux[i][j] =0;
96              BLT[f][c] = 0;
97          }
98      }
99
100      //ALGORITMO
101
102      //Matriz inicial
103      cout<<"La matriz inicial es:"<<endl;
104      for (i=0; i<f; i++) {
105          for (j=0; j<c; j++) {
106              cout << m[i][j];
107          }
108          printf("\n");
109      }
110
111      //1. Hallamos el emparejamiento inicial
112      emparejar(f,c,m,assignv);
113      cout << "El emparejamiento inicial es (f,v): ";
114      for (unsigned k=0; k<assignv.size(); k++) {
115          cout << "(" << (assignv[k]+1) << "," << (k+1) <<") ";
116      }
117      cout << '\n';
118
119      //2. Construimos los nodos del grafo de dependencia y inicialzamos las
     variables del algoritmo de Tarjan
120      for (i=0; i<f; i++) {
121          for (j=0; j<c; j++) {
122              if ((m[i][j] == 1)&&(assignv[j]!= i)) {
123                  conexiones[assignv[j]].push_back(i);
124              }
125          }
126      }
127
128      //3. Lanzamos el algoritmo de Tarjan
129      for (i=0; i<c; i++) {
130          //Para aquells nodos a los que no hemos metido en la pila todavia
131          if (num[i]==-1) {
132              f_connect(i,conexiones, pila, ind, cfuertes, lowlink, num, enpila);
133          }
134      }
135
136      //3.5Sacamos las componentes fuertes por pantalla
137      cout<<"Las componentes fuertes son: ";
138      for (i=0; i<cfuertes.size(); i++) {
139          vector<int> aux = cfuertes[i];
140          cout<<"{";
141          for (j=0; j<aux.size(); j++) {
142              cout<<" "<<(aux[j]+1)<<" ";
143              orden.push_back(aux[j]);
144              ordenv.push_back(-1);
145          }
146          cout<<"} ";
147      }
148      cout<<endl;
149      //Hallamos el orden de las filas para la  permutacion
150      for (i=0; i<f; i++) {
151          j = -1;
```

```
152         do {
153             j+=1;
154         } while (orden[i]!= assignv[j]);
155         ordenv[i]=j;
156     }
157 
158     //4. Reordenamos la matriz
159     //Permutamos las filas
160     for (i=0; i<f; i++) {
161         for (j=0; j<c; j++) {
162             maux[i][j] = m[orden[i]][j];
163         }
164     }
165 
166     //Permutamos las columnas
167     for (j=0; j<c; j++) {
168         for (i=0; i<f; i++) {
169             BLT[i][j] = maux[i][ordenv[j]];
170         }
171     }
172 
173     cout<<"La matriz BLT es:"<<endl;
174     for (i=0; i<f; i++) {
175         for (j=0; j<c; j++) {
176             cout << BLT[i][j];
177         }
178         printf("\n");
179     }
180 
181     return 0;
182 }
```

ANEXO B. REDUCCIÓN DEL ÍNDICE

1. ALGORITMO DE PANTELIDES

Como representante de los algoritmo de reducción del índice tenemos el algoritmo de *Pantelides*. Dentro del código tenemos dos archivos representativos: aquellos correspondientes a la clase Nodo (cabeceras y cuerpo) y el algoritmo como tal.

1.1. CLASE NODO

Las cabeceras de la clase se encuentran en el archivo Nodo.h:

```
01 #ifndef NODO_H
02 #define NODO_H
03 #include <iostream>
04 #include <cstdlib>
05 #include <stdio.h>
06 #include <vector>
07 class Nodo {
08   public:
09     //Constructores
10     Nodo();
11     Nodo(const Nodo& other); //Constructor de copia. Crea uno nuevo con los
           datos de otro nodo
12     Nodo ( char t, int nu);
13
14     //Funciones que devuelven el valor
15     int get_numtipo();
16     char get_tipo();
17
18   private:
19     char tipo; //f si es ecuacion (fila) o v (columna) si es una variable
20     int numtipo; //Numero de fila o columna
21 };
22
23 #endif // NODO_H
```

Mientras que las definiciones de las funciones en Nodo.cpp:

```
01 #include "Nodo.h"
02 #include <cstdlib>
03 #include <stdio.h>
04 #include <vector>
05
06 Nodo::Nodo() {
```

```
07      numtipo = 0;
08      tipo = 'x';
09  }
10
11  Nodo::Nodo(const Nodo& other) {
12      numtipo =other.numtipo;
13      tipo = other.tipo;
14  }
15  Nodo::Nodo ( char t, int nu) {
16      tipo = t;
17      numtipo = nu;
18  }
19
20  char Nodo::get_tipo() {
21      return tipo;
22  }
23
24  int Nodo::get_numtipo() {
25      return numtipo;
26  }
```

1.2. ALGORITMO PRINCIPAL

Primero tenemos la función de emparejamiento para cada uno de los nodos función:

```
01  #include <iostream>
02  #include "nodo.h"
03  #include <cstdlib>
04  #include <stdio.h>
05  #include <vector>
06  using namespace std;
07
08  int match_eq(Nodo f, vector<Nodo> &C, vector<Nodo> funciones, vector<Nodo>
                variables, vector<int> &assignv,
09              vector<int> vmap, vector<vector<int> > fconexiones, vector<bool>
                &vpila) {
10      bool encontrado, fin;
11      int i,n;
12      vector<int> aux;
13
14      //AÃ±adimos el nodo ecuacion a la pila (lo coloreamos)
15      C.push_back(f);
16
17      encontrado = false;
18      fin = false;
19      aux = fconexiones[f.get_numtipo()];
20
21      cout<<"Inidice: "<<f.get_numtipo()<<" Aux: ";
22      for (i=0; i<aux.size(); i++) {
23          cout<<aux[i];
24      }
25      cout<<endl;
26
27      i = 0;
28      //Buscamos una variable conectada a f que no haya sido asignada y sin
        derivada
29      while (!fin) {
30          n = aux[i];
```

```
31          if ((assignv[n] == -1) && (vmap[n] == -1)) {
32              encontrado = true;
33          }
34          i +=1;
35          fin = (encontrado || (aux.size() == i));
36      }
37      cout<<"Encontrado: "<<encontrado<<endl;
38
39      //Dos casos de posible asignacion
40      if (encontrado) {
41          assignv[n] = f.get_numtipo();
42          return true;
43      } else {
44          //No existe ningun nodo variable que pueda ser asignado a f directamente,
            exploramos todas sus conexiones
45          for (i=0; i < aux.size(); i++) {
46              Nodo v = variables[aux[i]];
47              n = v.get_numtipo();
48              //Los nodo que no se encuentren an la coloracion y no tengan
                derivada, intentamos reajustarles para
49              //poder emparejarlos con nustro f actual
50              if ((!vpila[n]) && (vmap[n] == -1)) {
51                  cout<<"Probamos por la variable "<<n<<endl;
52                  C.push_back(v);
53                  vpila[n] = true;
54                  if (match_eq(funciones[assignv[n]], C, funciones, variables,
                        assignv, vmap, fconexiones, vpila)) {
55                      assignv[n] = f.get_numtipo();
56                      return true;
57                  }
58              }
59          }
60      }
61  //Si es imposible asignar la funcion de ninguna manera
62      return false;
63  }
```

Luego el algoritmo principal en el que creamos los nuevos nodos derivados y sus conexiones:

```
65  int main() {
66      //DATOS
67      int filas=5;
68      int col=9;
69      int m[5][9] =  {{0, 0, 1, 0, 1, 0, 0, 0, 0},
70          {0, 0, 0, 1, 0, 1, 0, 0, 0},
71          {1, 0, 0, 0, 0, 0, 1, 0, 1},
72          {0, 1, 0, 0, 0, 0, 0, 1, 1},
73          {1, 1, 0, 0, 0, 0, 0, 0, 0}
74      };
75      //Variables (orden): x, y, u, v, derx,dery, deru, derv, landa (*)
76      //Funciones: f1, f2 f3, f4, f5
77      //Numeradas las variables segun (*), indicamos cual es el indice de su
                                        derivada.
78      //Nota: empezamos a contar desde 0, y ponemos -1 si no tiene derivada
79      int mapini [9] = {4, 5, 6, 7, -1, -1, -1, -1, -1};
80
81      //VARIABLES A UTILIZAR
82      vector<Nodo> variables;//Almacena los nodos de variables creados
83      vector<Nodo> funciones; //Almacena los nodos de funciones creados
84      vector<vector<int> > fconexiones;//Almacena las variables conectadas a cada
```

```
    funcion
85  vector<bool> vpila; //Almacena si se encuentra un nodo variable en la pila.
86  vector<int> vmap; //Almacena mapini y posteriores cambios
87  vector<int> eqmap; //Similar a vmap, pero con las funciones. Al principio es
    todo -1
88  vector<int> assignv; //Indica que numero equacion se le asigna a cada nodo.
    Al principio es todo -1
89  bool match; //Marca si hemos encontrado la asignacion para el caso que se
    trata en ese momento
90  int i,j, ind, n, h;
91  Nodo f,v, no;
92  vector<Nodo> e;
93
94  //Construcion de las estructuras necesarias para el algoritmo
95  //Nodos variable, vmap, assignv y vcolor
96  for (i=0; i < col; i++) {
97      variables.push_back(Nodo('v', i));
98      vmap.push_back(mapini[i]);
99      assignv.push_back(-1);
100      vpila.push_back(false);
101  }
102
103  //Nodos funcion y eqmap
104  for (i=0; i < filas; i++) {
105      funciones.push_back(Nodo('f', i));
106      eqmap.push_back(-1);
107  }
108  //Conexiones de los nodo funcion del grafo
109  for (i=0; i < filas; i++) {
110      vector<int> aux2;
111      for (j=0; j<col; j++) {
112          if (m[i][j] == 1) {
113              aux2.push_back(j);
114          }
115      }
116      fconexiones.push_back(aux2);
117  }
118
119  //ALGORITMO
120  for (ind =0; ind < filas; ind++) {
121      //funciones.size()
122      f = funciones[ind];
123      match = false;
124      do {
125          vector<Nodo> C;
126          match = match_eq(f, C, funciones, variables, assignv, vmap,
                    fconexiones, vpila);
127          cout<<"Match: "<<match<<endl;
128          cout<<"Assignv: ";
129          for (h=0; h<assignv.size(); h++) {
130              cout<<" "<<assignv[h];
131          }
132          cout<<endl;
133          cout<<"Contenido de C: { ";
134          for (h=0; h<C.size(); h++) {
135              cout<<C[h].get_tipo()<<C[h].get_numtipo()<<" ";
136          }
137          cout<<"}"<<endl;
138
139          //Si no hemos encotrado un emparejamiento, tenemos que derivar (Crear
                                                                          nuevo
                                                                          nodo)
140          if (!match) {
```

```
141                 for (i=0; i < C.size(); i++) {
142                     Nodo no = C[i];
143                     //cout<<"En variables: tamaÃ±o C "<<
144                     if (no.get_tipo() =='v') {
145                         cout<< "C: tipo "<< no.get_tipo() <<" num "<<no.
                                                        get_numtipo() <<endl;
146                         //Creamos la nueva variable y actualizamos todos los
                        vectores
147                         Nodo noprima = Nodo('v',variables.size());
148                         vmap[no.get_numtipo()] = noprima.get_numtipo();
149                         vpila[no.get_numtipo()] = false;
150                         variables.push_back(noprima);
151                         assignv.push_back(-1);
152                         vmap.push_back(-1);
153                         vpila.push_back(false);
154                         cout<<"Creamos una nueva variable " << noprima.
                        get_numtipo() << " a partir de "<< no.get_numtipo() <<
                        endl;
155                         cout<<"Vmap: ";
156                         for (h=0; h<vmap.size(); h++) {
157                             cout<<" "<<vmap[h];
158                         }
159                         cout<<endl;
160                     }
161                 }
162                 for (i=0; i < C.size(); i++) {
163                     Nodo no = C[i];
164                     if (no.get_tipo() =='f') {
165                         cout<< "C: tipo "<< no.get_tipo() <<" num "<<no.
                                                        get_numtipo() <<endl;
166                         //Creamos la nueva ecuacion derivada y actualizamos
                        todos los vecores
167                         Nodo noprima = Nodo('f',funciones.size());
168 
169                         eqmap[no.get_numtipo()] = noprima.get_numtipo();
170                         eqmap.push_back(-1);
171                         funciones.push_back(noprima);
172                         cout<<"Creamos una nueva ecuacion "<< noprima.
                        get_numtipo() << " a partir de " <<no.get_numtipo()<<
                        endl;
173                         cout<<"Eqmap: ";
174                         for (h=0; h<eqmap.size(); h++) {
175                             cout<<" "<<eqmap[h];
176                         }
177                         cout<<endl;
178 
179                         //Miramos sus conexiones
180                         n = no.get_numtipo();
181                         vector<int> aux;
182                         for (j=0; j < fconexiones[n].size(); j++) {
183                             //AÃ±adimos la variable y su derivada
184                             aux.push_back(fconexiones[n][j]);
185                             aux.push_back(vmap[fconexiones[n][j]]);
186                         }
187                         fconexiones.push_back(aux);
188                         cout<<"Y sus conexiones son ";
189                         for (h=0; h<aux.size(); h++) {
190                             cout<<" "<<aux[h];
191                         }
192                         cout<<endl;
193                     }
194                 }
195 
```

```
196                 //Restructuramos las asignaciones de las variables a las que
                    hemos derivado
197                 for (i=0; i < C.size(); i++) {
198                     Nodo no = C[i];
199                     if (no.get_tipo() == 'v') {
200                         assignv[vmap[no.get_numtipo()]] = eqmap[assignv[no.
                                                    get_numtipo()]];
201                     }
202                 }
203 
204                 //Actualizamos la nueva funcion a considerar
205                 n = f.get_numtipo();
206                 f = funciones[eqmap[n]];
207                 cout<<"Nueva funcion: "<<f.get_numtipo()<<endl;
208             }
209             cout<<"Eqmap: ";
210             for (h=0; h<eqmap.size(); h++) {
211                 cout<<" "<<eqmap[h];
212             }
213             cout<<endl;
214             cout<<"Vmap: ";
215             for (h=0; h<vmap.size(); h++) {
216                 cout<<" "<<vmap[h];
217             }
218             cout<<endl;
219             cout<<"============================================================
                      ================"<<endl;
220         } while (!match);
221     }
222 
223     //Resultado final
224     cout<<"======================= EL RESULTADO FINAL ES
               ================================"<<endl;
225     cout<<"El numero de nodos variable creados: "<<variables.size()<<" y
              ecuacion" << funciones.size)<<endl;
226     cout<<"Eqmap: ";
227     for (h=0; h<eqmap.size(); h++) {
228         cout<<" "<<eqmap[h];
229     }
230     cout<<endl;
231     cout<<"Vmap: ";
232     for (h=0; h<vmap.size(); h++) {
233         cout<<" "<<vmap[h];
234     }
235     cout<<endl;
236     cout<<"Assignv: ";
237     for (h=0; h<assignv.size(); h++) {
238         cout<<" "<<assignv[h];
239     }
240     cout<<endl;
241 
242     return 0;
243 }
```

ANEXO C. TEARING

1. ALGORITMO DE CORTE

Al igual que el caso del algoritmo de Pantelides tenemos la clase Nodo con sus cabeceras y cuerpo, y luego el algoritmo como tal.

1.1. CLASE NODO.

El archivo de las cabeceras sería:

```
01 #ifndef NODO_H
02 #define NODO_H
03
04
05 class Nodo {
06   public:
07     Nodo();
08     Nodo(const Nodo& other);
09     Nodo(int f);
10     int get_fila() {
11         return fila;
12     }
13     void set_fila( int f) {
14         fila = f;
15     }
16     bool get_actual() {
17         return actual;
18     }
19     void set_actual(bool ac) {
20         actual = ac;
21     }
22
23   protected:
24   private:
25     int fila; //Numero de ecuacion dentro de la matriz
26     bool actual; //Indica si esta enn la componente fuerte que estamos tratando
       en el momento actual
27 };
28
29 #endif // NODO_H
```

El archivo con los constructores:

```
01 #include "Nodo.h"
02
```

```
03 Nodo::Nodo() {
04     fila = 0;
05     actual = false;
06 }
07
08 Nodo::Nodo(const Nodo& other) {
09     fila = other.fila;
10     actual = other.actual;
11 }
12
13 Nodo::Nodo(int f) {
14     fila = f;
15     actual = false;
16 }
```

1.2. FUNCIÓN PRINCIPAL

En este Apartado sólo mostraremos la parte correspondiente a la función principal y el cálculo del alfa, para ver los códigos correspondientes a Tarjan y al algoritmo de emparejamiento ir a los Anexos A y B.

```
120 //RAMIFICACION Y PODA
121
122 float calc_alfa(int indice, vector<vector <int> > prelist, vector<vector <int> >
                succlist, vector<Nodo> grafodep) {
123     float p=0,s=0, a;
124     int i,j;
125     vector<int> aux;
126
127     //Suma ponderada de los predecesores
128     aux = prelist[indice];
129     for (i=0; i<aux.size(); i++) {
130         if (grafodep[aux[i]].get_actual() ==true) {
131             p += 1;
132         }
133     }
134
135     //Suma ponderada de los sucesores
136     aux = succlist[indice];
137     for (i=0; i<aux.size(); i++) {
138         if (grafodep[aux[i]].get_actual() ==true) {
139             s += 1;
140         }
141     }
142     //Control para evitar la divisiÃ³n por 0.
143     if (s==0) {
144         a = 0;
145     } else {
146         a = p/s;
147     }
148     return a;
149 }
150
151 //=============================================================================
152
153 int main() {
154
```

```
155     //DATOS
156     int f=6;
157     int c=6;
158     int m[6][6] = {{0, 0, 1, 1, 0, 1},
159         {1, 1, 0, 0, 1, 1},
160         {0, 1, 1, 0, 0, 0},
161         {1, 0, 1, 0, 1, 0},
162         {0, 1, 0, 0, 0, 1},
163         {0, 0, 1, 0, 0, 1},
164     };
165     //Nota: Hay que cambiar los valores de las dimensiones de las funciones en
    las que se pasa m como valor
166
167     //Variables del programa
168
169     int i,j,h;
170     Nodo v;
171     vector<int> assignv; //Marca a que numero de ecuacion se ha asignado cada
    variable
172     vector<int> pila;//Pila utilizada en el algoritmo de Tarjan
173     vector<vector <int> > succlist; //Vector donde apuntaremos por cada ecuacion
    sus sucesores en el grafo de dependencia
174     vector<vector <int> > prelist; //Vector donde apuntaremos por cada ecuacion
    sus predecesores en el grafo de dependencia
175     vector<vector <int> > fconexiones; //Vector donde apuntaremos por cada
    ecuacion con quien esta conectado en el grafo del modelo
176     vector<vector <int> > cfuertes; //Lista donde apuntaremos las componentes
    fuertes
177     vector<int> lowlink, num;
178     vector <bool> enpila;
179     int ind = 0; //Inicializacion del indice de nodo en tarjan
180     vector<int> aux, aux2;
181     bool encontrado,fin;
182
183
184     //INICIALIZACIÃ“N
185     //Inicializamos assignv, lowlink, num
186     for (i=0; i<c; i++) {
187         lowlink.push_back(-1);
188         num.push_back(-1);
189         assignv.push_back(-1);
190         enpila.push_back(false);
191     }
192
193     //Inicializamos succlist, prelist;
194     for (i=0; i<c; i++) {
195         vector <int> w;
196         succlist.push_back(w);
197         prelist.push_back(w);
198     }
199
200
201     //Conexiones de los nodos funcion del grafo
202     for (i=0; i < f; i++) {
203         vector<int> aux2;
204         for (j=0; j < c; j++) {
205             if (m[i][j] == 1) {
206                 aux2.push_back(j);
207             }
208         }
209         fconexiones.push_back(aux2);
210     }
211
```

```
212     //ALGORITMO
213
214     //Matriz inicial
215     cout<<"La matriz inicial es:"<<endl;
216     for (i=0; i<f; i++) {
217         for (j=0; j<c; j++) {
218             cout << m[i][j];
219         }
220         printf("\n");
221     }
222
223
224     //1. Hallamos el emparejamiento inicial
225     emparejar(f,c,m,assignv, fconexiones);
226     cout << "El emparejamiento inicial es (f,v): ";
227     for (unsigned k=0; k<assignv.size(); k++) {
228         cout << "(" << (assignv[k]+1) << "," << (k+1) <<") ";
229     }
230     cout << '\n';
231
232     //2. Construimos los vectores succlist y prelist del grafo de dependencia
233     for (i=0; i<f; i++) {
234         for (j=0; j<c; j++) {
235             if ((m[i][j] == 1)&&(assignv[j]!= i)) {
236                 succlist[assignv[j]].push_back(i);
237                 prelist[i].push_back(assignv[j]);
238             }
239         }
240     }
241     for (unsigned i=0; i<succlist.size(); i++) {
242         cout << "Succlist de "<<i<<": ";
243         for (unsigned k=0; k<succlist[i].size(); k++) {
244             cout << " " << succlist[i][k];
245         }
246         cout << '\n';
247         cout << "Prelist de "<<i<<": ";
248         for (unsigned k=0; k<prelist[i].size(); k++) {
249             cout << " " <<prelist[i][k];
250         }
251         cout << '\n';
252     }
253
254
255     //3. Lanzamos el algoritmo de Tarjan
256     vector<Nodo> grafodep;
257     for (i=0; i<c; i++) {
258         v = Nodo(i);
259         grafodep.push_back(v);
260     }
261     for (i=0; i<c; i++) {
262         //Para aquellos nodos a los que no hemos metido en la pila todavia
263         if (num[i]==-1) {
264             f_connect(i,succlist, pila, ind, cfuertes, lowlink, num, enpila);
265         }
266     }
267
268     //3.5. Sacamos las componentes fuertes por pantalla
269     imprimir_fuertes(cfuertes);
270
271     do {
272         //4. Buscamos una componente fuerte con dim mayor a 1.
273         encontrado = false;
274         fin = false;
```

```
275         i = 0;
276         vector<int> curr;
277         while (!fin) {
278             if (cfuertes[i].size()>1) {
279                 encontrado = true;
280                 curr = cfuertes[i];;
281             }
282             i +=1;
283             fin = (encontrado || (i ==cfuertes.size()));
284         }
285 
286         //5. Marcamos las ecuaciones de la actual componente fuerte
287         for (i=0; i<curr.size(); i++) {
288             grafodep[i].set_actual(true);
289         }
290 
291         //6. Ramificacion y poda
292         if (encontrado) {
293             vector<float> alfa;
294             for (i=0; i<curr.size(); i++) {
295                 alfa.push_back(calc_alfa(curr[i], prelist, succlist, grafodep));
296             }
297 
298             int l;
299             //Buscamos el minimo
300             int alfmin = alfa[0];
301             int k = curr[0];
302             for (i=1; i<curr.size(); i++) {
303                 if (alfa[i]< alfmin) {
304                     alfmin = alfa[i];
305                     k = curr[i];
306                 }
307             }
308             cout<<"Podamos por k = "<<k<<endl;
309             //Cortamos todas las conexiones que llegan a k
310             for (i=0; i<prelist[k].size(); i++) {
311                 h = prelist[k][i];
312                 for (j=0; succlist[h].size(); j++) {
313                     if (succlist[h][j]== k) {
314                         cout << "Succlist de "<<h<<" y quitamos "<<k<< endl;
315                         succlist[h].erase(succlist[h].begin()+j);
316                         break;
317                     }
318                 }
319             }
320         }
321         //Lanzamos el algoritmo de Tarjan
322         cout<<"Buscamos las nuevas componentes fuertes"<<endl;
323         for (i=0; i<c; i++) {
324             lowlink[i] = -1;
325             num[i] = -1;
326             enpila[i] = false;
327         }
328         cfuertes.clear();
329         ind = 0;
330         for (i=0; i<c; i++) {
331             //Para aquellos nodos a los que no hemos metido en la pila todavia
332             if (num[i]==-1) {
333                 f_connect(i,succlist, pila, ind, cfuertes, lowlink, num, enpila);
334             }
335         }
336 
337         //Sacamos las componentes fuertes por pantalla
```

```
338         imprimir_fuertes(cfuertes);
339      } while (f>cfuertes.size());
340      cout<<"El nÃºmero de ecuciones "<<f<<" es igual al de componentes fuertes "<<
       cfuertes.size()<<endl;
341
342      return 0;
343 }
```

BIBLIOGRAFÍA

A. Urquía y C. Martín 2011, *Modelado orientado a objetos y simulación de sistemas físicos*, Departamento de Informática y Automática.

Anónimo, *Ditutor*, <http://www.ditutor.com/ecuaciones_grado2/metodo_gauss.html>.

Broman, D 2013, 'Lecture 12b in EECS 144/244', *Universidad de California*.

Casella, F, *Modelica Minicourse*, <http://staff.polito.it//roberto.zanino/sub1/teach_files/modelica_minicourse/03%20-%20Symbolyc%2Manipulation.pdf>.

Computación, DDLYCDL, *Universidad de Málaga*, <http://www.lcc.uma.es/~av/Libro/CAP7.pdf>.

F.E. Cellier y E. Kofman 2006, *Continuous System Simulation*, Springer.

H. Elmqvist y M. Otter June 1994, 'Methods for tearing systems of equations in object-oriented modeling', European Simulation Multiconference, Institute for Robotics and System Dynamics, Weßling, German.

O'Connor, JLF 1997, *Técnicas de cálculo para sistemas de ecuaciones, programación lineal y programación entera*, 3rd edn.

Palacios, F 2008, *Escuela Politécnica Superior de Ingeniería de Manresa*, <http://www.epsem.upc.edu/~fpq/ale-hp/modulos/aplicaciones/newton.pdf>.

Pantelides, CC 1988, 'The consistent initialization of differencial-algebraic systems', *SIAM Journal on Scientific and Statistical Computing*, vol 9, no. 2, pp. 213-231.

Relajación, *Métodos de Relajación*, <http://users.dsic.upv.es/asignaturas/eui/cnu/libro/tema6/tema65.htm>.

Romero, S, *Universidad de Huelva*, <http://www.uhu.es/sixto.romero/ficheros/Temas/MII-Tema1.pdf>.

T. Gundersen y T. Hertzberg 1983, 'Partitioning and tearing of networks - applied to process flowsheeting, Modeling, Identification and Control', vol 4, no. 3, pp. 139-165.

Tarjan, R 1972, 'Depth-first search and linear graph algorithms', *SIAM Journal of Computation*, vol 1, no. 2, pp. 146-160.

Team Code:Blocks, *Code:Blocks*, <http://www.codeblocks.org/home>.

Volpi, L 2004, *Note of Numeric Calculus*, Foxes Team.