Trabajo Fin de Grado

Método Paralelo de Resolución de Ecuaciones de Ligadura para Moléculas Lineales

Parallel Constraint Solver for Linear Molecules

Autor

Rubén Langarita Benítez

Directores

Jesús Alastruey Benedé

Pablo Ibáñez Marín

ESCUELA DE INGENIERÍA Y ARQUITECTURA

2018

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe acompañar al Trabajo Fin de Grado (TFG)/Trabajo Fin de Máster (TFM) cuando sea depositado para su evaluación).

D./Dª. Rubén Langarita Benítez,

con nº de DNI 73105873M en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de (Grado/Máster) Ingeniería Informática, (Título del Trabajo) Método Paralelo de Resolución de Ecuaciones de Ligadura para Moléculas Lineales,

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 29 de Junio de 2018

Fdo: Ruben

Método Paralelo de Resolución de Ecuaciones de Ligadura para Moléculas Lineales

RESUMEN

La dinámica molecular es una técnica de simulación por computador que estudia la evolución en el tiempo de un sistema de partículas. Las herramientas que realizan estas simulaciones son esenciales en campos como la biomedicina o la industria química. Por ejemplo, la dinámica molecular se ha aplicado con éxito al diseño de nuevos fármacos o al análisis de materiales. En este campo, imponer ligaduras, es decir, fijar las longitudes de los enlaces atómicos, es una práctica habitual que permite aumentar el paso temporal de simulación (*time step*). De esta forma se pueden acelerar los experimentos o simular mayores intervalos temporales. Los algoritmos más usados para imponer ligaduras convergen linealmente y están basados en aproximaciones que afectan a su estabilidad numérica. Además, no se conoce una implementación paralela eficiente de los mismos.

En este trabajo se presenta la implementación paralela de un nuevo algoritmo para imponer ligaduras, ILVES, que converge de forma cuadrática. Para acotar la complejidad del proyecto, se ha abordado la resolución de las ecuaciones de ligadura de una molécula compuesta por una cadena lineal de átomos. Utilizamos el método del complemento de Schur para dividir el sistema de ecuaciones en varios subsistemas susceptibles de ser resueltos de forma vectorial y paralela. Se han implementado tres versiones: base (que es escalar y secuencial), vectorial y paralela, que resuelven las ecuaciones de ligadura hasta el límite de precisión máquina de forma eficiente. Para una tolerancia de 10^{-12}, la aceleración de las versiones vectorial y paralela con respecto a SHAKE es de 4.2 y 6.1 respectivamente.

Índice

Capítulo 1

Introducción

1.1. Motivación

La dinámica molecular es una técnica que permite simular con un computador el comportamiento a lo largo del tiempo de un sistema de átomos [1] [2] [3]. Esta herramienta se usa para realizar experimentos sin necesidad de disponer de las sustancias reales. Las simulaciones permiten aumentar la accesibilidad y reducir los costes de los experimentos. Por ejemplo, la dinámica molecular se ha aplicado con éxito al diseño de nuevos fármacos o al análisis de materiales.

Restringir los grados de libertad de los enlaces atómicos permite aumentar el paso temporal de simulación, de forma que pueden acelerarse los experimentos o simularse mayores intervalos temporales. A este proceso se le llama imposición de ligaduras.

Los métodos más usados para resolver los sistemas de ecuaciones resultantes tras la imposición de ligaduras son SHAKE [4], RATTLE [5] y LINCS [6]. Estos métodos convergen linealmente y están basados en aproximaciones que afectan a su estabilidad numérica. A pesar de múltiples esfuerzos, no se conoce una implementación paralela eficiente de los mismos.

1.2. Objetivos y Tareas

El objetivo de este trabajo es implementar una versión paralela de un algoritmo llamado ILVES [7] para resolver las ecuaciones de ligadura. Se va a abordar el caso de un sistema molecular compuesto por una cadena lineal de átomos. Hay algunas investigaciones que han usado moléculas lineales y circulares para aplicar algoritmos de imposición de ligaduras [8]. Además, esta aproximación servirá como base para implementar el algoritmo sobre una molécula real.

Se ha utilizado el método del complemento de Schur para dividir el sistema de ecuaciones en varios subsistemas que pueden resolverse de forma vectorial y paralela. Se

han implementado tres versiones: base, vectorial y paralela, que resuelven las ecuaciones de ligadura hasta el límite de precisión máquina de forma eficiente. La aceleración de las versiones vectorial y paralela con respecto a SHAKE es de hasta 4.2 y 6.1 respectivamente, para una tolerancia de 10^{-12}.

Previo al trabajo, se ha realizado una tarea de documentación acerca del tema de dinámica molecular, y más concretamente de imposición de ligaduras. A continuación, se ha estudiado código de imposición de ligaduras escrito en una aplicación de dinámica molecular (GROMACS), además de una versión simplificada de ILVES. Una vez entendidos los conceptos básicos y los objetivos, se ha implementado ILVES de forma incremental, y se ha conseguido que la versión final mejore los tiempos de ejecución con respecto a SHAKE. Por último, se han realizado pruebas y se ha escrito la presente memoria. Se puede ver una distribución aproximada del tiempo dedicado al proyecto en la Figura 1.1.

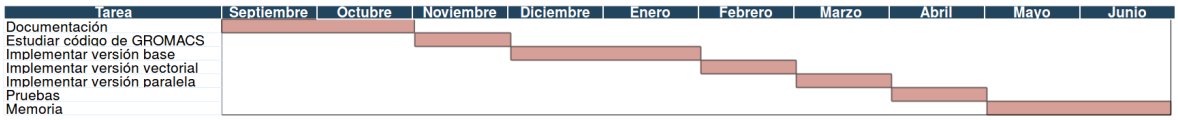

Figura 1.1: Tiempo dedicado al proyecto por tareas.

1.3. Descripción del Contenido

Esta memoria contiene cinco capítulos incluyendo la presente introducción. En el capítulo 2 se hace una breve introducción a la dinámica molecular y a la imposición de ligaduras. Los algoritmos para imponer ligaduras son una parte del proceso de dinámica molecular y son el objeto de estudio de este trabajo. Se presenta ILVES, un algoritmo de imposición de ligaduras que resulta en un sistema de ecuaciones que se desea resolver de forma paralela.

En el capítulo 3 se describe el algoritmo del complemento de Schur, el cual se usará para resolver el sistema de ecuaciones establecido por ILVES. También se explica su implementación y las distintas versiones desarrolladas.

En el capítulo 4 se presenta la metodología utilizada, se describen los experimentos realizados y se analizan los resultados obtenidos por las diversas implementaciones de ILVES.

Por último, en el capítulo 5 se extraen las conclusiones del trabajo y se mencionan las futuras tareas a realizar.

A modo de información adicional, se ha añadido un anexo que contiene una breve introducción a la vectorización. También se ha añadido un anexo con el código y

otro con un artículo que se publicará en las Jornadas de Paralelismo (JP2018) que se celebrarán en Teruel.

Capítulo 2

Algoritmos para la Imposición de Ligaduras

En este capítulo se hace una introducción al campo de la dinámica molecular. En la primera sección se describen los pasos en una simulación de dinámica molecular y se explican los conceptos básicos de los algoritmos para imponer ligaduras. A continuación, se describen dos de los algoritmos de imposición de ligaduras más usados: SHAKE y LINCS, que están implementados por ejemplo en la conocida aplicación de dinámica molecular GROMACS. Por último, se presenta otro algoritmo para imponer ligaduras llamado ILVES, que es el objeto de este trabajo.

2.1. Fundamentos

Para llevar a cabo las simulaciones de dinámica molecular se calculan las posiciones de los átomos en cada paso temporal de la simulación. Para saber dónde estarán los átomos al final de un paso temporal, primero se calculan las fuerzas que se ejercen sobre los átomos, y después se mueven los átomos de acuerdo a esas fuerzas. La ecuación que relaciona la fuerza ejercida sobre un átomo α con la aceleración que se produce sobre él es la siguiente:

$$\vec{F}_\alpha = m_\alpha \cdot \vec{a}_\alpha, \tag{2.1}$$

siendo m_α la masa de α y $\vec{a}_\alpha$ la aceleración de α.

Entre las fuerzas simuladas se encuentran, por ejemplo, las producidas entre los átomos debido a sus cargas eléctricas. Para este trabajo son de interés las fuerzas ejercidas entre los átomos que forman un enlace. Por ejemplo, en una molécula de agua (H_2O), las fuerzas que se ejercen entre el átomo de oxígeno y los dos átomos de hidrógeno, tal y como se ve en la Figura 2.1.

La distancia entre dos átomos que forman un enlace no es fija, ya que los átomos

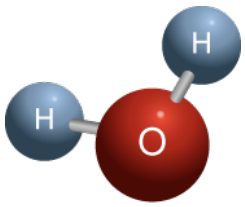

Figura 2.1: Enlaces atómicos en una molécula de agua (H_2O).

vibran ligeramente. El período de estas vibraciones es del orden de femtosegundos (fs = 10^{-15} s) y determina el paso temporal de la simulación. Si el paso temporal es demasiado grande respecto al periodo de vibración, la simulación podría ser incorrecta. El paso temporal suele establecerse en un quinto del mínimo periodo de vibración.

La duración del paso temporal puede aumentarse si se imponen ligaduras, es decir, si se restringen los grados de libertad internos asociados a las vibraciones de alta frecuencia de los enlaces atómicos. Imponer una ligadura a un enlace entre dos átomos consiste en fijar una distancia constante entre dichos átomos igual a la longitud del enlace químico. La ligadura de un enlace k entre dos átomos a y b se define como:

$$\sigma_{k(a,b)} = |\mathbf{r}_a - \mathbf{r}_b|^2 - (d_{a,b})^2 = 0, \tag{2.2}$$

donde $\mathbf{r}_\mathrm{a}$ y $\mathbf{r}_\mathrm{b}$ son los vectores posición de los átomos a y b. Esta ligadura expresa que la distancia entre estos átomos debe ser igual a $d_{a,b}$.

La imposición de ligaduras añade nuevos términos al sistema de ecuaciones clásico de Newton para un sistema molecular. Así, para un átomo α:

$$\vec{F}_\alpha + \vec{F}_{ligaduras} = \vec{F}_\alpha - \sum_{k=1}^{N} \lambda_k \frac{\partial \sigma_k}{\partial r_\alpha} = m_\alpha \cdot \vec{a}_\alpha, \tag{2.3}$$

donde $\vec{F}_\alpha$ es la fuerza ejercida sobre el átomo α debida a las interacciones entre átomos, a la que llamaremos fuerza externa, $-\sum_{k=1}^{N} \lambda_k \frac{\partial \sigma_k}{\partial r_\alpha}$ es la fuerza sobre el átomo α que aparece debida a las ligaduras, y los distintos λ_k son los multiplicadores de Lagrange. Cada uno de los multiplicadores de Lagrange está asociado a una ligadura y determina la fuerza que se aplica a los átomos de esa ligadura. Para que las distancias entre átomos sean las determinadas por las longitudes de sus enlaces ($d_{a,b}$), se realiza un procedimiento llamado imposición de ligaduras, que consiste en la aplicación de las fuerzas artificiales (fuerzas de ligadura) que se calculan a partir de los multiplicadores de Lagrange [9].

Las herramientas de dinámica molecular resuelven este sistema en dos fases. Poniendo como ejemplo una molécula de agua, la situación inicial sería la de la Figura 2.2.

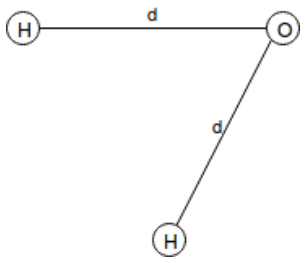

Figura 2.2: Posición de los átomos de una molécula de agua al comienzo de un paso temporal de simulación. d es la longitud del enlace entre un átomo de hidrógeno y uno de oxígeno.

En una primera fase, los átomos se mueven de acuerdo a las fuerzas externas sin tener en cuenta las fuerzas de ligadura. Al acabar esta fase, la distancia entre los átomos de un enlace puede ser distinta a la longitud de dicho enlace. Por ejemplo, para una molécula de agua el resultado podría ser el de la Figura 2.3.

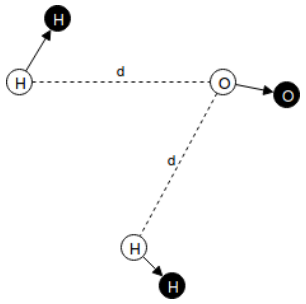

Figura 2.3: Posición de los átomos de una molécula de agua después de aplicar las fuerzas externas.

En una segunda fase se corrigen las posiciones de los átomos para que se cumplan las ligaduras, es decir, para que las distancias entre los átomos unidos por un enlace sea igual a la longitud de dicho enlace. Después de realizar este proceso, las posiciones de los átomos de la molécula de agua podrían ser las mostradas en la Figura 2.4.

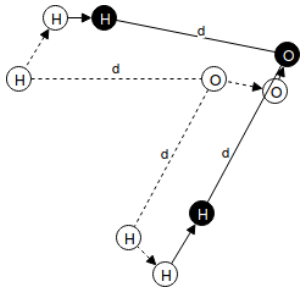

Figura 2.4: Posición de los átomos de una molécula de agua después de corregir sus posiciones para que se cumplan las ligaduras.

Las vibraciones descritas anteriormente ya no se simularían, de hecho no son necesarias para el objetivo de la simulación, ya que no aportan información relevante.

El proceso de imposición de ligaduras es un problema complejo, ya que la corrección de las posiciones de dos átomos para hacer cumplir una ligadura afecta al resto de ligaduras asociadas a esos átomos.

En la ecuación 2.4 se pueden observar las condiciones necesarias para que tres átomos (a, b y c) cumplan las restricciones impuestas por dos ligaduras: a-b y b-c.

$$\begin{aligned} \sigma_{k(a,b)} &= |\mathbf{r}_a - \mathbf{r}_b|^2 - (d_{a,b})^2 = 0 \\ \sigma_{k(b,c)} &= |\mathbf{r}_b - \mathbf{r}_c|^2 - (d_{b,c})^2 = 0 \end{aligned} \tag{2.4}$$

siendo $\sigma_{k(i,j)}$ la ligadura que actúa sobre los átomos i y j, $\mathbf{r}_i$ la posición del átomo i, y $d_{i,j}$ la distancia impuesta por la ligadura (longitud de enlace).

2.2. SHAKE

SHAKE fue el primer algoritmo implementado para la imposición de ligaduras [4]. Este método resulta en un sistema de n ecuaciones con n incógnitas, siendo n el número de ligaduras. Al resolver el sistema, se obtienen los multiplicadores de Lagrange asociados a cada una de las ligaduras. Los átomos se mueven de acuerdo a estos multiplicadores de Lagrange. Este proceso realiza tantas iteraciones como sean necesarias hasta conseguir minimizar el error hasta el punto establecido al principio de la simulación. Estas iteraciones no se deben confundir con los pasos temporales. En cada paso temporal se simula un periodo de tiempo, mientras que entre iteraciones el valor del tiempo no varía, solo se intenta reducir el error dentro de un mismo paso temporal.

SHAKE resuelve el sistema de ecuaciones con otro proceso iterativo: primero se calcula el multiplicador de Lagrange para la primera ligadura y se corrigen las posiciones de los átomos de esa ligadura, después hace lo mismo con la segunda ligadura, la tercera y así sucesivamente.

Cuando se aplican estas correcciones, los átomos se mueven en la dirección del enlace en el paso temporal anterior. Esto se puede apreciar en la Figura 2.5, la dirección de la corrección $\vec{c}$ es la misma que la dirección del enlace en el paso temporal anterior $\vec{d}$.

2.3. LINCS

LINCS (Linear Constraint Solver) es otro algoritmo muy usado para imponer ligaduras [6]. Al igual que SHAKE, consta de dos fases, y cuando se corrigen las posiciones, los átomos se mueven en la dirección del enlace en el anterior paso temporal. En la primera fase, cada átomo se mueve hasta la perpendicular con el enlace en el

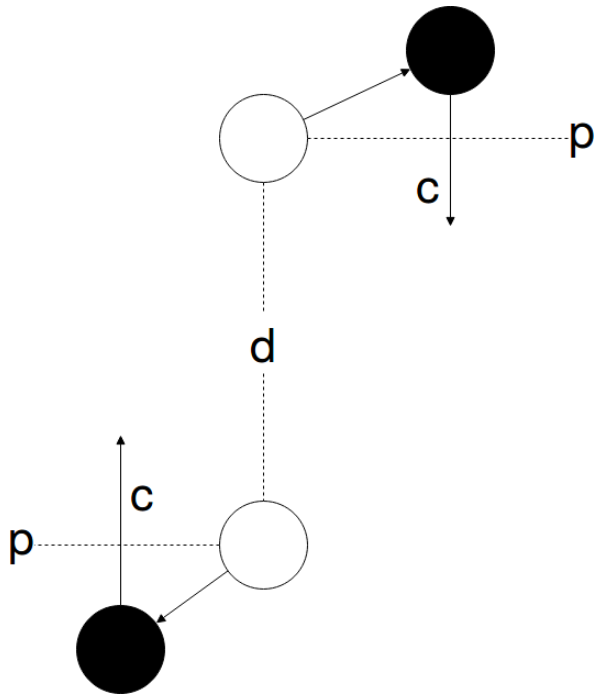

Figura 2.5: Dirección de la corrección en SHAKE y en LINCS. Los círculos blancos indican la posición de los átomos al final del paso temporal anterior y los negros en el actual pero sin haberse aplicado la corrección. La línea d indica el enlace en el paso temporal anterior. Las flechas c indican la dirección en la que tendrá lugar la corrección, que es paralela a d. En LINCS p es la línea sobre la que se parará el átomo después del primer paso.

anterior paso temporal en el punto donde estaba el átomo, en la Figura 2.5 la línea p. En la segunda fase se realiza otra corrección siguiendo la misma dirección que en la primera, hasta que la distancia entre los átomos sea igual a la longitud del enlace.

En ambas fases se multiplican una serie de matrices. Además, en cierto punto del algoritmo es necesario invertir una matriz. Para evitar esta costosa operación, se usa la siguiente aproximación:

$$(I - B)^{-1} \simeq I + B^1 + B^2 + B^3 + B^4 + B^5 + ... \tag{2.5}$$

Normalmente se calcula hasta la cuarta u octava potencia. Este es el cálculo del algoritmo que más tiempo consume.

2.4. ILVES

Este trabajo de fin de grado se enmarca dentro de una línea de investigación cuyo objetivo final es integrar ILVES en GROMACS, una de las aplicaciones más usadas de dinámica molecular.

ILVES es un algoritmo diseñado por Pablo García Risueño. En la implementación colabora el matemático Carl Christian Kjelgaard Mikkelsen, matemático especialista en la resolución de grandes sistemas de ecuaciones.

Existen dos versiones de ILVES: ILVES-S basada en SHAKE e ILVES-L basada en LINCS. ILVES-S, la versión que se ha implementado en este trabajo, resuelve el mismo sistema de ecuaciones que SHAKE, pero de forma directa. SHAKE corrige las posiciones de los átomos cada vez que calcula un multiplicador de Lagrange, en cambio, ILVES-S resuelve el sistema y aplica las correcciones de todos los multiplicadores a la vez. Tanto ILVES-S como SHAKE tienen que resolver el sistema varias veces para conseguir minimizar el error hasta el punto deseado.

Una implementación escalar y secuencial de ILVES fue desarrollada por M.A. Serrano [10]. En dicho trabajo se evaluaba la precisión y convergencia de ILVES, así como una primera estimación de sus prestaciones. El siguiente gran objetivo era resolver el sistema de ecuaciones resultante de forma paralela. Para conseguirlo, y dada su dificultad, se definieron varios hitos incrementales. En primer lugar se paralelizó la resolución de ILVES para muchas moléculas pequeñas e idénticas, que es el caso del disolvente en el que se lleva a cabo la simulación [11]. En este trabajo se presenta la ejecución del siguiente hito: vectorizar y paralelizar ILVES para una cadena lineal de átomos sin ramificaciones. En las siguientes tareas se pretende paralelizar el algoritmo para resolver el sistema de ecuaciones resultante al imponer ligaduras a una cadena de átomos con ramificaciones y a una molécula arbitraria. El último objetivo es integrar en GROMACS el algoritmo desarrollado.

Capítulo 3

Implementación Eficiente de ILVES-S

Abordamos el caso de una molécula compuesta por una cadena lineal de átomos (ver Figura 3.1). Este caso especial servirá como base para la implementación de versiones posteriores.

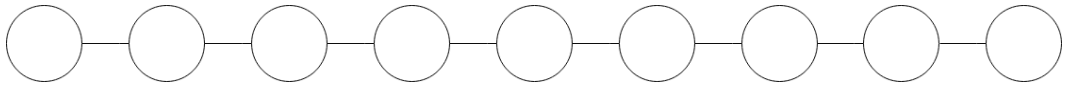

Figura 3.1: Representación de una cadena lineal de átomos.

ILVES-S, al igual que SHAKE, resulta en un sistema lineal de n ecuaciones con n incógnitas, siendo n el número de ligaduras. Las incógnitas se corresponden con los multiplicadores de Lagrange, tal y como se ha dicho en el capítulo anterior. Al igual que SHAKE, ILVES-S necesita resolver el sistema varias veces hasta alcanzar la tolerancia deseada. Al tratarse de una molécula lineal, si se tienen m átomos, el número de ligaduras n será igual a $m - 1$.

En el sistema de ecuaciones a resolver aparece un elemento no nulo en la posición i, j si los enlaces i y j poseen algún átomo en común. Por lo tanto, el sistema lineal equivalente ($Ax = b$) será tridiagonal, es decir, los únicos elementos no nulos de la matriz A están en la diagonal principal y sus adyacentes, tal y como se ve en la matriz de la Figura 3.2.

Para resolver un sistema de ecuaciones se puede, por ejemplo, utilizar el método de eliminación gaussiana o se puede invertir la matriz y multiplicar por el vector de términos independientes. Estos métodos son muy costosos computacionalmente. Se ha optado por el método del complemento de Schur, ya que es propicio para ser paralelizado.

3.1. Método del Complemento de Schur

El método del complemento de Schur divide la matriz en submatrices más pequeñas, de manera que entre las submatrices quede una fila y una columna que no pertenezcan a ninguna submatriz, tal y como se puede ver en la matriz de la Figura 3.2.

$$
\left(\begin{array}{cccc|c|cccc|c|cccc|c|cccc}
a & a & & & & & & & & & & & & & & & & & \\
a & a & a & & & & & & & & & & & & & & & & \\
 & a & a & a & & & & & & & & & & & & & & & \\
 & & a & a & a & & & & & & & & & & & & & & \\
\hline
 & & & a & a & a & & & & & & & & & & & & & \\
\hline
 & & & & a & a & a & & & & & & & & & & & & \\
\hline
 & & & & & a & a & a & & & & & & & & & & & \\
 & & & & & & a & a & a & & & & & & & & & & \\
 & & & & & & & a & a & a & & & & & & & & & \\
 & & & & & & & & a & a & a & & & & & & & & \\
\hline
 & & & & & & & & & a & a & & & & & & & & \\
\hline
 & & & & & & & & & & a & a & a & & & & & & \\
 & & & & & & & & & & a & a & a & a & & & & & \\
 & & & & & & & & & & & a & a & a & & & & & \\
 & & & & & & & & & & & & a & a & a & & & & \\
\hline
 & & & & & & & & & & & & & a & a & a & & & \\
\hline
 & & & & & & & & & & & & & a & a & a & a & & \\
 & & & & & & & & & & & & & & a & a & a & & \\
 & & & & & & & & & & & & & & & a & a & a & \\
 & & & & & & & & & & & & & & & & a & a &
\end{array}\right)
\left(\begin{array}{c}
x \\ x \\ x \\ x \\ \hline x \\ \hline x \\ x \\ x \\ x \\ \hline x \\ \hline x \\ x \\ x \\ x \\ \hline x \\ \hline x \\ x \\ x \\ x \\ x
\end{array}\right)
=
\left(\begin{array}{c}
g \\ g \\ g \\ g \\ \hline g \\ \hline g \\ g \\ g \\ g \\ \hline g \\ \hline g \\ g \\ g \\ g \\ \hline g \\ \hline g \\ g \\ g \\ g \\ g
\end{array}\right)
$$

Figura 3.2: Aspecto del sistema de ecuaciones inicial. Los ceros han sido representados con espacios en blanco y los elementos no nulos con a. La matriz se ha dividido en submatrices más pequeñas de dimensión 4×4. Entre las submatrices queda una fila y una columna que no pertenece a ninguna submatriz. En la figura se repite la misma letra para designar distintos elementos aunque tengan diferente valor, se hace por simplificar y se aplica también a las demás figuras.

Primero se resuelven los sistemas correspondientes a las submatrices. Cada una de las submatrices se resuelve mediante factorización LU. Sin embargo, por sencillez se va a usar eliminación gaussiana en la descripción del funcionamiento del método de Schur. A continuación, se explica el procedimiento para una submatriz.

El objetivo es que cada submatriz termine siendo la matriz identidad. Se empieza eliminando la diagonal inferior. En la Figura 3.3 se parte del estado inicial de una submatriz y se muestra cómo eliminar el elemento de la diagonal inferior de la fila 2.

A la fila 2 se le resta la fila 1 multiplicada por un factor c para eliminar el elemento de la diagonal inferior. Aparece un elemento e en la columna izquierda debido a la a de la misma columna de la fila superior. El elemento de la diagonal principal (b) queda

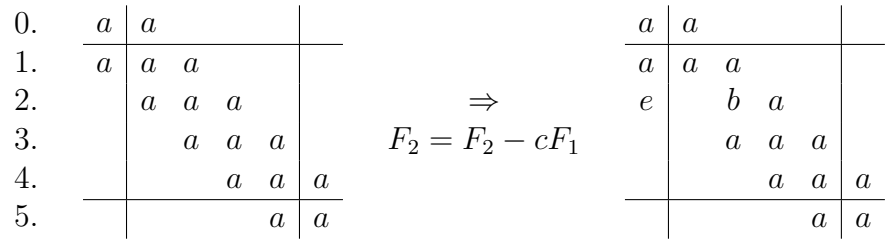

Figura 3.3: Eliminación de los elementos de la diagonal inferior.

modificado. Este proceso se repite con las filas restantes, hasta eliminar los elementos de la diagonal inferior.

A continuación hay que eliminar los elementos de la diagonal superior. En la Figura 3.4 se parte del estado en el que todos los elementos de la diagonal inferior han sido eliminados. Obsérvese que han ido apareciendo elementos a lo largo de la columna izquierda durante la eliminación de la diagonal inferior debido a la influencia de las filas superiores. Por el mismo motivo, al eliminar el elemento de la diagonal inferior de la fila 5 también ha aparecido un valor en la esquina inferior izquierda y se ha modificado el valor de la esquina inferior derecha de dicha fila. Estas S se denominan términos del complemento de Schur, del que se hablará más adelante. En la parte derecha de la Figura 3.4 se muestra cómo se elimina el elemento de la diagonal superior de la fila 3.

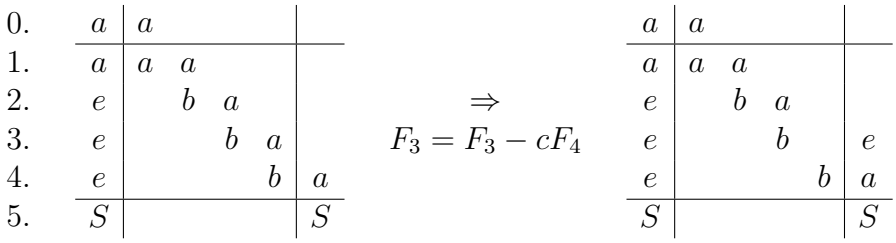

Figura 3.4: Eliminación de los elementos de la diagonal superior.

A la fila 3 se le ha restado la fila 4 multiplicada por un factor c de manera que el elemento de la diagonal superior queda eliminado. Aparece un elemento e en la columna derecha al igual que ha sucedido al eliminar los elementos de la diagonal inferior en la columna izquierda. El elemento e de la columna izquierda de la fila 3 también queda modificado. Esto se repite con las filas restantes hasta eliminar todos los elementos de la diagonal superior.

En la Figura 3.5 se parte del estado en el que se han eliminado las diagonales inferior y superior. Ha aparecido un elemento en la esquina superior derecha y se ha

modificado el valor de la esquina superior izquierda, estos valor también pertenecen al complemento de Schur. Obsérvese que los elementos de la esquina superior izquierda y de la esquina inferior derecha también sufrirán modificaciones debido a las submatrices contiguas. En la Figura 3.5 se muestra cómo se modifica la fila 1 para que el elemento de la diagonal sea igual a 1.

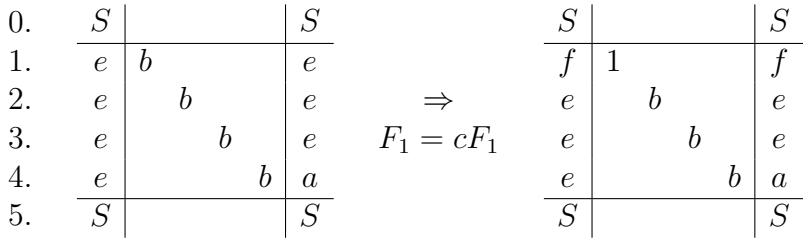

Figura 3.5: Escalado de los elementos de la diagonal principal.

La fila 1 se multiplica por un factor c de manera que el elemento de la diagonal pase a ser 1. Los elementos e de las columnas laterales quedan modificados. Esta acción se repite para todas las filas de la submatriz y el resultado se muestra en la Figura 3.6.

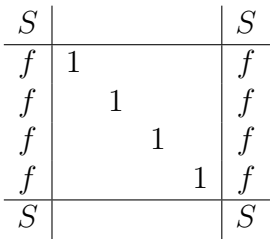

Figura 3.6: Estado de la submatriz y de las filas y columnas contiguas después de resolver el sistema de ecuaciones asociado a la submatriz.

Los elementos de la diagonal pasan a ser 1, y la submatriz es la matriz identidad. En las columnas laterales han quedado elementos f, a los que se les denomina *fill-ins*.

Una vez aplicado este procedimiento a todas las submatrices, el resultado es el de la Figura 3.7.

A continuación se forma un sistema tridiagonal con las ecuaciones de las filas entre las submatrices (en la Figura 3.7, aquellas filas que contienen S), que se llama complemento de Schur, tal y como se ve en la Figura 3.8. Este sistema se resuelve también mediante factorización LU. Si el complemento de Schur tuviera muchas ecuaciones podría resolverse a su vez mediante el método del complemento de Schur, ya que tiene las mismas características que el sistema inicial. En nuestro caso no merece la

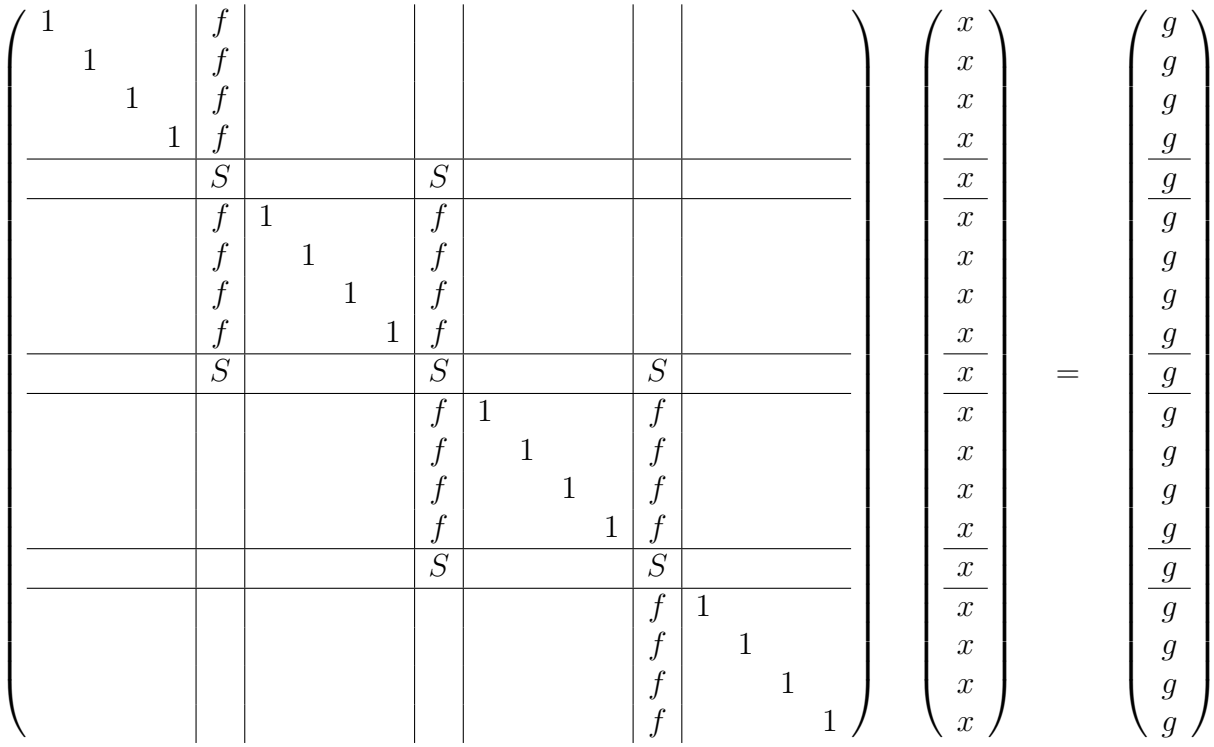

Figura 3.7: Representación del sistema de ecuaciones después de transformar las submatrices. Aparecen los *fill-ins* y los valores pertenecientes al complemento de Schur (f y S respectivamente).

pena hacer esto debido al pequeño tamaño del complemento de Schur. Tras la resolución del complemento de Schur queda el sistema de ecuaciones mostrado en la Figura 3.9.

$$\begin{pmatrix} S & S & \\ S & S & S \\ & S & S \end{pmatrix} \begin{pmatrix} x \\ x \\ x \end{pmatrix} = \begin{pmatrix} g \\ g \\ g \end{pmatrix}$$

Figura 3.8: Complemento de Schur.

Por último, se eliminan los *fill-ins* de cada submatriz mediante eliminación gaussiana usando las filas del complemento de Schur que acaba de ser resuelto. Cada ecuación resuelta del complemento de Schur servirá para eliminar los *fill-ins* que tenga encima y debajo.

Se parte de la matriz que aparece en la Figura 3.9. En la Figura 3.10 se observa como eliminar un *fill-in*.

El *fill-in* de la cuarta fila de la columna derecha ha sido eliminado. Este procedimiento se repetirá para los *fill-ins* restantes de ambas columnas. Con todos los *fill-ins* eliminados, la matriz se convierte en la matriz identidad, y por lo tanto los términos independientes son la solución del sistema de ecuaciones asociado.

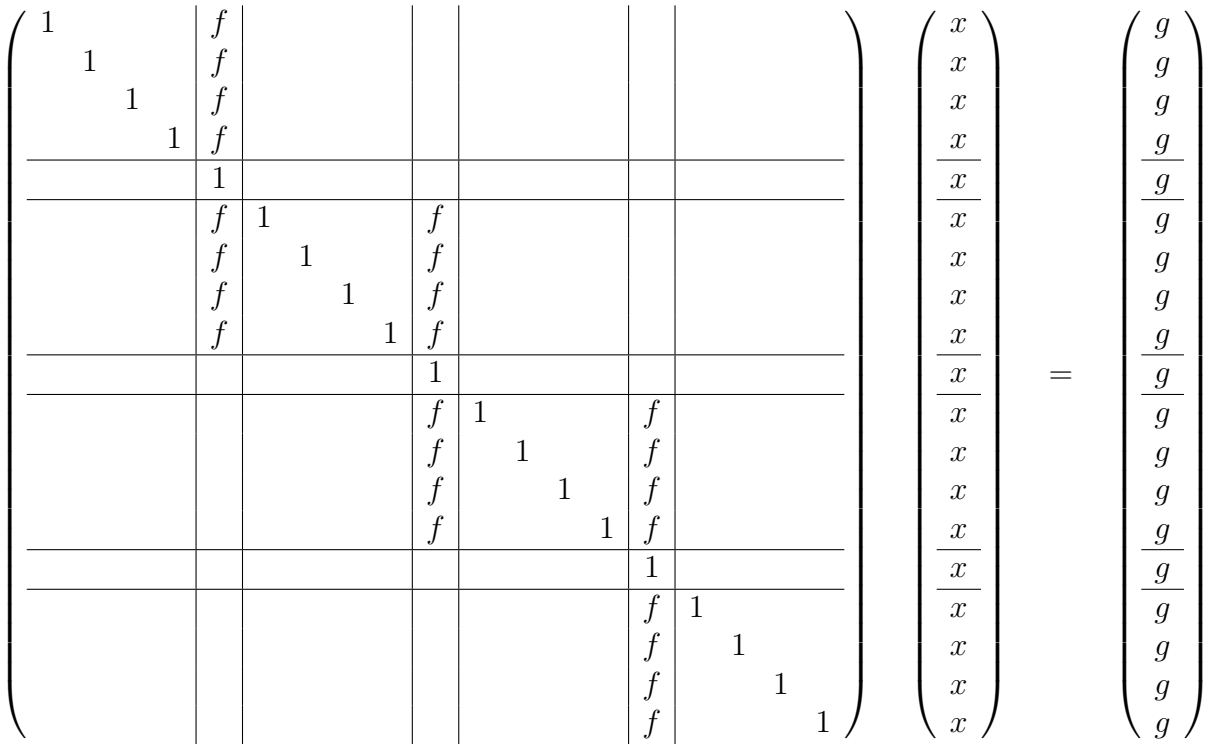

Figura 3.9: Representación del sistema de ecuaciones después de resolver el complemento de Schur.

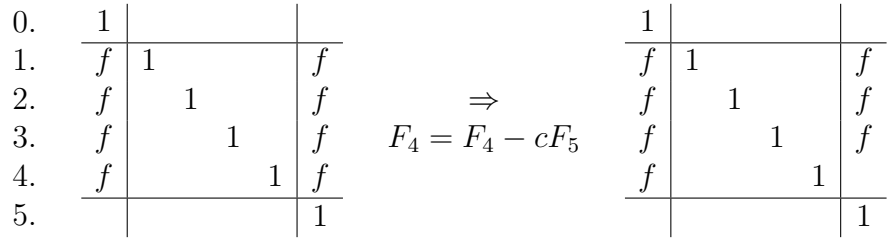

Figura 3.10: Eliminación de *fill-in*.

La ventaja de este algoritmo es que el trabajo puede ser paralelizado y vectorizado. Cada hilo se puede encargar de una submatriz distinta, a excepción de la resolución del complemento de Schur.

3.2. Implementación

El lenguaje de programación ha sido *C*, debido a que se ha partido del código escrito en la fase anterior del proyecto [11], y se ha usado *gcc* como compilador.

La implementación del algoritmo no se ha realizado de forma directa, sino que se ha llevado a cabo una aproximación sistemática e incremental. Primero se ha desarrollado una versión a modo de prueba, que utiliza eliminación gaussiana para resolver los

sistemas de las submatrices y una representación densa de la matriz. Esta versión es muy lenta y costosa en memoria, pero es la más fácil de implementar. Se ha usado como referencia para la verificación de resultados. Esta versión no se evaluará en la comparativa de tiempos.

A continuación, se ha implementado la versión base utilizando la factorización LU para resolver los sistemas de las submatrices y una representación dispersa de la matriz. Esta versión es escalar y secuencial, es decir, no usa las extensiones vectoriales y se ejecuta en un único núcleo. Una vez verificada su corrección, esta versión ha servido como referencia para compararla con las demás, tanto en tiempo como en precisión.

Por último, se han desarrollado las versiones vectorial y paralela, que son el objetivo final del trabajo. La versión vectorial hace uso de las extensiones vectoriales del procesador y se ejecuta en un solo hilo, mientras que la versión paralela, además de usar las extensiones vectoriales, hace uso de varios hilos. Se puede encontrar un breve resumen de las extensiones vectoriales en el Anexo A.

Previo a la resolución del sistema ha sido necesario generar la matriz y el vector de términos independientes. Después de resolver el sistema, la solución se utilizará para corregir las posiciones de los átomos. Como ya se ha dicho, este proceso se repetirá hasta alcanzar la tolerancia deseada.

3.2.1. Versión Base

En esta sección se analiza la versión base, implementada usando una representación dispersa de la matriz y utilizando la factorización LU para resolver las submatrices. Se han creado tres funciones, una por cada paso del método de Schur:

1. Resolver los sistemas de ecuaciones de las submatrices
2. Resolver el complemento de Schur
3. Eliminar los *fill-ins*

En el primer paso, durante la resolución de las submatrices, los resultados necesarios para los pasos posteriores son el vector de términos independientes, los *fill-ins* y los complementos de Schur. Las submatrices van a terminar siendo matrices identidad, por lo que no hace falta calcular su valor. Las ecuaciones 3.1-3.5 definen la manera de calcularlos. Estas ecuaciones hacen referencia a los bloques definidos en los sistemas de ecuaciones de la Figura 3.11, en concreto a los dos primeros.

$$M_i = A_i^{-1} B_i^U \tag{3.1}$$

$$\left(\begin{array}{c|c|c|c|c|c|c} A_1 & B_1^U & & & & & \\ \hline B_1^L & C_1 & D_2^U & & & & \\ \hline & D_2^L & A_2 & B_2^U & & & \\ \hline & & B_2^L & C_2 & D_3^U & & \\ \hline & & & D_3^L & A_3 & B_3^U & \\ \hline & & & & B_3^L & C_3 & D_4^U \\ \hline & & & & & D_4^L & A_4 \end{array}\right) \quad \left(\begin{array}{c} x_1 \\ \hline y_1 \\ \hline x_2 \\ \hline y_2 \\ \hline x_3 \\ \hline y_3 \\ \hline x_4 \end{array}\right) \quad = \quad \left(\begin{array}{c} g_1 \\ \hline h_1 \\ \hline g_2 \\ \hline h_2 \\ \hline g_3 \\ \hline h_3 \\ \hline g_4 \end{array}\right)$$

⇓ Paso 1: resolver los sistemas de ecuaciones de las submatrices

$$\left(\begin{array}{c|c|c|c|c|c|c} I & M_1 & & & & & \\ \hline & S_{11} & & S_{12} & & & \\ \hline & N_2 & I & M_2 & & & \\ \hline & S_{21} & & S_{22} & & S_{23} & \\ \hline & & & N_3 & I & M_3 & \\ \hline & & & S_{32} & & S_{33} & \\ \hline & & & & & N_4 & I \end{array}\right) \quad \left(\begin{array}{c} x_1 \\ \hline y_1 \\ \hline x_2 \\ \hline y_2 \\ \hline x_3 \\ \hline y_3 \\ \hline x_4 \end{array}\right) \quad = \quad \left(\begin{array}{c} i_1 \\ \hline j_1 \\ \hline i_2 \\ \hline j_2 \\ \hline i_3 \\ \hline j_3 \\ \hline i_4 \end{array}\right)$$

⇓ Paso 2a: construir complemento de Schur

$$\left(\begin{array}{ccc} S_{11} & S_{12} & \\ S_{21} & S_{22} & S_{23} \\ & S_{32} & S_{33} \end{array}\right) \quad \left(\begin{array}{c} y_1 \\ y_2 \\ y_3 \end{array}\right) \quad = \quad \left(\begin{array}{c} j_1 \\ j_2 \\ j_3 \end{array}\right)$$

⇓ Paso 2b: resolver complemento de Schur

$$\left(\begin{array}{c|c|c|c|c|c|c} I & M_1 & & & & & \\ \hline & 1 & & & & & \\ \hline & N_2 & I & M_2 & & & \\ \hline & & & 1 & & & \\ \hline & & & N_3 & I & M_3 & \\ \hline & & & & & 1 & \\ \hline & & & & & N_4 & I \end{array}\right) \quad \left(\begin{array}{c} x_1 \\ \hline y_1 \\ \hline x_2 \\ \hline y_2 \\ \hline x_3 \\ \hline y_3 \\ \hline x_4 \end{array}\right) \quad = \quad \left(\begin{array}{c} i_1 \\ \hline k_1 \\ \hline i_2 \\ \hline k_2 \\ \hline i_3 \\ \hline k_3 \\ \hline i_4 \end{array}\right)$$

Figura 3.11: Recopilación de los sistemas de ecuaciones mostrados en las figuras anteriores, pero dando nombre a los componentes (submatrices, *fill-ins*, términos de Schur, incógnitas y términos independientes). El primer sistema se corresponde con el estado inicial (Figura 3.2), los bloques A son de tamaño 4×4, los bloques B^U y D^L son de tamaño 4×1, los bloques B^L y D^U son de tamaño 1×4 y los bloques de C de tamaño 1×1. El segundo sistema corresponde con el estado después de haber resuelto los sistemas de ecuaciones asociados a las submatrices (Figura 3.7), los bloques I representan la matriz identidad 4×4, los bloques M y N son los *fill-ins* y los bloques S son los términos del complemento de Schur. El tercer sistema representa el complemento de Schur extraído del segundo sistema. El último sistema representa el estado después de haber resuelto el complemento de Schur (Figura 3.9). El paso tres del algoritmo (la eliminación de *fill-ins*) no ha sido representado, ya que el resultado es la matriz identidad.

$$N_i = A_i^{-1} D_i^L \tag{3.2}$$

$$i_i = A_i^{-1} g_i \tag{3.3}$$

$$Sij = \begin{cases} i = j & C_i - M_i B_i^L - N_{i+1} D_{i+1}^U \\ i > j & -N_i B_i^L \\ i < j & -M_i D_j^U \end{cases} \tag{3.4}$$

$$j_1 = h_1 - i_i B_i^L - i_{i+1} D_{i+1}^U \tag{3.5}$$

Para calcular todos los componentes es necesario calcular las inversas de las submatrices. Como ya se ha dicho, esto es muy costoso computacionalmente, por lo que se optará por realizar una factorización LU.

La factorización LU consiste en descomponer la matriz a resolver en dos más simples: la matriz L, que no contiene elementos por encima de la diagonal, y la matriz U, que no contiene elementos por debajo de la diagonal. Con la matriz factorizada, el sistema de ecuaciones $Ax = g$ se convierte en $LUx = g$, que se resolverá en dos pasos: $forward$ y *backward*, que resolverán los sistemas $Ly = g$ y $Ux = y$ respectivamente. Esta técnica se utilizará cada vez que haya que invertir una matriz. Por ejemplo, calcular $M_i = A_i^{-1} B_i^U$ es lo mismo que resolver el sistema de ecuaciones $A_i M_i = B_i^U$, y factorizando la matriz A habría que resolver el siguiente sistema: $L_i U_i M_i = B_i^U$.

Una vez calculados todos los componentes, en el segundo paso del algoritmo es necesario tratar el complemento de Schur como si fuese un sistema independiente. Hay que extraer de la matriz original el complemento de Schur a una nueva matriz, el resultado sería el tercer sistema de ecuaciones de la Figura 3.11. El sistema se resolverá mediante factorización LU, ya que no merece la pena paralelizar debido al pequeño tamaño de la matriz. En este caso solo es necesario factorizar el complemento de Schur y aplicar $forward$ y *backward* sobre los términos independientes correspondientes. Una vez resuelto el sistema de complementos de Schur, las filas volverán a insertarse en el sistema original, el resultado será el cuarto sistema de ecuaciones de la Figura 3.11.

Por último es necesario eliminar los *fill-ins*. Al hacer desparecer los *fill-ins* los términos independientes se modifican. A cada término independiente hay que aplicarle la influencia de las filas con las que se han eliminado los *fill-ins*. Los términos independientes son los únicos datos relevantes, ya que después de eliminar los *fill-ins*, la matriz A se convertirá en la matriz identidad y por lo tanto los términos independientes serán la solución del sistema.

3.2.2. Versión Vectorial

Para que un código sea vectorizable, se deben cumplir dos condiciones. La primera, es que las operaciones a realizar se repitan de forma regular sin dependencias entre ellas. En el caso del método del complemento de Schur, el algoritmo que se aplica a todas las submatrices es el mismo. Por ejemplo, a la hora de factorizar las submatrices, con una longitud vectorial de 4 se podrían factorizar 4 submatrices simultáneamente.

La segunda condición es que los datos sobre los que se realizan las operaciones estén contiguos en memoria. Los procesadores actuales soportan operaciones vectoriales para cargar y guardar datos no contiguos en memoria, pero son más lentas que si se reorganizan los datos. En el caso de factorizar las submatrices, se necesita que los elementos i-ésimos de las submatrices que se van a factorizar estén contiguos en memoria. Para este propósito se han creado unas estructuras que contienen la misma información que en la versión base, pero con los datos reordenados.

Para la versión base se necesitan dos estructuras: una para la matriz y otra para los términos independientes. Estas estructuras contienen los elementos de forma secuencial tal y como los distribuiría un programador que no estuviese pensando en vectorizar. En la versión vectorial han sido necesarias tres estructuras para la matriz y dos para los términos independientes. Estas estructuras contienen los mismos elementos que en la versión base pero cambiando el orden para que la vectorización sea posible. Entre paréntesis se indica el nombre de las estructuras en el programa. La matriz se ha divido en submatrices ($valVec$), *fill-ins* ($fillLeft$ y $fillRight$) y el complemento de Schur ($schurMatrix$). Los términos independientes se han dividido en los que corresponden a las submatrices ($gVec$) y en los que corresponden al complemento de Schur ($gSchurVec$).

$valVec$, $fillLeft$, $fillRight$ y $gVec$ han sido reorganizados de manera que todos los elementos i-ésimos estén contiguos en memoria. Por ejemplo, si la longitud vectorial fuese de 4 elementos, la estructura de las submatrices contendría lo siguiente: 1$^{\text{er}}$ elemento de la 1$^{\text{a}}$ submatriz, 1$^{\text{er}}$ elemento de la 2$^{\text{a}}$ submatriz, 1$^{\text{er}}$ elemento de la 3$^{\text{a}}$ submatriz, 1$^{\text{er}}$ elemento de la 4$^{\text{a}}$ submatriz, 2$^{\text{o}}$ elemento de la 1$^{\text{a}}$ submatriz, 2$^{\text{o}}$ elemento de la 2$^{\text{a}}$ submatriz, 2$^{\text{o}}$ elemento de la 3$^{\text{a}}$ submatriz, 2$^{\text{o}}$ elemento de la 4$^{\text{a}}$ submatriz, 3$^{\text{er}}$ elemento de la 1$^{\text{a}}$ submatriz, y así sucesivamente. Se puede ver una reorganización de los términos independientes en la Figura 3.12

No hace falta reorganizar las estructuras correspondientes al complemento de Schur ($schurMatrix$ y $gSchurVec$), ya que las operaciones que se realizan con ellas (factorización, *forward* y *backward* en el paso 2) no van a ser vectorizadas.

Como ya se ha dicho, antes de la resolución del sistema se deben generar la matriz y

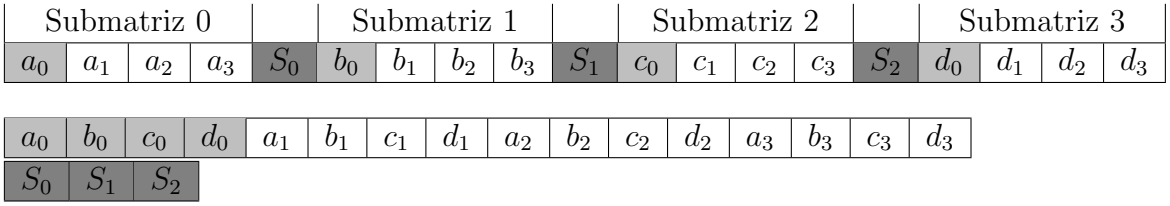

Figura 3.12: Ejemplo de reorganización de los términos independientes con 19 elementos y con una longitud vectorial de 4 elementos. Arriba se observa la estructura utilizada para la versión escalar. Abajo se observan las dos estructuras usadas en la versión vectorial: $gVec$ y $gSchurVec$ respectivamente.

los términos independientes, y posteriormente, los resultados se usan para corregir las posiciones de los átomos. En una primera aproximación para vectorizar el algoritmo, se han reorganizado los datos justo antes y después de la resolución del sistema, pasando de las estructuras de la versión base a las vectoriales y viceversa en funciones específicas. Sin embargo, en versiones posteriores se han modificado la función que construye la matriz, la que construye el vector de términos independientes y la que corrige las posición de los átomos para que escriban o lean directamente de las estructuras vectoriales.

Esto se ha implementado en dos fases. En la primera se ha usado un solo bucle para todas las estructuras, de manera que una instrucción condicional indica en que estructura se escribe. Posteriormente se usó un bucle para cada estructura, de forma que se puede separar el código que rellena la parte compartida por las submatrices ($schurMatrix$ y $gSchurMatrix$) del que rellena la parte privada de cada submatriz ($valVec$, $fillLeft$, $fillRight$ y $gVec$).

Una vez realizado todo este proceso, se observó que para la versión vectorial era más eficiente construir la matriz y el vector de términos independientes sobre las estructuras de la versión base y posteriormente reorganizarlos en las estructuras vectoriales que escribir directamente sobre las estructuras vectoriales.

Para vectorizar el código hace falta que el bucle más interno itere sobre las submatrices, de manera que el compilador detecte que esos elementos están contiguos en memoria y pueda vectorizar. En la Figura 3.13 se muestran las versiones base y vectorial de la función de factorización. En el código correspondiente a la versión vectorial se puede apreciar que el bucle más interno itera sobre las submatrices. A cada elemento del vector $valVec$ se le suma j, ya que. como se ha explicado antes, los elementos i-ésimos de las submatrices están contiguos en memoria. También se ha añadido la directiva *#pragma GCC ivdep* para indicar al compilador que no existen dependencias entre iteraciones. El compilador es capaz de vectorizar este bucle automáticamente, detecta que en el bucle interno los operandos y los resultados están contiguos en memoria.

Además, se ha definido la variable $VLEN$ como la longitud vectorial del procesador, por lo que el compilador es capaz de detectar que el bucle interno se puede sustituir por una operación vectorial.

```
for(int i = start; i < start + block_size - 1; i++) {
    double factor = val[xadj[i+1]-1]/val[xadj[i+1]-2];
    val[xadj[i+1]-1] = factor;
    val[xadj[i+1]+1] -= val[xadj[i+1]]*factor;
}
```

```
for(int i = 0; i < block_size - 1; i++) {
    #pragma GCC ivdep
    for (int j = 0; j < VLEN; j++) {
        double factor = valVec[xadjVec[i]+2*VLEN+j]/valVec[xadjVec[i]+VLEN+j];
        valVec[xadjVec[i]+2*VLEN+j] = factor;
        valVec[xadjVec[i+1]+VLEN+j] -= valVec[xadjVec[i+1]+j]*factor;
    }
}
```

Figura 3.13: Código para factorizar una submatriz. Arriba la versión base, abajo la vectorial.

3.2.3. Versión Paralela

Por último, se ha implementado la versión paralela. La idea es dividir la matriz en submatrices de manera que estas puedan ser procesadas en paralelo. Para la versión vectorial, la matriz se dividía en $VLEN$ submatrices. Para la versión paralela, la matriz se dividirá en el número de hilos que se vayan a utilizar multiplicado por $VLEN$. De esta manera cada hilo tendrá que procesar $VLEN$ submatrices.

Por ejemplo, para una cadena de 2048 átomos, el número de ligaduras (y por tanto la dimensión de la matriz) será de 2047. Con una longitud vectorial de 4 y usando 8 hilos, habría que dividir la matriz en 32 submatrices. El tamaño del complemento de Schur sería de 31 filas y el tamaño de las submatrices de 63 filas. Las 32 submatrices se repartirían entre los 8 hilos, a cada uno de los cuales le tocarían 4 submatrices. A su vez, cada uno de los hilos vectorizará los cálculos de las 4 submatrices que le toquen.

Para la versión paralela, cada hilo tendrá sus propias estructuras $valVec$, $fillLeft$, $fillRight$ y $gVec$. Las estructuras $schurMatrix$ y $gSchurMatrix$ serán compartidas, debido a que el complemento de Schur es compartido por todos. Esto permite paralelizar la generación del sistema y la corrección de posiciones, cada hilo se encargará de generar su parte del sistema (submatriz y términos independientes) y corregir las posiciones que le toquen. El hilo principal es el encargado de resolver el complemento de Schur,

ya que esta sección de código no está paralelizada.

Se ha usado OpenMP para paralelizar el algoritmo. OpenMP es un API diseñada para la programación de aplicaciones multiproceso con memoria compartida. OpenMP permite paralelizar el algoritmo mediante directivas, de manera que si se compila sin usar OpenMP se generará una versión secuencial. Estas directivas permiten especifiar qué secciones se ejecutarán de forma paralela, definiendo, entre otras cosas, el tipo de planificación y el número de hilos. La opción de compilación de *gcc* que activa el soporte OpenMP es *-fopenmp*.

Durante la resolución de los sistemas formados por las submatrices, dos submatrices contiguas comparten el término de Schur que se queda entre ellas. Para que no se produzcan inconsistencias en los datos, la sección de código que modifica los términos de Schur de la diagonal se protege con un mutex para cada término.

La división de trabajo se ha hecho de forma estática. Se ha implementado un bucle que itera tantas veces como hilos se deseen usar. Encima del bucle se ha insertado una directiva OpenMP para indicar que cada iteración del bucle tiene que ser ejecutada por un hilo distinto. El resultado sería el de la Figura 3.14.

```
#pragma omp parallel for schedule(static,1) default(shared)
for(int l=0; l < NTHREADS; l++)
```

Figura 3.14: Directiva de OpenMP insertada para paralelizar un bucle.

Capítulo 4

Resultados

En esta sección se describe la metodología utilizada y se presentan los resultados de las pruebas realizadas.

4.1. Metodología

El lenguaje de programación ha sido *C*. Se ha usado el compilador *gcc* versión 7.1, junto con *Make* para facilitar la compilación y *git* para realizar el control de versiones.

Los experimentos se han realizado en un sistema con procesador Intel Core i5-4570 (microarquitectura Haswell) a 3.20 GHz con 4 núcleos y 12 GB de memoria. Este procesador soporta la extensión vectorial AVX2, cuya longitud vectorial es de 256 bits.

Para evaluar la precisión hemos utilizado el error relativo. El error relativo de una ligadura entre dos átomos a y b se define como:

$$e^{(a,b)} := \frac{|\vec{r}_a - \vec{r}_b|^2 - (d_{a,b})^2}{(d_{a,b})^2} \tag{4.1}$$

donde $|\vec{r}_a - \vec{r}_b|^2$ es el cuadrado de la distancia entre los dos átomos, y $(d_{a,b})^2$ es el cuadrado de la longitud del enlace atómico. El error de un paso temporal es el máximo de los errores relativos de todas las ligaduras. La tolerancia se define como el umbral máximo de error permitido en cada paso de la simulación.

Se realizarán tantas iteraciones como sean necesarias hasta conseguir reducir el error por debajo de la tolerancia indicada en la configuración del experimento. El número de iteraciones se refiere al número de veces que se ha resuelto el sistema de ecuaciones para corregir las posiciones de los átomos.

Las pruebas se han realizado con moléculas de 2048 átomos y simulando 100 pasos temporales. En cada paso temporal, cada átomo se mueve de forma aleatoria para simular las fuerzas externas (interacciones entre átomos) y se aplica el algoritmo de imposición de ligaduras para corregir las posiciones.

Los experimentos se han realizado con números reales de doble precisión (64 bits), por lo que la longitud vectorial es de 4 elementos. La versión paralela se ha ejecutado en los 4 núcleos del procesador.

Se ha implementado SHAKE para comparar su precisión y velocidad con ILVES-S.

4.2. Resultados

Para mostrar los resultados obtenidos con las implementaciones de ILVES-S se han realizado dos experimentos. En el primero se ha medido el error relativo en función del número de iteraciones. En el segundo se mide el tiempo de ejecución de las distintas versiones de ILVES-S en función de la tolerancia.

La figura 4.1 muestra el error relativo (eje Y en escala logarítmica) en función del número de iteraciones (eje X) para los algoritmos SHAKE e ILVES-S. ILVES-S necesita unas pocas iteraciones para conseguir un error mínimo que viene impuesto por la precisión de la máquina, mientras que SHAKE requiere más de 100 iteraciones. Este resultado ya se puso de manifiesto en el trabajo de M.A. Serrano [10]. La convergencia de ILVES-S es cuadrática mientras que la de SHAKE es lineal. Pese a esto, en el trabajo citado se observó que el tiempo de ejecución de aquella primera versión de ILVES-S era similar al tiempo de SHAKE o incluso mayor en algunos casos. El resultado era debido a un tiempo de ejecución mucho mayor para cada iteración de ILVES-S respecto a las de SHAKE.

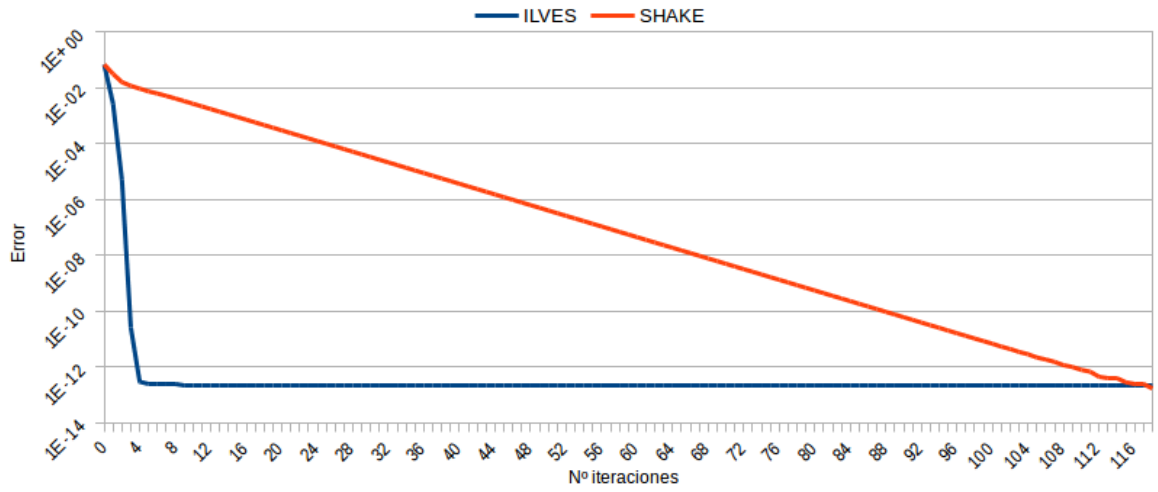

Figura 4.1: Error en función del número de iteraciones.

En el segundo experimento se ha medido el tiempo de ejecución que requieren las versiones implementadas de ILVES-S. La Figura 4.2 muestra los tiempos de ejecución de SHAKE y de la versión base de ILVES-S (eje Y) para distintos valores de tolerancia (eje X).

Al reducir la tolerancia se requiere un número mayor de iteraciones para conseguirla y, por lo tanto, un mayor tiempo de ejecución. En el tiempo de ILVES-S se puede observar una forma en escalera. Esto es debido a que una reducción de la tolerancia no implica necesariamente un aumento de iteraciones. Los puntos de la gráfica en los que se produce un aumento del tiempo marcan que el algoritmo necesita una iteración más para reducir el error. El tiempo de ejecución de SHAKE es siempre mayor que el de ILVES-S, y la diferencia aumenta cuanto más se desea reducir el valor de la tolerancia. Con una tolerancia de 10^{-12}, la versión base de ILVES-S es 3.46 veces más rápida que SHAKE. Sin embargo, ya se ha comentado que la ventaja de ILVES-S respecto a SHAKE es la posibilidad de ser vectorizado y paralelizado.

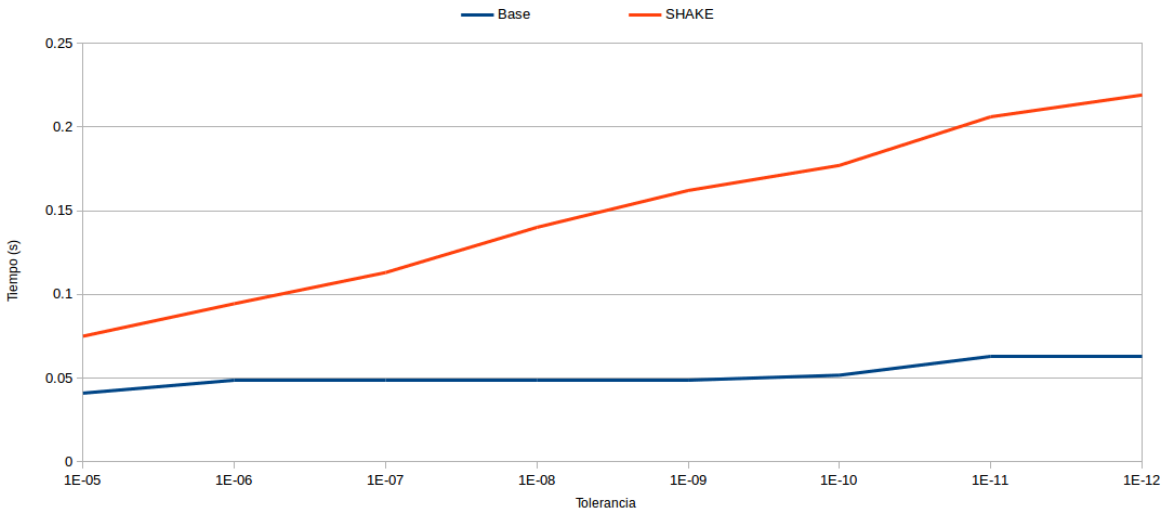

Figura 4.2: Tiempo de ejecución en función de la tolerancia especificada para SHAKE e ILVES-S.

En la Figura 4.3 se muestran los tiempos de ejecución de las tres versiones de ILVES-S (base, vectorial y paralela). Las diferencias entre las versiones secuenciales (base y vectorial) y la paralela aumentan al disminuir la tolerancia debido a la sobrecarga en la creación de hilos y en el reparto de trabajo. El paralelismo del problema está limitado al cálculo en cada paso temporal, ya que cada paso temporal depende del resultado del anterior. Al simular una cadena lineal sin ramificaciones, las ligaduras generan un sistema poco complejo para la dimensión de la molécula. Las moléculas reales resultarán en sistemas de ecuaciones más complejos que requerirán más cálculo.

El *speedup* conseguido al vectorizar varía entre 1.2 y 1.31 mientras que la paralelización consigue *speedups* entre 1.28 y 1.44 respecto a la versión vectorizada.

La versión final es entre un 66 % y un 79 % más rápida que la versión base y entre un 209 % y un 510 % más rápida que SHAKE.

Pese a los buenos resultados obtenidos somos conscientes de que el potencial que

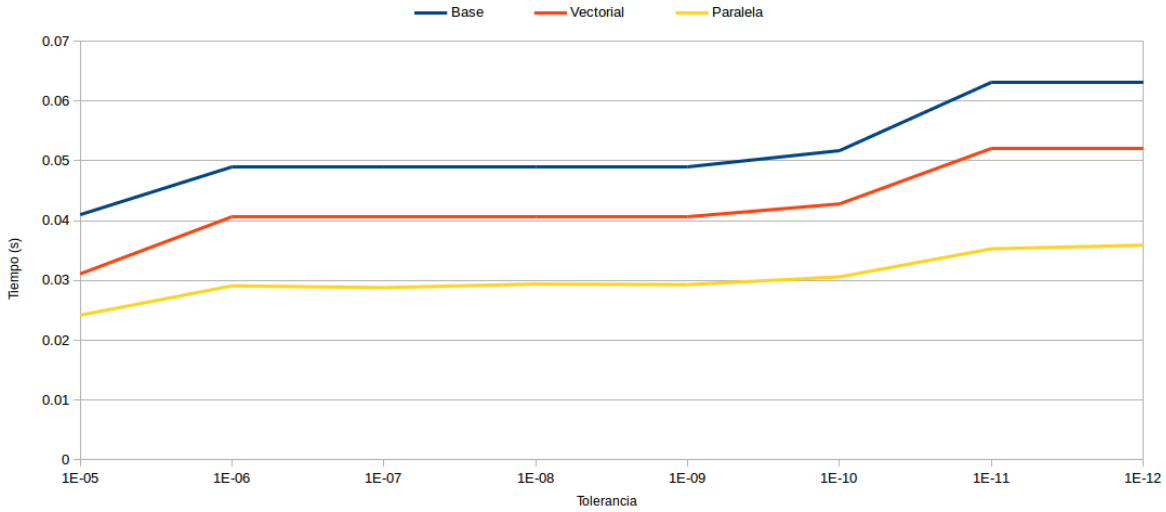

Figura 4.3: Tiempo de ejecución respecto a la tolerancia especificada para las versiones base, vectorial y paralela de ILVES-S.

ofrecen la vectorización y la paralelización es mucho mayor que el conseguido. Por ello, como línea de trabajo futuro se podría caracterizar la ejecución vectorial y paralela para detectar los cuellos de botella que están afectando al rendimiento.

Capítulo 5

Conclusiones

En este trabajo de final de grado se ha conseguido vectorizar y paralelizar el algoritmo de imposición de ligaduras ILVES, para una molécula lineal. Para ello, primero se ha hecho un trabajo de documentación acerca de la dinámica molecular y la imposición de ligaduras. Se ha estudiado el código de GROMACS y una versión preliminar de ILVES.

Para implementar ILVES de forma paralela sobre una molécula lineal, ha sido necesario resolver un sistema de ecuaciones tridiagonal mediante el método del complemento de Schur. Este método es vectorizable y paralelizable, y se puede aplicar fácilmente a un sistema de ecuaciones tridiagonal.

Se han implementado varias versiones de ILVES. Primero se ha implementado una versión a modo de prueba que usaba una representación densa de la matriz. A continuación, se ha implementado una versión base que se ha servido de referencia para compararla con las demás versiones. Después se ha implementado una versión vectorial. Para ello ha necesario definir nuevas estructuras reorganizando los datos para que el compilador sea capaz de vectorizar. Se ha implementado la versión paralela que hace uso de varios hilos del procesador para resolver el sistema, y a su vez cada hilo hace uso de las extensiones vectoriales. Por último, se ha implementado SHAKE para comparar sus prestaciones con las de ILVES-S.

Se han evaluado SHAKE y las versiones base, vectorial y paralela de ILVES-S. Las iteraciones de ILVES son más costosas que las de SHAKE, pero a cambio se necesitan muchas menos iteraciones para alcanzar la solución. La versión paralela, la más rápida de todas, es hasta 6.1 veces más rápida que SHAKE.

La próxima fase del proyecto consistirá en generalizar esta solución para una cadena de átomos con ramificaciones. El proyecto finalmente acabará por integrarse en GROMACS, de manera que se necesitará menos tiempo para ejecutar los algoritmos de dinámica molecular en esta aplicación.

Se ha elaborado un artículo con el resumen del TFG que se va a presentar en las

Jornadas de Paralelismo 2018 en Teruel, se adjunta el artículo en un anexo.

En cuanto al conocimiento adquirido, cabe destacar la introducción en nuevos campos como el de la dinámica molecular el cual requiere conocimientos de física y matemáticas. También se ha profundizado en temas como la vectorización, la paralelización con memoria compartida y el desarrollo de aplicaciones científicas. En definitiva, el balance de este trabajo ha sido positivo y se espera que el proyecto acabe consolidándose en GROMACS.

Bibliografía

[1] S.A. Adcock and J.A McCammon. *Molecular Dynamics: Survey of Methods for Simulating the Activity of Protein.* 2006.

[2] D. Frenkel and B. Smit. *Understanding molecular simulations: From algorithms to applications.* 1996.

[3] Frank Jensen. *Introduction to Computational Chemistry.* 1999.

[4] J. P. Ryckaert, G. Ciccotti, and H. J. C. Berendsen. Numerical integration of the cartesian equations of motion of a system with constraints: Molecular dynamics of n-alkanes. *J. Comp. Phys. 23, 327–341*, 1977.

[5] H. C. Andersen. Rattle: A "velocity" version of the shake algorithm for molecular dynamics calculations. *J. Comp. Phys. 52, 24–34*, 1983.

[6] Berk Hess, Henk Bekker, Herman J. C. Berendsen, and Johannes G. E. M. Faaije. Lincs: A linear constraint solver for molecular simulations. *Journal of Computational Chemistry*, 1997.

[7] P. García-Risueño. Constraint implementation based on analytical calculations: a possible way to improve widely used solvers. Technical report.

[8] A. G. Bailey, C. P. Lowe, and A. P. Sutton. Efficient constraint dynamics using milc shake. *Journal of Computational Physics 227*, 2008.

[9] P. García-Risueño, P. Echenique, and J.L. Alonso. Exact and efficient calculation of lagrange multipliers in biological polymers with constrained bond lengths and bond angles: Proteins and nucleic acids as example cases. *Journal of Computational Chemistry 32: 3039-3046*, 2011.

[10] María Astón Serrano Gracia. Implementación e integración en gromacs de un algoritmo eficiente y preciso para imponer ligaduras en simulaciones de dinámica molecular. Proyecto fin de carrera, Universidad de Zaragoza, 2013.

[11] Carl Christian Kjelgaard Mikkelsen, Jesús Alastruey-Benedé, Pablo Ibáñez-Marín, and Pablo García Risueño. Accelerating sparse arithmetic in the context of newton's method for small molecules with bond constraints. In *Parallel Processing and Applied Mathematics - 11th International Conference (PPAM 2015)*, pages 160–171, September 6-9, 2015.

Lista de Figuras

Anexos

Anexo A

Vectorización

Las versiones de ILVES-S implementadas en este trabajo hacen uso de las extensiones vectoriales. A continuación se explica brevemente en qué consisten.

Para esta sección se usará como ejemplo un código que suma dos vectores y guarda el resultado en un tercero, como aparece en la Figura A.1.

```
do i=1,100
    a[i] = b[i] + c[i]
```

Figura A.1: Suma de vectores, versión escalar.

En la versión escalar, para cada iteración hacen falta 4 instrucciones en ensamblador: cargar cada uno de los operandos fuente en registros, hacer la suma y guardar el resultado en memoria. Usando las extensiones vectoriales queda el bucle que aparece en la Figura A.2.

```
do i=1,100,4
    load <b[i],b[i+1],b[i+2],b[i+3]> into Vb
    load <c[i],c[i+1],c[i+2],c[i+3]> into Vc
    Va = Vb + Vc
    store Va into <a[i],a[i+1],a[i+2],a[i+3]>
```

Figura A.2: Suma de vectores, versión vectorizada.

Las extensiones vectoriales permiten operar con registros en los que se almacenan varios elementos. La longitud vectorial se define como el número de elementos que pueden almacenarse en un registro.

En el código de la Figura A.2 se usa una longitud vectorial de cuatro. De esta manera se divide el número de instrucciones por cuatro. Cada iteración de la versión vectorial realiza el trabajo de cuatro iteraciones de la versión escalar.

La vectorización solo es posible si no existen dependencias entre iteraciones. En caso de que la iteración $i + 1$ necesite un dato calculado en la iteración i, el resultado

sería incorrecto. Las instrucciones load se ejecutan de 4 en 4, por lo que la suma de la iteración $i+1$ estaría cargando el dato antes de que la iteración i guarde en memoria el resultado con la instrucción store.

En la Figura A.3 se pueden ver las dependencias para el ejemplo de la suma de vectores. Se observa que no existen dependencias entre iteraciones y por lo tanto el código es vectorizable.

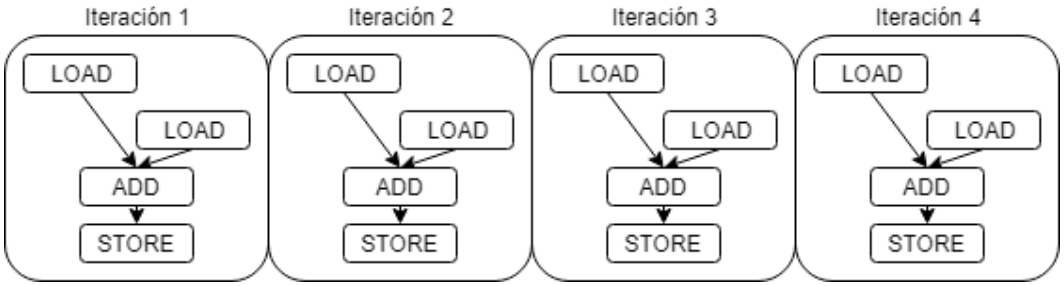

Figura A.3: Dependencias entre las instrucciones de la suma de vectores.

Además, los datos con los que se trabaja están contiguos en memoria ($b[i]$, $b[i+1]$, $b[i+2]$, $b[i+3]$), por lo que para las instrucciones load y store los datos pueden viajar de memoria a registro y viceversa sin necesidad de un reordenamiento.

Para poder ejecutar instrucciones vectoriales son necesarias ALUs y registros especiales capaces de realizar múltiples operaciones al mismo tiempo. En la Figura A.4 se puede ver una representación de las mismas. Las unidades vectoriales cuentan con registros capaces de almacenar varios datos y ALUs que pueden realizar varias operaciones en paralelo.

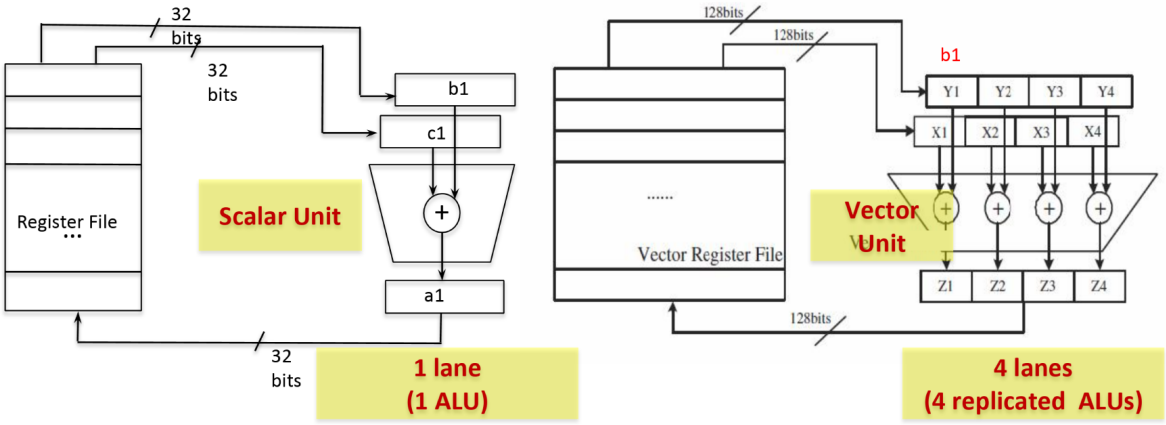

Figura A.4: A la izquierda se puede ver una ALU convencional que realiza las sumas de una en una. A la derecha una ALU vectorial que puede realizar varias operaciones en una sola instrucción, los registros tienen capacidad para varios elementos.

Anexo B

Código

B.1. test-newton.c

```
//------------------------------------------------------------------------------
//      DUMMY STATEMENT 78 CHARACTERS LONG TO FACILITATE PRINTING
//------------------------------------------------------------------------------

/* TEST-NEWTON

   A test program which illustrates the effect of three distinct types of
   optimizations as applied to Newton's method for solving the constraint
   equations for a solvent consisting of many copies of the same molecule

 */

// Standard libraries
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>
#include <getopt.h>

#include "md-newton.h"
#include "newton000.h"
#include "schur.h"
#include "schurSparse.h"
#include "schurSparseVec.h"
#include "lincs.h"
#include "shake.h"
#include "aux.h"
#include "rdtsc.h"
#include "smalloc.h"
#include "sort.h"

#define BYTETOBINARYPATTERNSHORT "%d%d%d"
#define BYTETOBINARYPATTERN "%14d%d%d"
#define BYTETOBINARY(byte) \
  (byte & 0x04 ? 1 : 0), \
```

```
  (byte & 0x02 ? 1 : 0), \
  (byte & 0x01 ? 1 : 0)

#define VERSIONS  7

#define PRINT_MEM  0
#define ULTRAVERBOSE 0

// maximum number of repetitions of the experiment
#define MAX_REPEAT 100

#define HLINE "----------------------------------------------------------------\n"


static void
start_timers(struct timeval *start, unsigned long long *tic)
{
  gettimeofday(start, NULL);
  *tic = rdtscp();
}

static void
stop_timers(struct timeval *start, unsigned long long tic, double *secs, unsigned long lo
{
  struct timeval stop;
  unsigned long long toc = rdtscp();
  gettimeofday(&stop, NULL);
  *tictoc = toc - tic;
  *secs = (stop.tv_sec - start->tv_sec) + (stop.tv_usec - start->tv_usec)/1000000.0;
}

static int verbose = 0;

//------------------------------------------------------------------------------
static const char *optString = "s:m:n:l:r:i:t:p:k:vh?";
// static const char *optString = "s:m:n:l:r:i:t:p:vh?";

static const struct option longOpts[] =
{
  {"suite",    required_argument, NULL, 's'},
  {"molecule", required_argument, NULL, 'm'},
  {"num",         required_argument, NULL, 'n'},
  {"tol",      required_argument, NULL, 'l'},
  {"iter",     required_argument, NULL, 'i'},
  {"seed",     required_argument, NULL, 'r'},
  {"repeat",   required_argument, NULL, 't'},
  {"perturbation", required_argument, NULL, 'p'},
  {"kernel",    required_argument, NULL, 'k'},
  {"verbose",   no_argument,       NULL,  'v'},
  {"help",     no_argument,       NULL,  'h'},
  {NULL,         0,                 NULL, 0}
```

```
};

//------------------------------------------------------------------------
static struct option_help {
  const char *long_opt, *short_opt, *desc;
} opts_help[] = {
  { "--suite", "-s",
    "test-suite [default=1]" },
  { "--molecule", "-m",
    "file describing the molecule" },
  { "--num", "-n",
    "number of molecules [default=16000]" },
  { "--tol", "-l",
    "tolerance [default=1.0e-6]. It is measured as the constraint violation\n"
    "       g = 0.5*(||rab||^2 - (bond length)^2))\n"
    "     The factor 0.5 is only included to give the Jacobian of the constraint function\
    "     a form which CCKM considers aesthetically pleasing.\n"
    "     Set to zero to force the maximun number of iterations." },
  { "--iter", "-i",
    "maximum number of newton iterations [default=10]" },
  { "--seed", "-r",
    "random seed [default=2015]" },
  { "--repeat", "-t",
    "number of repetitions (maximum and minimum measures are discarded)\
    [default=6]" },
#if 0
  { "--nthreads", "-p",
    "number of threads [default=1]" },
#endif
  { "--verbose", "-v",
    "Print detailed output (only for --repeat=1 ) [default = no]" },
  { "--help", "-h",
    "Show program usage"},
  { NULL, NULL, NULL }
};

//------------------------------------------------------------------------
static void
show_usage(char *name, int exit_code)
{
  struct option_help *h;

  printf("usage: %s options\n", name);
  for (h = opts_help; h->long_opt; h++)
  {
    printf(" %s, %s\n ", h->short_opt, h->long_opt);
    printf("   %s\n", h->desc);
  }
  printf("Example:\n");
  printf(" %s -s1 -m solvents/thf.txt -n16000 -r 2015 -i 10 -t 10\n", name);
  exit(exit_code);
```

```
}

#if 0
//---------------------------------------------------------------------------
double compute_mean(int repeat, double *secs)
{
  int i;

  double total = 0;
  double loop_mean;

  for (i=0; i<repeat; i++) {
      total += secs[i];
      //total_tictoc += tictoc[i];
  }

  loop_mean = total / repeat;
  //loop_mean_tictoc = (double) total_tictoc / repeat;

  return loop_mean;
}
#endif

//---------------------------------------------------------------------------
static double
compute_cv2(int repeat, double *secs)
//double compute_cv2(int repeat, double *secs, unsigned long long *tictoc)
{
  int i;
  /* mean, variance and coefficient of variation */
  double total = 0, total_sqrd = 0;
  double loop_mean, loop_var, loop_cv2;
  // double total_tictoc, loop_mean_tictoc;

  /* Variance and standard deviation computation (repeat loop) */
  /* VAR[X] = E[X^2] - E[X]^2 */
  /* STD[X] = sqrt(VAR[X]) */
  for (i=0; i<repeat; i++)
  {
      total += secs[i];
      total_sqrd += (secs[i] * secs[i]);
      // total_tictoc += tictoc[i];
  }

  loop_mean = total / repeat;
  loop_var = (total_sqrd / repeat) - (loop_mean * loop_mean);
  loop_cv2 = loop_var / (loop_mean*loop_mean);
  //loop_mean_tictoc = (double) total_tictoc / repeat;

  return loop_cv2;
}
```

```
//------------------------------------------------------------------------

static void
print_metrics(double *t, int num, int *numit, double *tictoc, int mean)
{
  int i;
  double freq[VERSIONS];

  for (i=0; i<VERSIONS; i++) freq[i] = tictoc[i]/t[i]; // CPU freq estimation
  for (i=0; i<VERSIONS; i++) printf("%16.3f", freq[i]/1000000000.0);
  if (mean) printf(" estimated frequency (GHz, median)\n");
  else      printf(" estimated frequency (GHz)\n");

  for (i=0; i<VERSIONS; i++) printf("%#16.4g", 1000000000.0*t[i]/num);
  if (mean) printf(" ns/molecule (median)\n");
  else      printf(" ns/molecule\n");

  for (i=0; i<VERSIONS; i++) printf("%#16.4g", freq[i]*t[i]/num);
  if (mean) printf(" cycles/molecule (median)\n");
  else      printf(" cycles/molecule\n");

#if 0
  for (i=0; i<VERSIONS; i++) printf("%16.2f", 1000000000.0*t[i]/(num*numit[i]));
  if (mean) printf(" ns/newton_step (median)\n");
  else      printf(" ns/newton_step\n");

  for (i=0; i<VERSIONS; i++) printf("%16.2f", freq[i]*t[i]/(num*numit[i]));
  if (mean) printf(" cycles/newton step (median)\n");
  else      printf(" cycles/newton step\n");
#endif
}
//------------------------------------------------------------------------

static void
print_metrics_kernel(double t, int num, int *numit, double tictoc, int mean)
{
  double freq;

  freq = tictoc/t; // CPU freq estimation
  printf("%16.3f", freq/1000000000.0);
  if (mean) printf(" estimated frequency (GHz, median)\n");
  else      printf(" estimated frequency (GHz)\n");

  printf("%#16.4g", 1000000000.0*t/num);
  if (mean) printf(" ns/molecule (median)\n");
  else      printf(" ns/molecule\n");

  printf("%#16.4g", freq*t/num);
  if (mean) printf(" cycles/molecule (median)\n");
  else      printf(" cycles/molecule\n");
}
```

```
//------------------------------------------------------------------------

#if ULTRAVERBOSE
static void
print_atoms_positions_bond_lenghts(unsigned long int num, molecule_t *mol, real *x_initia
{
      char version[16] = { 0 };

      printf("\n");
      printf("-------------------------------------------------\n");
      printf("PRINTING ATOMS COORDINATES\n\n");
      print_atom_headers(mol);
      print_atoms_positions(num, x_initial, mol, "init");
      print_atoms_positions(num, x, mol, "moved");

      for(int i = 0; i < VERSIONS; i++)
      {
        sprintf(version, "%d%d%d ", BYTETOBINARY(i));
        print_atoms_positions(num, &x_backup[3*mol->m*i], mol, version);
      }
      printf("\n");

      printf("-------------------------------------------------\n");
      printf("PRINTING CONSTRAINT VIOLATIONS\n\n");
      print_bond_headers(mol);
      print_bond_lengths(num, x_initial, mol, "init");
      print_bond_lengths(num, x, mol, "moved");

      for(int i = 0; i < VERSIONS; i++)
      {
        sprintf(version, "%d%d%d ", BYTETOBINARY(i));
        print_bond_lengths(num, &x_backup[3*mol->m*i], mol, version);
      }
      printf("\n");
}
#endif
//------------------------------------------------------------------------

//------------------------------------------------------------------------
// COMPARE RESULTS TO VERSION 000
//------------------------------------------------------------------------

static void
compare_versions(unsigned long int num, int m, real *x_backup)
{
  // Display short message
  // printf("\n");
  // printf("-------------------------------------------------\n");
  // printf("COMPARING VERSIONS AGAINST VERSION 000\n\n");

  // Set the length of the arrays which will be compared
```

```
  int length = num*(3*m);

  // Absolute and relative errors
  real ae[VERSIONS] = { 0 }, re[VERSIONS] = { 0 };

  // Run (destructive) comparisons of schur with 000.
  for(int v = 1; v < VERSIONS; v++)
  {
    // printf("VERSION "BYTETOBINARYPATTERNSHORT" ", BYTETOBINARY(i));
    compare(&length, x_backup, &x_backup[v*(num*3*m)], &ae[v], &re[v]);
  }

  for (int v = 0; v < VERSIONS; v++) printf("%16.2e", ae[v]);
  printf(" absolute diff wrt 000\n");
  for (int v = 0; v < VERSIONS; v++) printf("%16.2e", re[v]);
  printf(" relative diff wrt 000\n");
}
//------------------------------------------------------------------------------

// show one repetition results
static void
show_results_kernel(int kernel, unsigned long int num, real *res, int *rc, int *numit,
                    unsigned long long tictoc[VERSIONS][MAX_REPEAT], double secs[VERSIONS][
{
  double loop_median, loop_median_tictoc;
  // double loop_mean[VERSIONS];

  printf (""BYTETOBINARYPATTERN, BYTETOBINARY(kernel));
  printf(" version\n");

  printf("%#16.3g", secs[kernel][0]);
  printf(" seconds\n");
  printf("%16llu", tictoc[kernel][0]);
  printf(" cycles\n");

  // cast because print_metrics() expects a double argument
  loop_median_tictoc = (double) tictoc[kernel][0];
  loop_median = secs[kernel][0];
  print_metrics_kernel(loop_median, num, numit, loop_median_tictoc, 0);

  printf("%16d", numit[kernel]);
  printf(" iterations\n");
  printf("%16d", rc[kernel]);
  printf(" rc\n");

  // Print the final relative constraint violations for each equation
  printf("%16.8e relative constraint violation\n", res[kernel]);

  printf(""BYTETOBINARYPATTERN, BYTETOBINARY(kernel));
  printf(" version\n");
}
```

```
//------------------------------------------------------------------------

// show one repetition results
static void
show_results(unsigned long int num, real *res, int *rc, int *numit,
             unsigned long long tictoc[VERSIONS][MAX_REPEAT], double secs[VERSIONS][MAX_REP
             molecule_t *mol, real *x_backup)
{
  double loop_median[VERSIONS], loop_median_tictoc[VERSIONS];
  // double loop_mean[VERSIONS];

  for (int i = 0; i < VERSIONS; i++) printf (""BYTETOBINARYPATTERN, BYTETOBINARY(i));
  printf(" version\n");

  for (int i = 0; i < VERSIONS; i++) printf("%#16.3g", secs[i][0]);
  printf(" seconds\n");
  for (int i = 0; i < VERSIONS; i++) printf("%16llu", tictoc[i][0]);
  printf(" cycles\n");
  for (int i = 0; i < VERSIONS; i++) printf("%16.2f", (double) tictoc[0][0]/(double) tict
  printf(" speedup\n");
  // cast because print_metrics() expects a double argument
  for (int i = 0; i < VERSIONS; i++)
  {
    loop_median_tictoc[i] = (double) tictoc[i][0];
    loop_median[i] = secs[i][0];
  }
  print_metrics(loop_median, num, numit, loop_median_tictoc, 0);

  for (int i = 0; i < VERSIONS; i++) printf("%16d", numit[i]);
  printf(" iterations\n");
  for (int i = 0; i < VERSIONS; i++) printf("%16d", rc[i]);
  printf(" rc\n");
  // Print the final relative constraint violations for each equation
  ddm(num, VERSIONS, res, num, "%16.8e", " relative constraint violation\n");

  if (verbose)
  {
    // compare final coordinates with respecto to version 000
    compare_versions(num, mol->m, x_backup);
  }

  for (int i = 0; i < VERSIONS; i++) printf(""BYTETOBINARYPATTERN, BYTETOBINARY(i));
  printf(" version\n");
}
//------------------------------------------------------------------------

// show one repetition results
static void
show_results_init(unsigned long long tictoc[VERSIONS][MAX_REPEAT], double secs[VERSIONS][
{
  for (int i = 0; i < VERSIONS; i++) printf (""BYTETOBINARYPATTERN, BYTETOBINARY(i));
```

```
  printf(" version\n");

  for (int i = 0; i < VERSIONS; i++) printf("%#16.3g", secs[i][0]);
  printf(" seconds\n");
  for (int i = 0; i < VERSIONS; i++) printf("%16llu", tictoc[i][0]);
  printf(" cycles\n");
  // for (int i = 0; i < VERSIONS; i++) printf("%16.2f", (double) tictoc[0][0]/(double) t
  // printf(" speedup\n");

  for (int i = 0; i < VERSIONS; i++) printf(""BYTETOBINARYPATTERN, BYTETOBINARY(i));
  printf(" version\n");
}
//---------------------------------------------------------------------------

// show one repetition results
static void
show_results_init_kernel(int kernel, unsigned long long tictoc[VERSIONS][MAX_REPEAT], dou
{
  printf (""BYTETOBINARYPATTERN, BYTETOBINARY(kernel));
  printf(" version\n");

  printf("%#16.3g", secs[kernel][0]);
  printf(" seconds\n");
  printf("%16llu", tictoc[kernel][0]);
  printf(" cycles\n");
  // for (int i = 0; i < VERSIONS; i++) printf("%16.2f", (double) tictoc[0][0]/(double) t
  // printf(" speedup\n");

  printf(""BYTETOBINARYPATTERN, BYTETOBINARY(kernel));
  printf(" version\n");
}
//---------------------------------------------------------------------------


// show median results of several repetition results
static void
show_median_results(int repeat, unsigned long int num, real *res, int *rc,
                    int *numit, unsigned long long tictoc[VERSIONS][MAX_REPEAT], double sec
{
  /* medians, coefficient of variation and mean */
  double loop_median[VERSIONS], loop_cv2[VERSIONS];
  double loop_median_tictoc[VERSIONS];
  // double loop_mean[VERSIONS];

  // Compute medians
  for (int i = 0; i < VERSIONS; i++)
  {
    insertion_sort_ull(repeat, &tictoc[i][0]);
    loop_median_tictoc[i] = (double) median_ull(repeat, &tictoc[i][0]);

    insertion_sort_double(repeat, &secs[i][0]);
```

```
    loop_median[i] = median_double(repeat, &secs[i][0]);

    loop_cv2[i] = compute_cv2(repeat, &secs[i][0]);
    // loop_mean[i] = compute_mean(repeat, &secs[i][0]);
  }

  printf("      ------------------------------------------");
  printf("-------------------------------------------------");
  printf("-------------------------------------------------\n");
  // for (int i = 0; i < VERSIONS; i++)
  //  printf (""BYTETOBINARYPATTERN, BYTETOBINARY(i));
  // printf(" version\n");
  for (int i = 0; i < VERSIONS; i++) printf("%#16.3g", loop_median[i]);
  printf(" seconds (median)\n");
  //for (i=0; i<VERSIONS; i++) printf("%#16.3g", loop_mean[i]);
  //printf(" seconds (mean)\n");
  for (int i = 0; i < VERSIONS; i++) printf("%#16.3g", loop_median_tictoc[i]);
  printf(" cycles (median)\n");
  for (int i = 0; i < VERSIONS; i++) printf("%16.2f", 100.0*sqrt(loop_cv2[i]));
  printf(" cv (%%)\n");
  for (int i = 0; i < VERSIONS; i++) printf("%16.2f", loop_median[0]/loop_median[i]);
  printf(" speedup (median)\n");
  //for (i = 0; i < VERSIONS; i++) printf("%16.2f", loop_mean[0]/loop_mean[i]);
  //printf(" speedup (mean)\n");

  print_metrics(loop_median, num, numit, loop_median_tictoc, 1);

  for (int i = 0; i < VERSIONS; i++) printf("%16d", numit[i]);
  printf(" iterations (final)\n");
  for (int i = 0; i < VERSIONS; i++) printf("%16d", rc[i]);
  printf(" rc (final)\n");
  ddm(num, VERSIONS, res, num, "%16.8e", " relative constraint violation (final)\n");

  for (int i = 0; i < VERSIONS; i++) printf (""BYTETOBINARYPATTERN, BYTETOBINARY(i));
  printf(" version\n");

  for (int i = 0; i < VERSIONS; i++)
  {
    if (loop_cv2[i] > 0.0025)
    {
      printf("WARNING: coefficient of variation too high (>5%%) for test"
             BYTETOBINARYPATTERNSHORT "\n", BYTETOBINARY(i));
    }
  }
}
//-----------------------------------------------------------------------------

#if PRINT_MEM
static void
printMem()
{
```

```
  FILE* file;
  char line[256];

  file = fopen("/proc/self/status", "r");
  while (fgets(line, sizeof(line), file))
  {
    printf("%s", line);
  }
  fclose(file);
}
#endif
//--------------------------------------------------------------------------

static void
example1(char *filename, int seed, int maxit, unsigned long int num, double tol, int repe
{
  // tol: convergence tolerance

  /* Reads a test molecule from a file and solves NUM copies of the constraint
     equation using several different versions of Newton's method. */

  // ------------------------------------------------------------------------
  // Declaration of internal variables
  // ------------------------------------------------------------------------

  // version counter index for timers
  int v;

  // The number of atoms
  int m;

   // Size of the block in Schur complement method
  int block_size, block_size_par;

  // Data structure for one class of molecule
  molecule_t *mol;

  // Spatial coordinates for the all atoms
  real *x;
   real *x_initial;

  // Archive used to reset the initial coordinates.
  real *xarc;

  // Backup of final coordinates. Used to compare all routines to the base.
  real *x_backup = NULL;

  // The graph of the lower triangular part of A
  graph_t *graph, *graph000;

  // weights pointers
```

```
real *pweight, *pweight000;

// The final residuals
real *res;

// Return code from the different routines
int rc[VERSIONS];

// number of iterations executed
int numit[VERSIONS];

// Cycle and time counters
unsigned long long tictoc[VERSIONS][MAX_REPEAT], tic;
double secs[VERSIONS][MAX_REPEAT];
struct timeval start;

// ------------------------------------------------------------------------
// Start of instructions
// ------------------------------------------------------------------------

// Allocate space for one molecule
mol = (molecule_t *)malloc(sizeof(molecule_t));

// Read the molecule from the file
read_molecule(mol, filename);

// Extract the number of atoms
m = mol->m;

// Check block size
if (m % (VLEN*NTHREADS))
{
  printf("ERROR: the number of atoms (%d) has to be"
         " multiple of the partition number (%d)\n", m, VLEN*NTHREADS);
  exit(1);
}
if (m <= VLEN)
{
  printf("ERROR: the number of atoms (%d) has to be"
         " greater than the vector length (%d)\n", m, VLEN);
  exit(1);
}
 // If the number of atoms is correct, calculate block_size
 block_size = m/VLEN - 1;
 block_size_par = m/(NTHREADS*VLEN) - 1;

// Allocate space for an archive of initial coordinates;
smalloc_aligned(xarc, 3*m*num, AVX_ALIGNMENT);

//------------------------------------------------------------------------
```

```
// Allocate and initialize space for the final residuals,
// all versions of Newton.
smalloc_aligned(res, VERSIONS*num, AVX_ALIGNMENT);

//----------------------------------------------------------------------------
// Analyse the molecule and create the relevant graphs
//----------------------------------------------------------------------------

// Generate the bond graph from the list of bonds
make_bond_graph(mol);

// Allocate space for the adjacency graph of tril(A)
graph000 = (graph_t *)malloc(sizeof(graph_t));

// backup bond_graph
graph = mol->bond_graph;

// Allocate space for Cholesky's graph
mol->chol_graph = (graph_t *)malloc(sizeof(graph_t));

// Calculate weight array and pointer for versions different to newton000
make_weights(mol, graph);
pweight = mol->weights;

// Extract lower triangular pattern of the matrix A
tril(mol->bond_graph, graph000);
mol->bond_graph = graph000;

// Replace the bond graph with the lower triangular portion
// free_graph(mol->bond_graph); free(mol->bond_graph); mol->bond_graph=graph;

// Calculate the weights for the lower triangular half of matrix A
make_weights(mol, graph000);
pweight000 = mol->weights;

// Compute the fill (only affects to newton000)
compute_fill(graph000, mol->chol_graph);
// compute_fill(graph, mol->chol_graph);

 // Allocate and initialize the spatial coordinates for all atoms
 x_initial = fixed_positions(num, mol, seed);
// x_initial = random_positions(num, mol, seed);

// print_molecule(mol, x_initial);

 x = move_initial_positions(num, mol, x_initial, seed);
// x = malloc(3*n*num*sizeof(real));
// for (int k=0; k < 3*num*m; k++)
//    x[k] = x_initial[k];

// print_molecule(mol, x);
```

```
  // printf("PRINTING ATOMS COORDINATES\n\n");
  // print_atom_headers(mol);
  // print_atoms_positions(num, x_initial, mol, "init");
  // print_atoms_positions(num, x, mol, "moved");
  // print_bond_headers(mol);
  // print_bond_lengths(num, x, mol, "moved");

   // Copy the initial positions into the archive
  memcpy(xarc, x, 3*m*num*sizeof(*xarc));
  // for (i=0; i<3*m*num; i++) xarc[i]=x[i];

  // Allocate enough memory to store all final results for future comparison
  if (verbose) smalloc_aligned(x_backup, VERSIONS*num*3*m, AVX_ALIGNMENT);

  //-------------------------------------------------------------------------
  // THIS COMPLETES THE ANALYSIS OF THE GIVEN MOLECULE
  //-------------------------------------------------------------------------

  // Repeat the benchmark REPEAT number of times.
  for (int k = 0; k < repeat; k++)
  {
    // Clear return codes
    for (int i = 0; i < VERSIONS; i++) rc[i] = 0;

    // version counter index
    v = 0;

    //---------------------------------------------------------------------------
    // Display progress
    if (verbose)
    {
      printf (""BYTETOBINARYPATTERNSHORT, BYTETOBINARY(v));
      printf(": APPLYING NEWTON's METHOD VERSION 000 ... ");
      fflush(stdout);
    }
    mol->bond_graph = graph000;
    mol->weights = pweight000;

    // Run standard Newton's algorithm using the general code
    start_timers(&start, &tic);
    rc[v] = newton000(num, mol, x, tol, maxit, &numit[v], &res[v*num]);
    stop_timers(&start, tic, &secs[v][k], &tictoc[v][k]);
    if (verbose) printf("DONE\n");

#if PRINT_MEM
    printMem();
#endif

    // Save the results
    if (verbose)
```

```
    {
      memcpy(&x_backup[v*num*3*m], x, 3*m*num*sizeof(*x_backup));
    }

    // Reset the initial coordinates from the archive
    memcpy(x, xarc, 3*m*num*sizeof(*xarc));

    v++;

#if 0
    //-------------------------------------------------------------------------
    // Display progress
    if (verbose)
    {
      printf (""BYTETOBINARYPATTERNSHORT, BYTETOBINARY(v));
         printf(": APPLYING NEWTON's METHOD VERSION SCHUR ... ");
         fflush(stdout);
      }
    mol->bond_graph = graph;
    mol->weights = pweight;

    // Run standard Newton's algorithm using the schur code
    start_timers(&start, &tic);
    rc[v] = newtonSchur(num, mol, x, tol, maxit, &numit[v], &res[v*num], block_size);
    stop_timers(&start, tic, &secs[v][k], &tictoc[v][k]);
    if (verbose) printf("DONE\n");

    // Save the results
    if (verbose)
    {
      memcpy(&x_backup[v*num*3*m], x, 3*m*num*sizeof(*x_backup));
    }

    // Reset the initial coordinates from the archive
    memcpy(x, xarc, 3*m*num*sizeof(*xarc));

    v++;
#endif

    //-------------------------------------------------------------------------
    // Display progress
    if (verbose)
    {
      printf (""BYTETOBINARYPATTERNSHORT, BYTETOBINARY(v));
      printf(": APPLYING NEWTON's METHOD VERSION SCHUR SPARSE ... ");
      fflush(stdout);
    }
    mol->bond_graph = graph;
    mol->weights = pweight;

    // Run standard Newton's algorithm using the schur code
```

```
    start_timers(&start, &tic);
    rc[v] = newtonSchurSparse(num, mol, x, tol, maxit, &numit[v], &res[v*num], block_size)
    stop_timers(&start, tic, &secs[v][k], &tictoc[v][k]);
    if (verbose) printf("DONE\n");

#if PRINT_MEM
    printMem();
#endif

    // Save the results
    if (verbose)
    {
      memcpy(&x_backup[v*num*3*m], x, 3*m*num*sizeof(*x_backup));
    }

    // Reset the initial coordinates from the archive
    memcpy(x, xarc, 3*m*num*sizeof(*xarc));

    v++;

    //------------------------------------------------------------------------
    // Display progress
    if (verbose)
    {
      printf (""BYTETOBINARYPATTERNSHORT, BYTETOBINARY(v));
      printf(": APPLYING NEWTON's METHOD VERSION SCHUR SPARSE VECTORIZED ... ");
      fflush(stdout);
    }

    // Run standard Newton's algorithm using the schur code
    start_timers(&start, &tic);
    rc[v] = newtonSchurSparseVec(num, mol, x, tol, maxit, &numit[v], &res[v*num], block_si
    stop_timers(&start, tic, &secs[v][k], &tictoc[v][k]);
    if (verbose) printf("DONE\n");

#if PRINT_MEM
    printMem();
#endif

    // Save the results
    if (verbose)
    {
      memcpy(&x_backup[v*num*3*m], x, 3*m*num*sizeof(*x_backup));
    }

    // Reset the initial coordinates from the archive
    memcpy(x, xarc, 3*m*num*sizeof(*xarc));

    v++;

      //------------------------------------------------------------------------
```

```
    // Display progress
    if (verbose)
    {
      printf (""BYTETOBINARYPATTERNSHORT, BYTETOBINARY(v));
      printf(": APPLYING NEWTON's METHOD VERSION SCHUR SPARSE PARALLELIZED ... ");
      fflush(stdout);
    }

    // Run standard Newton's algorithm using the schur code
    start_timers(&start, &tic);
    rc[v] = newtonSchurSparsePar(num, mol, x, tol, maxit, &numit[v], &res[v*num], block_si
    stop_timers(&start, tic, &secs[v][k], &tictoc[v][k]);
    if (verbose) printf("DONE\n");

#if PRINT_MEM
    printMem();
#endif

    // Save the results
    if (verbose)
    {
      memcpy(&x_backup[v*num*3*m], x, 3*m*num*sizeof(*x_backup));
    }

    // Reset the initial coordinates from the archive
    memcpy(x, xarc, 3*m*num*sizeof(*xarc));

    v++;

#if 0
      //-------------------------------------------------------------------------
    // Display progress
    if (verbose)
    {
      printf (""BYTETOBINARYPATTERNSHORT, BYTETOBINARY(v));
      printf(": APPLYING LINCS ... ");
      fflush(stdout);
    }

    // mol->bond_graph = graph000;
    // mol->weights = pweight000;

    // Run LINCS algorithm
    // int maxit_lincs = 1000;
    start_timers(&start, &tic);
    rc[v] = lincs(num, mol, x, x_initial, maxit /* maxit_lincs */, &numit[v], &res[v*num]
    stop_timers(&start, tic, &secs[v][k], &tictoc[v][k]);
    if (verbose) printf("DONE\n");

#if PRINT_MEM
    printMem();
```

```
#endif

    // Save the results (sequential)
    if (verbose)
    {
      memcpy(&x_backup[v*num*3*m], x, 3*m*num*sizeof(*x_backup));
    }

    // Reset the initial coordinates from the archive
    memcpy(x, xarc, 3*m*num*sizeof(*xarc));

    v++;
#endif
      //------------------------------------------------------------------------

    // Display progress
    if (verbose)
    {
      printf (""BYTETOBINARYPATTERNSHORT, BYTETOBINARY(v));
      printf(": APPLYING SHAKE ... ");
      fflush(stdout);
    }

    // mol->bond_graph = graph000;
    // mol->weights = pweight000;

    // Run SHAKE algorithm
    // int maxit_shake = 1000;
    start_timers(&start, &tic);
    rc[v] = shake(num, mol, x, x_initial, tol, maxit /* maxit_shake */, &numit[v], &res[v*
    stop_timers(&start, tic, &secs[v][k], &tictoc[v][k]);
    if (verbose) printf("DONE\n");

    // Save the results (sequential)
    if (verbose)
    {
      memcpy(&x_backup[v*num*3*m], x, 3*m*num*sizeof(*x_backup));
    }

    // Reset the initial coordinates from the archive
    memcpy(x, xarc, 3*m*num*sizeof(*xarc));

    v++;

    //------------------------------------------------------------------------
    // DISPLAY DATA FOR THE CURRENT INSTANCE OF THE BENCHMARK
    //------------------------------------------------------------------------

    if (repeat != 1)
    {
```

```
      if (k == 0)
      {
        for (int i = 0; i < VERSIONS; i++)
          printf (""BYTETOBINARYPATTERN, BYTETOBINARY(i));
        printf(" version\n");
      }

      // dump time
      for (int i = 0; i < VERSIONS; i++) printf("%#16.3g", secs[i][k]);
      printf(" seconds\n");
    }
  } // END OF LOOP REPEATING THE BENCHMARK

  // If there is only one experiment, dump results.
  if (repeat == 1)
  {
#if ULTRAVERBOSE
    print_atoms_positions_bond_lenghts(num, mol, x_initial, x, x_backup);
#endif
    show_results(num, res, rc, numit, tictoc, secs, mol, x_backup);
  }
  else /* repeat != 1 */
  {
    show_median_results(repeat, num, res, rc, numit, tictoc, secs);
  }

  //------------------------------------------------------------------------
  // Release memory directly allocated by the subroutine
  //------------------------------------------------------------------------

  // Residual history
  free(res);

  // Spatial data
  free(x); free(xarc);

  if (verbose) free(x_backup);

  // The bond graph and the graph of the Cholesky factor
  free_graph(mol->bond_graph);
  free_graph(mol->chol_graph);

  mol->bond_graph = graph;
  free(mol->bond_graph); free(graph000);

  // The auxiliary weights
  // free(mol->weights);
  free(pweight); free(pweight000);

  // The inverse mass
  free(mol->invmass);
```

```
  // atoms, bonds, sigma2
  free(mol->atoms); free(mol->bonds); free(mol->sigma2);

  // Name and abbreviation
  free(mol->name); free(mol->abb);

  // The molecule itself.
  free(mol);

  // END OF EXAMPLE 1 - The primary benchmark routine.
}
//------------------------------------------------------------------------------

static void
example2(char *filename, int seed, int maxit, unsigned long int num, double tol, int nste
{
  // tol: convergence tolerance

  // Data structure for one class of molecule
  molecule_t *mol;

  // The number of atoms (m) and the number of bonds (n)
  int m;

  // Total number of coordinates
  int length;

  // Spatial coordinates for the all atoms
   real *x0, *x1, *x2;

  // Backup of final coordinates. Used to compare all routines to the base.
  real *x_backup = NULL;

   // Size of the block in Schur complement method
  int block_size, block_size_par;

  // The graph of the lower triangular part of A
  graph_t *graph, *graph000;

  // weights pointers
  real *pweight, *pweight000;

  // The final residuals
  real *res;

  // Return code from the different routines
  int rc[VERSIONS];

  // number of iterations executed
  int numit[VERSIONS];
```

```
// Cycle and time counters
unsigned long long tictoc[VERSIONS][MAX_REPEAT], tic;
unsigned long long tictoc_init[VERSIONS][MAX_REPEAT];
// unsigned long long tictoc_kernel[VERSIONS][MAX_REPEAT], tictoc_free[VERSIONS][MAX_REP
double secs[VERSIONS][MAX_REPEAT];
double secs_init[VERSIONS][MAX_REPEAT];
// double secs_kernel[VERSIONS][MAX_REPEAT], secs_free[VERSIONS][MAX_REPEAT];
struct timeval start;

////////////////////
// initialization //
////////////////////

// Allocate space for one molecule
mol = (molecule_t *)malloc(sizeof(molecule_t));

// Read the molecule from the file
read_molecule(mol, filename);

// Extract the number of atoms and bonds
m = mol->m;

// Compute the total number of coordinates, 3*m per molecule.
length = 3*m*num;

 // If the number of atoms is correct, calculate block_size
 block_size = m/VLEN - 1;
 block_size_par = m/(NTHREADS*VLEN) - 1;

 // Allocate and initialize the spatial coordinates for all atoms
 x0 = fixed_positions(num, mol, seed);
// x0 = random_positions(num, mol, seed);

// Allocate enough space for NUM copies of the molecule
smalloc_aligned(x1, length, AVX_ALIGNMENT);
smalloc_aligned(x2, length, AVX_ALIGNMENT);

// Allocate enough memory to store all final results for future comparison
if (verbose) smalloc_aligned(x_backup, VERSIONS*num*3*m, AVX_ALIGNMENT);

// Allocate and initialize space for the final residuals,
// all versions of Newton.
smalloc_aligned(res, VERSIONS*num, AVX_ALIGNMENT);

//----------------------------------------------------------------------------
// Analyse the molecule and create the relevant graphs
//----------------------------------------------------------------------------

// Generate the bond graph from the list of bonds
make_bond_graph(mol);
```

```
// Allocate space for the adjacency graph of tril(A)
graph000 = (graph_t *)malloc(sizeof(graph_t));

// backup bond_graph
graph = mol->bond_graph;

// Allocate space for Cholesky's graph
mol->chol_graph = (graph_t *)malloc(sizeof(graph_t));

// Calculate weight array and pointer for versions different to newton000
make_weights(mol, graph);
pweight = mol->weights;

// Extract lower triangular pattern of the matrix A
tril(mol->bond_graph, graph000);
mol->bond_graph = graph000;

// Replace the bond graph with the lower triangular portion
// free_graph(mol->bond_graph); free(mol->bond_graph); mol->bond_graph=graph;

// Calculate the weights for the lower triangular half of matrix A
make_weights(mol, graph000);
pweight000 = mol->weights;

// Compute the fill (only affects to newton000)
compute_fill(graph000, mol->chol_graph);
// compute_fill(graph, mol->chol_graph);

int v = 0;   // version index

//--------------------------------------------------------------------------
// kernel 0

if ((kernel == -1) || (kernel == v))
{
  // Display progress
  if (verbose)
  {
    printf (""BYTETOBINARYPATTERNSHORT, BYTETOBINARY(v));
    printf(": APPLYING NEWTON's METHOD VERSION 000 ... ");
    fflush(stdout);
  }

  // set initial atoms coordinates
  #pragma GCC ivdep
  for (int j=0; j < length; j++) x1[j] = x0[j];

  mol->bond_graph = graph000;
  mol->weights = pweight000;
```

```
    // Seed the generator
    srand(seed);

    // initialize constraint algorithm
    start_timers(&start, &tic);
    newton000_init(mol);
    stop_timers(&start, tic, &secs_init[v][0], &tictoc_init[v][0]);
    for (int step = 0; step < nsteps; step++)
    {
      update_coords(num, mol, x1, x1, perturb);

      // Run standard Newton's algorithm using the newton000
      rc[v] = newton000_kernel(num, mol, x1, tol, maxit, &numit[v], &res[v*num]);
    }
    newton000_free();
    stop_timers(&start, tic, &secs[v][0], &tictoc[v][0]);
    if (verbose) printf("DONE\n");

    // Save the results
    if (verbose)
    {
      memcpy(&x_backup[v*num*3*m], x1, 3*m*num*sizeof(*x_backup));
    }
  }
  v++;

  //------------------------------------------------------------------------
  // kernel 1
  //------------------------------------------------------------------------

#if 0

  if ((kernel == -1) || (kernel == v))
  {
    // Display progress
    if (verbose)
    {
      printf (""BYTETOBINARYPATTERNSHORT, BYTETOBINARY(v));
      printf(": APPLYING NEWTON's METHOD VERSION SCHUR ... ");
      fflush(stdout);
    }

    // set initial atoms coordinates
    #pragma GCC ivdep
    for (int j = 0; j < length; j++) x1[j] = x0[j];

    mol->bond_graph = graph;
    mol->weights = pweight;

    // Seed the generator
    srand(seed);
```

```
    // initialize constraint algorithm
    start_timers(&start, &tic);
    newtonSchur_init(mol, block_size);
    stop_timers(&start, tic, &secs_init[v][0], &tictoc_init[v][0]);
    for (int step = 0; step < nsteps; step++)
    {
      update_coords(num, mol, x1, x1, perturb);

      // Run standard Newton's algorithm using the schur code
      rc[v] = newtonSchur_kernel(num, mol, x1, tol, maxit, &numit[v], &res[v*num]);
    }
    newtonSchur_free();
    stop_timers(&start, tic, &secs[v][0], &tictoc[v][0]);
    if (verbose) printf("DONE\n");

    // Save the results
    if (verbose)
    {
      memcpy(&x_backup[v*num*3*m], x1, 3*m*num*sizeof(*x_backup));
    }
  }
#endif
  v++;

  //------------------------------------------------------------------------
  // kernel 2
  //------------------------------------------------------------------------

  if ((kernel == -1) || (kernel == v))
  {
    // Display progress
    if (verbose)
    {
      printf (""BYTETOBINARYPATTERNSHORT, BYTETOBINARY(v));
      printf(": APPLYING NEWTON's METHOD VERSION SCHUR SPARSE ... ");
      fflush(stdout);
    }

    // set initial atoms coordinates
    #pragma GCC ivdep
    for (int j = 0; j < length; j++) x1[j] = x0[j];

    mol->bond_graph = graph;
    mol->weights = pweight;

    // Seed the generator
    srand(seed);

    // initialize constraint algorithm
    start_timers(&start, &tic);
```

```
  newtonSchurSparse_init(mol, block_size);
  stop_timers(&start, tic, &secs_init[v][0], &tictoc_init[v][0]);
  for (int step = 0; step < nsteps; step++)
  {
    update_coords(num, mol, x1, x1, perturb);

    // Run standard Newton's algorithm using the schur code
    rc[v] = newtonSchurSparse_kernel(num, mol, x1, tol, maxit, &numit[v], &res[v*num]);
  }
  newtonSchurSparse_free();
  stop_timers(&start, tic, &secs[v][0], &tictoc[v][0]);
  if (verbose) printf("DONE\n");

#if 0
  // Save the results
  if (verbose)
  {
    memcpy(&x_backup[v*num*3*m], x1, 3*m*num*sizeof(*x_backup));
  }
#endif
}
v++;

//------------------------------------------------------------------------
// kernel 003
//------------------------------------------------------------------------

if ((kernel == -1) || (kernel == v))
{
  // Display progress
  if (verbose)
  {
    printf (""BYTETOBINARYPATTERNSHORT, BYTETOBINARY(v));
    printf(": APPLYING NEWTON's METHOD VERSION SCHUR SPARSE VECTORIZED ... ");
    fflush(stdout);
  }

  // set initial atoms coordinates
  #pragma GCC ivdep
  for (int j=0; j < length; j++) x1[j] = x0[j];

  mol->bond_graph = graph;
  mol->weights = pweight;

  // Seed the generator
  srand(seed);

  // initialize constraint algorithm
  start_timers(&start, &tic);
  newtonSchurSparseVec_init(mol, block_size);
  stop_timers(&start, tic, &secs_init[v][0], &tictoc_init[v][0]);
```

```
  for (int step = 0; step < nsteps; step++)
  {
    update_coords(num, mol, x1, x1, perturb);
  //rc[v] = newtonSchurSparseVec(num, mol, x1, tol, maxit, &numit[v], &res[v*num], block
    // Run standard Newton's algorithm using the schur code
    rc[v] = newtonSchurSparseVec_kernel(num, mol, x1, tol, maxit, &numit[v], &res[v*num]
  }
  newtonSchurSparseVec_free();
  stop_timers(&start, tic, &secs[v][0], &tictoc[v][0]);
  if (verbose) printf("DONE\n");

  // Save the results
  if (verbose)
  {
    memcpy(&x_backup[v*num*3*m], x1, 3*m*num*sizeof(*x_backup));
  }
}
v++;

//------------------------------------------------------------------------
// kernel 004
//------------------------------------------------------------------------

if ((kernel == -1) || (kernel == v))
{
  // Display progress
  if (verbose)
  {
    printf (""BYTETOBINARYPATTERNSHORT, BYTETOBINARY(v));
    printf(": APPLYING NEWTON's METHOD VERSION SCHUR SPARSE PARALLELIZED ... ");
    fflush(stdout);
  }

  // set initial atoms coordinates
  #pragma GCC ivdep
  for (int j=0; j < length; j++) x1[j] = x0[j];

  mol->bond_graph = graph;
  mol->weights = pweight;

  // Seed the generator
  srand(seed);

  // initialize constraint algorithm
  start_timers(&start, &tic);
  newtonSchurSparsePar_init(mol, block_size);
  stop_timers(&start, tic, &secs_init[v][0], &tictoc_init[v][0]);
  for (int step = 0; step < nsteps; step++)
  {
    update_coords(num, mol, x1, x1, perturb);
  //rc[v] = newtonSchurSparsePar(num, mol, x1, tol, maxit, &numit[v], &res[v*num], block
```

```
    // Run standard Newton's algorithm using the schur code
    rc[v] = newtonSchurSparsePar_kernel(num, mol, x1, tol, maxit, &numit[v], &res[v*num]
  }
  newtonSchurSparsePar_free();
  stop_timers(&start, tic, &secs[v][0], &tictoc[v][0]);
  if (verbose) printf("DONE\n");

  // Save the results
  if (verbose)
  {
    memcpy(&x_backup[v*num*3*m], x1, 3*m*num*sizeof(*x_backup));
  }
}
v++;

//------------------------------------------------------------------------
// kernel 005
//------------------------------------------------------------------------

#if 0
if ((kernel == -1) || (kernel == v))
{
  // Display progress
  if (verbose)
  {
    printf (""BYTETOBINARYPATTERNSHORT, BYTETOBINARY(v));
    printf(": APPLYING NEWTON's METHOD VERSION LINCS ... ");
    fflush(stdout);
  }

  // set initial atoms coordinates
  #pragma GCC ivdep
  for (int j = 0; j < length; j++) x1[j] = x0[j];

  mol->bond_graph = graph;
  mol->weights = pweight;

  // Seed the generator
  srand(seed);

  // initialize constraint algorithm
  start_timers(&start, &tic);
  lincs_init(mol);
  stop_timers(&start, tic, &secs_init[v][0], &tictoc_init[v][0]);
  for (int step = 0; step < nsteps; step++)
  {
    update_coords(num, mol, x1, x2, perturb);

    // Run LINCS
    rc[v] = lincs_kernel(num, mol, x2, x1, maxit /* maxit_lincs */, &numit[v], &res[v*nu
    for (int j = 0; j < length; j++) x1[j] = x2[j];
```

```
  }
  lincs_free();
  stop_timers(&start, tic, &secs[v][0], &tictoc[v][0]);
  if (verbose) printf("DONE\n");

  // Save the results
  if (verbose)
  {
    memcpy(&x_backup[v*num*3*m], x1, 3*m*num*sizeof(*x_backup));
  }
}
#endif
v++;

//------------------------------------------------------------------------
// kernel 006
//------------------------------------------------------------------------

if ((kernel == -1) || (kernel == v))
{
  // Display progress
  if (verbose)
  {
    printf (""BYTETOBINARYPATTERNSHORT, BYTETOBINARY(v));
    printf(": APPLYING NEWTON's METHOD VERSION SHAKE ... ");
    fflush(stdout);
  }

  // set initial atoms coordinates
  #pragma GCC ivdep
  for (int j = 0; j < length; j++) x1[j] = x0[j];

  mol->bond_graph = graph;
  mol->weights = pweight;

  // Seed the generator
  srand(seed);

  // initialize constraint algorithm
  start_timers(&start, &tic);
  shake_init(mol);
  stop_timers(&start, tic, &secs_init[v][0], &tictoc_init[v][0]);
  for (int step = 0; step < nsteps; step++)
  {
    update_coords(num, mol, x1, x2, perturb);

    // Run shake algorithm
    rc[v] = shake_kernel(num, mol, x2, x1, tol, maxit /* maxit_shake */, &numit[v], &res
    for (int j = 0; j < length; j++) x1[j] = x2[j];
  }
  shake_free();
```

```
    stop_timers(&start, tic, &secs[v][0], &tictoc[v][0]);
    if (verbose) printf("DONE\n");

    // Save the results
    if (verbose)
    {
      memcpy(&x_backup[v*num*3*m], x1, 3*m*num*sizeof(*x_backup));
    }
  }
  v++;
  //------------------------------------------------------------------------

#if 0
  // dump time
  for (int i = 0; i < VERSIONS; i++) printf("%#16.3g", secs[i][k]);
  printf(" seconds\n");
#endif

#if ULTRAVERBOSE
    print_atoms_positions_bond_lenghts(num, mol, x_initial, x, x_backup);
#endif

  if (kernel == -1)
    show_results(num, res, rc, numit, tictoc, secs, mol, x_backup);
  else
    show_results_kernel(kernel, num, res, rc, numit, tictoc, secs);

  printf(HLINE);

  printf("\n\ninit() results\n");
  if (kernel == -1)
    show_results_init(tictoc_init, secs_init);
  else
    show_results_init_kernel(kernel, tictoc_init, secs_init);
  printf(HLINE);

#if 0
  for (int i = 0; i < VERSIONS; i++)
  {
    tictoc_kernel[i][0] = tictoc[i][0] - tictoc_init[i][0];
    secs_kernel[i][0] = secs[i][0]  - secs_init[i][0];
  }

  printf("\n\nkernel() results\n");
  show_results(num, res, rc, numit, tictoc_kernel, secs_kernel, mol, x_backup);
#endif
  // END OF EXAMPLE 2
}
//--------------------------------------------------------------------------

int main(int argc, char *argv[])
```

```
{
  // ------------------------------------------------------------------------
  // Declaration of internal variables
  // ------------------------------------------------------------------------

  // The number of the test suite
  int ts = 1;

  // The file describing the molecule
  char filename[256] = { 0 };

  // Random seed, maximum number of Newton steps, number of molecules
  int seed = 2015, maxit = 10;
  unsigned long int num = 16000;
  int repeat = 6;
  // int nthreads = 1;
  int n, option = 0;
  // requested tolerance for the constraint function
  double tol = 1e-6;
  // atom perturbation in nm.
  double pert = PERTURB;
  int kernel = -1;

  // ------------------------------------------------------------------------
  // Start of instructions
  // ------------------------------------------------------------------------

  /* Read the command line parameters. Values for ALL parameters MUST be
     supplied, but not all parameters are relevant for each test-suite.
     CCKM has not thought of a simple way around this and it prepared to
     live with this for a while :)
  */

  printf("Program version: 20180619\n");
  printf(HLINE);

  if (argc < 2)
    show_usage(argv[0], 0);

  while(1)
  {
    option = getopt_long(argc, argv, optString, longOpts, NULL /* &longIndex */);
    if (option == -1) break;

    switch(option)
    {
      case 's':  /* suite number */
        n = sscanf(optarg, "%d", &ts);
        if ((ts < 1) || (ts > 2))
        {
          printf("ERROR: incorrect suite number\n\n");
```

```
    exit(1);
  }
  break;

case 'm': /* path to the file that describes the molecule*/
  strncpy(filename, optarg, sizeof(filename));
  break;

case 'n':  /* number of molecules */
  n = sscanf(optarg, "%lu", &num);
  if ((num <= 0) || (n == 0)) {
    printf("ERROR: the number of molecules has to be a positive number\n\n");
    exit(1);
  }
  break;

case 'l':  /* tolerance */
  n = sscanf(optarg, "%lf", &tol);
  if (n == 0) {
    printf("ERROR: bad tolerance value\n\n");
    exit(1);
  }
  break;

case 'i':  /* max. number of newton iterations */
  n = sscanf(optarg, "%d", &maxit);
  if ((maxit <= 0) || (n == 0))
  {
    printf("ERROR: the maximum number of iterations"
           " has to be a positive number\n\n");
    exit(1);
  }
  break;

case 't':  /* number of repetitions (suite 1)/time steps (suite 2) */
  n = sscanf(optarg, "%d", &repeat);
  if ((repeat <= 0) || (n == 0))
  {
    printf("ERROR: the number of repetitions/time steps"
           " has to be a positive number\n\n");
    exit(1);
  }
  break;

case 'r':  /* random seed */
  n = sscanf(optarg, "%d", &seed);
  break;

case 'p':  /* perturbation */
  n = sscanf(optarg, "%lf", &pert);
  if ((pert < 0) || (pert > 2) || (n == 0))
```

```
      {
        printf("ERROR: the perturbation has to be between 0 and 2 nm.\n\n");
        exit(1);
      }
      break;

    case 'k':
      n = sscanf(optarg, "%d", &kernel);
      break;

    case 'v':
      verbose = 1;
      break;

    case 'h':
      show_usage(argv[0], 0);
      break;

    default:
      show_usage(argv[0], 1);
  }
}

if (filename[0] == 0)
{
   printf("ERROR: the file describing the molecule has not been specified\n\n");
  exit(1);
}

if ((ts == 1) && (verbose) && (repeat > 1))
{
   printf("ERROR: verbose cannot be set with repetitions > 1 for test-suite 1\n\n");
  exit(1);
}

if ((ts == 1) && (repeat > MAX_REPEAT))
{
  printf("ERROR: the maximum number of repetitions is %d\n\n", MAX_REPEAT);
  exit(1);
}

printf("** executing ts-%d with parameters:\n", ts);
printf("- Molecule filename: %s\n", filename);
printf("- Number of molecules: %lu\n", num);
printf("- Random seed: %d\n", seed);
printf("- Tolerance: %e\n", tol);
printf("- Maximum number of iterations: %d\n", maxit);
printf("- Number of threads: %d\n", NTHREADS /* nthreads */);
printf("- Absolute perturbation: %e nm\n", pert);
printf("- Kernel: %d (-1 => all)\n", kernel);
printf("- Verbose: %d\n", verbose);
```

```
  printf(HLINE);
  printf("Each algorithm kernel will be executed %d %s.\n", repeat, ts == 1? "times" : "t
  // printf("The *best* throughput will be reported (excluding the first iteration).\n");
  printf(HLINE);

  switch(ts)
  {
    case 1:
      example1(filename, seed, maxit, num, tol, repeat);
      break;
    case 2:
      example2(filename, seed, maxit, num, tol, repeat, pert, kernel);
      break;
#if 0
    case 3:
      example3(filename, seed, maxit, num);
      break;
#endif
    default:
      printf("Invalid test-suite specified\n");
      break;
  }
  // Standard return
  return 0;
}
```

B.2. md-newton.h

```
//-----------------------------------------------------------------------------
//-- DUMMY STATEMENT 78 CHARACTERS LONG TO ENSURE THE LINES ARE NOT TOO LONG -
//-----------------------------------------------------------------------------

/* MD-NEWTON

   Different versions of Newton's method for molecules with bond constraints.

*/

#ifndef MD_NEWTON
#define MD_NEWTON

// Modules written specifically for the MD project
#include "md-sparse.h"


// Standard libraries
#include <math.h>
#include <malloc.h>
```

```
// Perturbation in nm
#define PERTURB 0.02
// perturbation is between (-PERTURB/2) and (+PERTURB/2) = (-0.01,+0.01)
// since bond length is 0.1, relative perturbation is between -10% and +10%

#define MAKE_MATRIX_OPT 1
#define NTHREADS 8
#define DISPLAY_TIME_SCHUR 0
#define TAU_OUTPUT 0

// ------------------------------------------------------------------------
// COMPILE TIME CONSTANTS
// ------------------------------------------------------------------------

//------------------------------------------------------------------------
// DATA STRUCTURES
//------------------------------------------------------------------------

//------------------------------------------------------------------------
// SUBROUTINE PROTOTYPES WITH BRIEF DESCRIPTION
//------------------------------------------------------------------------

// Generate random initial positions for a group of identical molecules
real *
random_positions(int num, molecule_t *mol, int seed);

/* Initializes the positions of NUM copies of the same linear chain molecule */
real *
fixed_positions(int num, molecule_t *mol, int seed);

// Allocate space and generate new positions from the positions of a previous step emulat
real *
move_initial_positions(int num, molecule_t *mol, real *x_initial, int seed);

// Update new positions from the positions of a previous step emulated
void
update_coords(int num, molecule_t *mol, real *x1, real *x2, double perturb);

#endif // MD_NEWTON
```

B.3. md-newton.c

```
//------------------------------------------------------------------------
//-- DUMMY STATEMENT 78 CHARACTERS LONG TO ENSURE THE LINES ARE NOT TOO LONG -
//------------------------------------------------------------------------

/* MD-NEWTON

   Different versions of Newton's method for molecules with bond constraints.
```

```
*/

#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <pthread.h>
#include <time.h>
#include <sys/time.h>

#include "md-newton.h"
#include "smalloc.h"
#include "aux.h"
#include "md-sparse.h"
#include "schur.h"
#include "schurSparse.h"
#include "schurSparseVec.h"
#include "vectorization.h"
#include "perf.h"

#ifndef M_PI
  #define M_PI (3.14159265358979323846)
#endif

// Coordinate initialization flags, used at fixed_positions()
// 1-> 1D: atoms placed in a line (X axis)
// 2-> 2D: atoms placed in a plane (XY plane)
// 3-> 3D: atoms placed in 3D space (XYZ)
#define MOLECULE_LAYOUT 3

//------------------------------------------------------------------------------
void update_coords(int num, molecule_t *mol, real *x1, real *x2, double perturb)
{
  //----------------------------------------------------------------------------
  // DECLARATION OF INTERNAL VARIABLES
  //----------------------------------------------------------------------------

  // Number of atoms
  int m;

  // Total number of coordinates
  int length;

  //----------------------------------------------------------------------------
  // START OF INSTRUCTIONS
  //----------------------------------------------------------------------------

  // Extract the number of atoms
  m = mol->m;

  // Compute the total number of coordinates, 3*m per molecule.
```

```
  length = 3*m*num;

  // Seed the generator
  // Since update_coords() is used in a loop, srand() has to be called before that loop
  // srand(seed);

  for(int i=0; i < length; i++)
  {
      x2[i] = x1[i] + perturb*((1.0*rand()/RAND_MAX) - 0.5);
   }
}
//------------------------------------------------------------------------

real *move_initial_positions(int num, molecule_t *mol, real *x_initial, int seed)
{
  //------------------------------------------------------------------------
  // DECLARATION OF INTERNAL VARIABLES
  //------------------------------------------------------------------------

  // Number of atoms
  int m;

  // Total number of coordinates
  int length;

  // Local pointer to array of coordinates
  real *lx;

  //------------------------------------------------------------------------
  // START OF INSTRUCTIONS
  //------------------------------------------------------------------------

  // Extract the number of atoms
  m = mol->m;

  // Compute the total number of coordinates, 3*m per molecule.
  length = 3*m*num;

  // Allocate enough space for NUM copies of the molecule
  smalloc_aligned(lx, length, AVX_ALIGNMENT);

  // Seed the generator
  srand(seed);

  for(int i=0; i < length; i++)
  {
      lx[i] = x_initial[i] + PERTURB*((1.0*rand()/RAND_MAX) - 0.5);
   }

#if 0
  for(int i=0; i < num*m; i++)
```

```
  {
      lx[i*3+XX] = x_initial[i*3+XX] + PERTURB*((1.0*rand()/RAND_MAX) - 0.5);
      lx[i*3+YY] = x_initial[i*3+YY] + PERTURB*((1.0*rand()/RAND_MAX) - 0.5);
      lx[i*3+ZZ] = x_initial[i*3+ZZ] + PERTURB*((1.0*rand()/RAND_MAX) - 0.5);
   }
#endif
  return lx;
}

//------------------------------------------------------------------------------
real *random_positions(int num, molecule_t *mol, int seed)
{
  /* Generates random positions for NUM copies of the same molecule using
     a specified SEED. This is a wrapper around the built-in random number
     generator */

  //------------------------------------------------------------------------
  // DECLARATION OF INTERNAL VARIABLES
  //------------------------------------------------------------------------

  // Number of atoms
  int m;

  // Total number of coordinates
  int length;

  // Local pointer to array of coordinates
  real *lx;

  //------------------------------------------------------------------------
  // START OF INSTRUCTIONS
  //------------------------------------------------------------------------

  // Extract the number of atoms
  m = mol->m;

  // Compute the total number of coordinates, 3*m per molecule.
  length = 3*m*num;

  // Allocate enough space for NUM copies of the molecule
  smalloc_aligned(lx, length, AVX_ALIGNMENT);

  // Fill in random numbers
   random_array(length, lx, seed);

  // Return a pointer to the array;
  return lx;
}

//------------------------------------------------------------------------------
```

```
/* Iniitializes the positions of NUM copies of the same linear chain molecule */
real *fixed_positions(int num, molecule_t *mol, int seed)
{
  // Local pointer to array of coordinates
  real *lx, *aux;

  float azimut_angle, polar_angle;

   int nAtoms = mol->m;
   int nConstrains = mol->n;

  // Seed the generator
  srand(seed);

  // Allocate enough space for NUM copies of the molecule
  lx = malloc(3*nAtoms*num*sizeof(real));

  for (int i = 0; i < num; i++)
  {
    aux = &lx[i*nAtoms*3];
    aux[XX] = 0.0;
    aux[YY] = 0.0;
    aux[ZZ] = 0.0;

    // Loop over the bonds
    for (int j = 0; j < nConstrains; j++)
    {
      int a = mol->bonds[2*j];
      int b = mol->bonds[2*j+1];

      switch (MOLECULE_LAYOUT)
      {
        case 1:
          azimut_angle = 0;
          polar_angle = M_PI/2;
          break;

        case 2:
          // generate random number between -pi and +pi (-180->180)
          azimut_angle = M_PI*(2.0*rand()/RAND_MAX - 1.0);
          polar_angle = M_PI/2;
          break;

        case 3:
          // generate random number between -pi and +pi (-180->180)
          azimut_angle = M_PI*(2.0*rand()/RAND_MAX - 1.0);

          // generate random number between -pi/2 and +pi/2 (-90->90)
          polar_angle = M_PI*(1.0*rand()/RAND_MAX - 0.5);
          break;
      }
```

```
      // printf("[%02d] %8.4f /%8.4f (%6.1f /%6.1f)\n", j, azimut_angle, polar_angle, az

      aux[3*b+XX] = aux[3*a+XX] + mysqrt(mol->sigma2[j])*sin(polar_angle)*cos(azimut_angle
      aux[3*b+YY] = aux[3*a+YY] + mysqrt(mol->sigma2[j])*sin(polar_angle)*sin(azimut_angle
      aux[3*b+ZZ] = aux[3*a+ZZ] + mysqrt(mol->sigma2[j])*cos(polar_angle);
    }
  }

  // Return a pointer to the array;
  return lx;
}
//-------------------------------------------------------------------------
```

B.4. md-sparse.h

```
//-------------------------------------------------------------------------
// STATEMENT 78 CHARACTERS LONG TO ENSURE THE LINES ARE NOT TOO LONG -
//-------------------------------------------------------------------------

/* MD-SPARSE

   Routines for applying Newton's method in the context of molecular dynamics.

*/

#ifndef MD_SPARSE_GUARD
#define MD_SPARSE_GUARD

// Modules written specifically for the MD project
#include "matrix.h"
#include "chol.h"
#include "vectorization.h"

// Standard libraries
#include <math.h>

// ------------------------------------------------------------------------
// COMPILE TIME CONSTANTS
// ------------------------------------------------------------------------

// Vectors with 3m components are used to given the location of m atoms
// Atom i uses components 3*i+XX, 3*i+YY, 3*i+ZZ

#define XX 0
#define YY 1
#define ZZ 2

//-------------------------------------------------------------------------
```

```
// Data structure(s)
//---------------------------------------------------------------------------

/* This structure contains all the information which can be derived from the
   bond list and the list of masses. As soon as x, and y are given, this is
   the information you need to compute the matrix A = A(x,y) rapidly with
   monotone access of the main memory.

   You will only need one structure for each type of molecule present in your
   simulation.
*/

typedef struct molecule {
  char    *name;        // name of molecule
  char    *abb;         // abbreviation of name
  int      m;           // number of atoms
  int      n;           // number of bonds
  int     *atoms;       // the element of each atom
  real    *invmass;     // invmass[i] is 1/mass of the ith atom
  int     *bonds;       // sequential bond list
  real    *sigma;       // sequential list of bond lengths
  real    *sigma2;      // sequential list of square of bond lengths
  graph_t *atomic_graph; // the atomic structure
  graph_t *bond_graph;  // the line graph of the atomic graph
  graph_t *chol_graph;  // the adjacency graph of the Cholesky factor
  real    *weights;     // weights used to compute matrix A(x,y)
} molecule_t;

//---------------------------------------------------------------------------
// Subroutine/function prototypes with brief description
//---------------------------------------------------------------------------

// Computes the vector from atom a to atom b.
void dr(real *vec, int a, int b, real *dest);

// Update spatial data for a (bonded) pair of atoms a and b
void ur(real *vec, int a, int b, real alpha, real *aux);

// Compute scalar product of length 3
real sp3(real *a, real *b);

// Builds the bond graph from the bond list
void make_bond_graph(molecule_t *mol);

// Renumber the bonds using a given permutation
void renumber_bonds(molecule_t *mol, int *p);

// Computes auxiliary weights needed for the construction of A
void make_weights(molecule_t *mol, graph_t *graph);

// Free memory allocated by make_weights
```

```
void free_weights(molecule_t *mol);

// Allocate space for the displacement vector
real **alloc_displacements(molecule_t *mol);

// Free space for the displacement vector
void free_displacements(real **displacements);

// Compute the displacement vector
void compute_displacements(molecule_t *mol, real *x, real **displacements);

// Fills in the values of the matrix A(x,y) = Dg(x)inv(M)Dg(y)
void make_matrix(molecule_t *mol, real *x, matrix_t *matrix);
void make_matrix_opt(molecule_t *mol, matrix_t *matrix, real *displacements[3]);
void make_matrix_vec(molecule_t *mol, real *x, matrix_t *matrix,
     real *valVec, int *xadjVec, real *fillLeft, real *fillRight,
     int block_size, int offset, int isFirst, int isLast);
void make_matrix_vec_opt(molecule_t *mol, matrix_t *matrix,
     real *valVec, int *xadjVec, real *fillLeft, real *fillRight,
     int block_size, int offset, int isFirst, int isLast, real **displacements);
void make_matrix_Svec(molecule_t *mol, real *x, matrix_t *matrix,
     int *xadjVec, real schurMatrix[][3], int block_size, int schur_size);
void make_matrix_Svec_opt(molecule_t *mol, matrix_t *matrix, int *xadjVec,
                  real schurMatrix[][3], int block_size, int schur_size, real **displace

// Allocates enough memory to hold the Jacobian Dg() in various formats
void init_jacobian(molecule_t *mol, int format, matrix_t *Dg);

// Builds the Jacobian matrix Dg(x) explicitly
void make_jacobian(molecule_t *mol, real *x, matrix_t *a);

// Compute z = z + alpha * op(Dg(x))*y, without forming Dg(x) explicitly
void apply_jacobian(char *trans, real alpha, molecule_t *mol,
         real *restrict x, real * restrict y, real * restrict z);

// Compute z = x + alpha * op(Dg(x))*y, without forming Dg(x) explicitly
void apply_jacobian_mod(char *trans, real alpha, molecule_t *mol,
    real *restrict x, real * restrict y, real * restrict z);
void apply_jacobian_vec(real alpha, molecule_t *mol, real * restrict x,
     real * restrict gVec, real * restrict z, int block_size, int offset);
void apply_jacobian_Svec(real alpha, molecule_t *mol, real * restrict x,
     real * restrict gSchurVec, real * restrict z, int block_size, int schur_size);

// Specialized fusion of BACKWARD from CHOL and APPLY_JACOBIAN (trans="T")
void bajt(int n, int *xadj, int *adj, real * restrict val, real * restrict f,
     real alpha, molecule_t *mol, real * restrict x, real * restrict z);

// Evaluates the constraint function and computes maximum violation
real evaluate_constraint_function(molecule_t *mol, real * restrict x,
             real * restrict g);
real evaluate_constraint_function_vec(molecule_t *mol, real * restrict x,
```

```
     real * restrict gVec, int block_size, int offset);
// real
real evaluate_constraint_function_Svec(molecule_t *mol, real * restrict x,
            real * restrict gSchurVec, int block_size, int schur_size);

// computes the distance between atoms a and b
real compute_atom_distance(real *x, int a, int b);

// Reads the description of a molecule from a text file
void read_molecule(molecule_t *mol, char *filename);

// Prints the description and positions of a molecule
void print_molecule(molecule_t *mol, real *positions);

void print_molecule_coordinates(molecule_t *mol, real *positions);

void print_atom_headers(molecule_t *mol);

void print_bond_headers(molecule_t *mol);

void print_atoms_positions(int num, real *positions, molecule_t *mol, char *string);

void print_bond_lengths(int num, real *positions, molecule_t *mol, char *string);

// Extracts one sparse matrix from another; can make room for future fill
void extract_matrix(int m, matrix_t *a, matrix_t *b, real *work);


#endif // MD_SPARSE_GUARD
```

B.5. md-sparse.c

```
//-------------------------------------------------------------------------
// STATEMENT 78 CHARACTERS LONG TO ENSURE THE LINES ARE NOT TOO LONG -
//-------------------------------------------------------------------------

/* MD-SPARSE

   Routines for applying Newton's method in the context of molecular dynamics.

*/

// Standard libraries
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>

// Project specific libraries
```

```
#include "aux.h"        // access max and min macros
#include "md-sparse.h"
#include "smalloc.h"

//------------------------------------------------------------------------
void
print_atom_headers(molecule_t *mol)
{
  // Extract the number of atoms
  int m = mol->m;

  printf("atoms");
  for(int i=0; i < m/2; i++)
  {
    printf ("%24d", i);
  }
  printf("\n");
}
//------------------------------------------------------------------------

void
print_bond_headers(molecule_t *mol)
{
  // Extract the number of bonds
  int n = mol->n;

  printf("bonds");
  for(int i=0; i < n/2; i++)
  {
    printf ("%34d", i);
  }
  printf("\n");
}
//------------------------------------------------------------------------

void
print_atoms_positions(int num, real *positions, molecule_t *mol, char *string)
{
  // Extract the number of atoms
  int m = mol->m;

  printf ("%5s|", string);
  for(int j=0; j < m/2; j++)
  {
    printf("%6.3fX,%6.3fY,%6.3fZ|", positions[3*j+XX], positions[3*j+YY], positions[3*j+Z
  }
  printf("\n");
}
//------------------------------------------------------------------------

void
```

```
print_bond_lengths(int num, real *positions, molecule_t *mol, char *string)
{
  // Extract the number of bonds
  int n = mol->n;

  printf ("%5s|", string);
  for(int j=0; j < n/2; j++)
  {
    int a = mol->bonds[2*j];
    int b = mol->bonds[2*j+1];
    printf("%9.2e = %9.2e - %9.2e|",
           compute_atom_distance(positions, a, b) - mysqrt(mol->sigma2[j]),
           compute_atom_distance(positions, a, b),
           mysqrt(mol->sigma2[j]));
  }
  printf("\n");
}
//------------------------------------------------------------------------------

void dr(real *vec, int a, int b, real *dest) {

  /* Computes the vector from atom a to atom b, storing the result.

     This function is short and called MANY times which is should be inlined.
  */

  dest[XX]=vec[b*3+XX]-vec[a*3+XX];
  dest[YY]=vec[b*3+YY]-vec[a*3+YY];
  dest[ZZ]=vec[b*3+ZZ]-vec[a*3+ZZ];
}
//------------------------------------------------------------------------------

void ur(real *vec, int a, int b, real alpha, real *aux) {

  /* UR = Update R

     This is an auxiliary routine which updates spatial information for a pair
     of atoms a and b. It is used to compute the action of the Jacobian of the
     bond constraint function.

  */

  // Update spatial data for atom a
  vec[a*3+XX]=vec[a*3+XX]-alpha*aux[XX];
  vec[a*3+YY]=vec[a*3+YY]-alpha*aux[YY];
  vec[a*3+ZZ]=vec[a*3+ZZ]-alpha*aux[ZZ];

  // Update spatial dat for atom b
  vec[b*3+XX]=vec[b*3+XX]+alpha*aux[XX];
  vec[b*3+YY]=vec[b*3+YY]+alpha*aux[YY];
  vec[b*3+ZZ]=vec[b*3+ZZ]+alpha*aux[ZZ];
```

```
}


//------------------------------------------------------------------------
real sp3(real *a, real *b) {

/* Computes the scalar product between two vectors of length 3.

   TO DO: Determine if it is NECESSARY to fight over/underflow as in LAPACK.
   This is a nontrivial question, which should be answered in the context of
   the entire process, i.e. the solution of the constraint equation.
 */

  real aux;

  // It is NOT a mistake that we do not being with nullifying aux
  aux =a[XX]*b[XX]; // Note missing "+"
  aux+=a[YY]*b[YY];
  aux+=a[ZZ]*b[ZZ];
  return aux;
}

//------------------------------------------------------------------------
void make_bond_graph(molecule_t *mol) {
  /* Builds the bond graph from the bond list

     Description of interface

     ON ENTRY:
         mol->m      the number of atoms
         mol->n      the number of bonds
         mol->bonds  sequential list of n bonds

     ON EXIT:
         mol->graph  a pointer to a compact representation of the bond graph

     No other variables are modified!

     In the bond graph there is an edge between vertices i and j if and only
     if bonds i and j have one atom in common. The bond graph is stored using
     two arrays called XADJ and ADJ. Each adjacency list is stored in strictly
     increasing order. The n lists are stored in strictly increasing order in
     the array ADJ. The number XADJ[j] is the index inside ADJ of the first
     entry of the jth adjacency list. XADJ[n] points just beyond the end of
     ADJ and is the length of ADJ.

     IDEA:

     Every atom participates in at most K bonds. Suppose that we have m
     auxiliary lists, such that the ath list gives the bonds which atom a
```

```
   partakes in. Then for each bond i, involving atoms a(i) and b(i), we can
   immediately construct the adjacency list for bond i, simply by merging
   all the auxiliary list for atoms a(i) and b(i). Therefore our first step
   is the construction of these auxiliary lists.

   ALGORITHM

   STEP 1: Construction of the auxiliary lists

   for each bond i do
     for each atom a in bond i do
       insert i into the ath auxiliary list
     end
   end

   COST: There are n bonds which involve 2 atoms each. Each auxiliary list
   will never be longer than K, and inserting an element into a sorted list
   of length L requires at most L comparisons, so the cost is less than 2nK
   comparisons.

   STEP 2: Construction of the adjacency lists

   for each bond i do
     for each atom a in bond i do
       for each bond j in the ath auxiliary list do
         insert j into the ith adjacency lists
       end
     end
   end

   COST: There are again n bonds which involve at 2 atoms each. Each
   auxiliary list has a length which is at most K. Each adjacency list will
   have a length which is at most 2K+1. Inserting M elements into a list of
   length at most L requires less than LM comparisons, so in total we do
   O(nK^2) comparisons.

   STEPS 3: Compress the adjacency lists into the array ADJ and create XADJ.
   There are n adjacency lists of length at most 2K+1, so the cost is O(nK).

*/


// ------------------------------------------------------------------------
// Declaration of internal variables
// ------------------------------------------------------------------------

// The number of atoms
int m;

// The number of bonds
int n;
```

```
// The list of bonds
int *bonds;

// The bond graph
graph_t *graph;

// The current bond is between atoms a and b.
int a,b;

// An auxilary variable used to generate XADJ
int temp;

// Variables needed to manipulate linked lists
struct my_node *conductor;
struct my_node **aux, **root;

// ------------------------------------------------------------------------
// Start of instructions
// ------------------------------------------------------------------------

// Extract the number of atoms (m) and the number of bonds
m=mol->m; n=mol->n;

// Establish shortcut to the list of bonds
bonds=mol->bonds;

// Allocate space for the bond graph
graph=(graph_t *)malloc(sizeof(graph_t));

// allocate space for the auxilary lists
aux=(struct my_node **)malloc(m*sizeof(struct my_node*));

/* Initialize the auxiliary lists. The ith auxiliary list will record the
   bonds which atom i partakes in. */

for(int i=0; i<m; i++) {
  aux[i]=NULL;
}

// Loop over the sequential list of bonds
for(int j=0; j<n; j++) {
  // Isolate the numbers of the atoms which partake in bond j.
  a=bonds[2*j]; b=bonds[2*j+1];
  // Insert bond j into the ath auxiliary list
  insert(j,&aux[a]);
  // Insert bond j into the bth auxiliary list
  insert(j,&aux[b]);
}

/* This completes the construction of the auxiliary list. For each atom we
```

```
   now have a list of the bonds it partakes in! */

// Allocate space for n adjacency lists
root=(struct my_node **)malloc(n*sizeof(struct my_node*));

// Initialize the adjacency lists
for(int j=0; j<n; j++) {
  root[j]=NULL;
}

// Loop over the sequential list of bonds
for(int j=0; j<n; j++) {
  // Isolate the numbers of the atoms which partake in bond j.
  a=bonds[2*j]; b=bonds[2*j+1];
  /* Insert every element of the a'th auxiliary list into the adjacency
     list for bond j. */
  conductor=aux[a];
  while (conductor!=NULL) {
    insert(conductor->number,&root[j]); conductor=conductor->next;
  }
  /* Insert every element of the b'th auxiliary list into the adjacency
     list for bond j */
  conductor=aux[b];
  while (conductor!=NULL) {
    insert(conductor->number,&root[j]); conductor=conductor->next;
  }
}

// Free the memory used by the auxilliary lists
for(int i=0; i<m; i++) {
  free_list(aux[i]);
}
// Do not forget to release aux itself.
free(aux);

/* This complete the construction of the adjacency lists as simply linked
   list. It remains to compress the information into two arrays XADJ and
   ADJ. */

// Allocate space for the array of indices into the combined adjacency list
graph->xadj=(int *)malloc((n+1)*sizeof(int));

// Initialize the counters
temp=0; graph->xadj[0]=temp;

// Find the length of the adjacency lists and compute the entries of XADJ
for(int j=0; j<n; j++) {
  temp=temp+count(root[j]); graph->xadj[j+1]=temp;
}

// Allocate space for the graph
```

```
  graph->adj=(int *)malloc(temp*sizeof(int));

  // Copy the lists into graph->adj and free the memory used.
  for(int j=0; j<n; j++) {
    copy_list(root[j],&graph->adj[graph->xadj[j]]); free_list(root[j]);
  }
  // Do not forget to release root itself
  free(root);

  // Set the number of vertices in the bond graph
  graph->m=n;

  // Do NOT forget to save the result of your work into MOL!
  mol->bond_graph=graph;
}

//-----------------------------------------------------------------------------
void renumber_bonds(molecule_t *mol, int *p) {
  /* Renumber the bonds (list/graph) using a given permutation.

     There is a important point to consider here:

     Do we or do we not allow subroutines to change pointer values when we
     are simply updating the information which they target?

     Here it would have been easy to save a 2*n memory operations by simply
     NOT copying the bond list into the auxiliary list and simply changing
     the value of the pointer mol->bond_graph!

     However, CCKM finds that it is far safer not to change this pointer.
     Suppose the user had created another pointer to the bond graph before
     calling renumber_bonds. This pointer **might** be invalid on the return
     from renumber_bonds.

     Tentative guiding principles:

       1) If the target size is unchanged, then preserve the pointer
       2) If the target size might change, then do not preserve the pointer.

     WARNING: There is only limited check of the sanity of the input.


   */

  // ---------------------------------------------------------------------------
  // Declaration of internal variables
  // ---------------------------------------------------------------------------

  // Number of bonds
  int n;
```

```
  // Auxiliary bond list
  int *aux;

  // ------------------------------------------------------------------------
  // Start of instructions
  // ------------------------------------------------------------------------

  if (mol->bonds!=NULL) {
     // Extract the number of bonds
    n=mol->n;

    // Allocate space for the bond auxiliary list
    aux=(int *)malloc(2*n*sizeof(int));

    // Copy the original bond list into the auxiliary list
    for(int i=0; i<2*n; i++) aux[i]=mol->bonds[i];

    // Apply the permutation to the bond list
    for(int i=0; i<n; i++) {
      mol->bonds[2*i+0]=aux[2*p[i]+0];
      mol->bonds[2*i+1]=aux[2*p[i]+1];
    }

    // Free the auxiliary bond list
    free(aux);
  }
  if (mol->bond_graph!=NULL) {
    // Apply the permutation to the bond graph
    renumber_vertices(mol->bond_graph, p);
  }
}
//------------------------------------------------------------------------

void make_weights(molecule_t *mol, graph_t *graph)
{

  /* Precomputes the weights needed to generate the matrix A(x,y).

     Description of interface:

     ON ENTRY:

       mol->m       the number of atoms
       mol->invmass pointer to the inverse of the atomic masses
       mol->n       the number of bonds
       mol->bonds   a pointer to a sequential list of n bonds
       graph        a pointer to a compact representation of the graph
                    to use, typically the bond graph or the lower triangular
                    portion of it

     ON EXIT:
```

```
  mol->weights a pointer to list of weights compatible with the graph

No other variables have been modified.

REMARK(S):

 1) At this point we do not take fill into account. Fill is identified
    by playing the elimination game on the bond graph. Right now, I am
    simply not sure where this code should go.


 Description of the construction of the weights:

 The following example explains the origins of the weights and how to
 compute them.

 EXAMPLE:

 A molecule with m = 6 atoms and n = 5 bonds. The atoms are numbered 0
 through 5, and the bonds are numbered 0 through 4. It is irrelevant if
 this molecule exists in the real world or not :)


 0               4          The bonds are given by the array
  \             /
  (0)          (3)          bonds = {0,2; 1,2; 2,3; 3,4; 3,5}
    \          /
     2 ------- 3            Each bond involves 2 atoms. Bond 2 is
    /    (2)   \            between atoms 2 and 3.
  (1)          (4)
  /             \
1               5

 The adjacency graph for the bonds is the graph


   0   3            matrix A is 5 by 5
  / \ / \
 |   2   |          |xxx  |
  \ / \ /           |xxx  |
   1   4            |xxxxx|
                    |  xxx|
                    |  xxx|

 This is how the graph is encoded

  adj  = {0,1,2; 0,1,2; 0,1,2,3,4; 2,3,4; 2,3,4} (17 entries)
 xadj  = {0, 3, 6, 11, 14, 17}                   ( 6 entries)

 Please note that there are n+1 entries in XADJ, and the last entry point
 just beyond the end of adj. Therefore XADJ[n+1] is the number of nonzero
```

```
entries in the matrix.

NOTATION:

  1) We write rij or r(i,j) for the vector from atom i to j.
  2) We write <x,y> for the scalar product between the vectors x and y.

In the above example bond number 2 involves atoms 2 and 3 and is a
(mathematical) constraint of the type

          0.5*(||r23||^2 - (bond length)^2) = 0

The factor 0.5 is only included to give the Jacobian of the constraint
function a form which I (CCKM) consider aesthetically pleasing.

Now, our matrix of the form

          A =  Dg(r)*inv(M)*Dg(s)'

where r and s are vectors describing two different configurations of
the m atoms, so r and s each have 3m components each.

Below is a table of the nonzero entries of the matrix A for our current
example:

           bond = {0,2,1,2,2,3,3,4,3,5}

i    j     entry A(i,j)
-----------------------------------------------------------------
0    0    (invmass(0)+invmass(2))*<r02,s03>
1    0    +invmass(2)*<r12,s02>
2    0    -invmass(2)*<r23,s02>

0    1    +invmass(2)*<r02,s12>
1    1    (invmass(1)+invmass(2))*<r12,s12>
2    1    -invmass(2)*<r23,s12>

0    2    -invmass(2)*<r02,s23>
1    2    -invmass(2)*<r12,s23>
2    2    (invmass(2)+invmass(3))*<r23,s23>
3    2    -invmass(3)*<r34,s23>
4    2    -invmass(3)*<r35,s23>

2    3    -invmass(3)*<r23,s34>
3    3    (invmass(3)+invmass(4))*<r34,s34>
4    3    +invmass(3)*<r35,s34>

2    4    -invmass(3)*<r23,s35>
3    4    +invmass(3)*<r34,s35>
4    4    (+invmass(3)+invmass(5))*<r35,s35>
```

This table is very carefully constructed! Please note the following:

a) Reading the table from the top to the bottom takes us through the nonzero entries of A in column major format, exactly as matrices are stored in LAPACK. Moreover, we are writing to main memory in an order which is strictly increasing.

b) For each column of the matrix A we deal with a FIXED vector s(a,b), rather than both s(a,b) and s(b,a) = -s(a,b).

c) The order of the indices as in r(a,b) for a bond k, is EXACTLY the order in which the atoms which partake in bond k are given in the bond table.

EXAMPLE: As noted above bond 2 is a bond between atoms 2 and 3. The bond table lists atom 2 BEFORE atom 3. Therefore, we use the vector r23, rather than the (equvalent) vector r32 = -r23

d) The diagonal entries are different from the off diagonal entries because the weights are different.

In order to efficiently generate the matrix A we precompute the following weights

i	j	weight
0	0	+invmass(0)+invmass(2)
1	0	+invmass(2)
2	0	-invmass(2)
0	1	+invmass(2)
1	1	+invmass(1)+invmass(2)
2	1	-invmass(2)
0	2	-invmass(2)
1	2	-invmass(2)
2	2	+invmass(2)+invmass(3)
3	2	-invmass(3)
4	2	-invmass(3)
2	3	-invmass(3)
3	3	+invmass(3)+invmass(4)
4	3	+invmass(3)
2	4	-invmass(3)
3	4	+invmass(3)
4	4	+invmass(3)+invmass(5)

In short, the weights includes all the relevant information about the signs as well as the masses of the atoms.

```
   Given vectors r and s with a 3m components as well as the bond graph, we
   can now generate the matrix A one entry at a time, moving through RAM
   memory in a strictly increasing order.

*/

// ------------------------------------------------------------------------
// Declaration of internal variables
// ------------------------------------------------------------------------

// The number of bonds;
int n;

// Shortcut to the list of inverse masses
real *invmass;

// Shortcut to the list of bonds
int *bonds;

// Shortcut to the list of weights
real *weights;

// The number of nonzeros
int nnz;

// Atomic labels
int a, b, c, d, e;

// ------------------------------------------------------------------------
// Start of instructions
// ------------------------------------------------------------------------

// Extract the number of bonds
n = mol->n;

// Establish shortcut to the list of inverse masses.
invmass = mol->invmass;

 // Establish shortcut to the list of bonds.
bonds = mol->bonds;

// Extract the number of nonzero entries
nnz = graph->xadj[n];

// Allocate enough space for nnz weights
weights = (real *)malloc(nnz*sizeof(*weights));

// Initialize the pointer to the nonzero entries
int k = 0;

// Loop over the bonds or equivalently the columns of the matrix
```

```
  for(int j = 0; j<n; j++) {
    // Isolate the atoms which partake in the jth bond
    a = bonds[2*j]; b = bonds[2*j+1];
    // Loop over the entries of the adjacency list of the jth bond.
    for(int i = graph->xadj[j]; i < graph->xadj[j+1]; i++)
    {
      if (j != graph->adj[i])
      {
        // This is an off diagonal diagonal entry.
        // We begin by isolating the atoms which partake in the ith bond
        c = bonds[2*graph->adj[i]]; d = bonds[2*(graph->adj[i])+1];
        // Determine the atom which is common to bond j and bond i
        if ((a == c) || (a == d)) {
          // The common atom is atom a
          e = a;
        }
        else
        {
          // The common atom must necessarily be atom b.
          e = b;
        }
        weights[k] = invmass[e];
        // Determine the appropriate sign
        if ((a == d) || (b == c))
        {
          /* You should reverse the order the atoms for one of the two bonds,
             but this impractical, so we just reverse the sign of the weight
          */
          weights[k] = -weights[k];
        }
      }
      else
      {
        /* This is a diagonal entry.
           The weight is special, but the order of the atoms in the bond list
           is irrelevant. Yes, you could flip the order of one pair of atoms,
           but then you would be compelled to flip the order of the second
           pair, and so you would change sign twice.
        */
        weights[k] = invmass[a] + invmass[b];
      }
      // Move on to the next nonzero entry of the matrix
      k++;
    }
  }
  // Remember to save the results of your work!
  mol->weights = weights;
}


//---------------------------------------------------------------------------
```

```
void free_weights(molecule_t *mol)
{
  /* Release the memory allocated by make_weights */
  free(mol->weights);
}
//-------------------------------------------------------------------------

// Allocate space for the displacement vector
real **alloc_displacements(molecule_t *mol)
{
   int nBonds = mol->n;
   real **displacements = (real **)malloc(3*sizeof(real *));
   displacements[0] = (real *)malloc(nBonds*sizeof(real));
   displacements[1] = (real *)malloc(nBonds*sizeof(real));
   displacements[2] = (real *)malloc(nBonds*sizeof(real));
   return displacements;
}

// Free space for the displacement vector
void free_displacements(real **displacements)
{
   free(displacements[0]);
   free(displacements[1]);
   free(displacements[2]);
   free(displacements);
}

//-------------------------------------------------------------------------
// Compute the displacement vector
void compute_displacements(molecule_t *mol, real *x, real **displacements)
{
   int c, d;
   int nBonds = mol->n;
   int *bonds = mol->bonds;

   for(int i = 0; i < nBonds; i++)
   {
      // Which atoms are involved in bond j?
      c = bonds[2*i]; d = bonds[2*i+1];

      // Compute vector from atom c to atom d
      displacements[0][i] = x[3*d+0] - x[3*c+0];
      displacements[1][i] = x[3*d+1] - x[3*c+1];
      displacements[2][i] = x[3*d+2] - x[3*c+2];
   }
}

//-------------------------------------------------------------------------
void make_matrix(molecule_t *mol, real *x, matrix_t *matrix)
{
  /* Constructs the matrix A given by
```

```
            A = A(x,y) = Dg(x)*inv(M)*Dg(y)'

   where g is the function which imposes the given bond length constraints.
   The construction is aided by using the table with information about the
   nonzero entries.

   ON ENTRY:

   mol->m        the number of atoms
   mol->n        the number of bonds
   mol->bonds    a sequential list of n bonds
   mol->weights  the weights needed compute the matrix rapidly

   matrix        a structure which must be unchanged since a call to
                 INIT_MATRIX

   ON EXIT:

   The entries of the matrix have been computed.
   No other changes are made.

*/

// Alignment hints
x = __builtin_assume_aligned(x, AVX_ALIGNMENT, 0);

// ------------------------------------------------------------------------
// Declaration of internal variables
// ------------------------------------------------------------------------

// The number of bonds n
int n;

// Atomic indices. Bond i is from a to b, bond j is from c to d.
int a, b, c, d;

// Vector from atom a to atom b (drawn from vector x)
real rab[3];

// Vector from atom c to atom d (drawn from vector y)
real scd[3];

// Scalar holds the inner product between rab and scd;
real scalar;

// Shortcut to the list of bonds
int *bonds;

// Short cut to the list of weights
real *weights;
```

```
// Shortcut to the matrix format
int format;

// ------------------------------------------------------------------------
// Start of instructions
// ------------------------------------------------------------------------

// Extract the number of bonds
n = mol->n;

// Establish shortcut to the list of bonds
bonds = mol->bonds;

// Establish shortcut to the list of weights
weights = mol->weights;

// Extract the desired format
format = matrix->format;

if ((format == COO) || (format == CSC))
{
  /* COO (Coordinate) or CSC (compressed sparse column)
     A previous call to INIT_MATRIX must have allocated space for VAL and
     recorded the sparsity pattern using arrays ICN and JCN in the case COO
     format and arrays XADJ and ADJ in the case of CSC format. Regardless
     of format the values of the matrix are entered into the array VAL. */

  // Loop over the columns of the matrix
  for(int j = 0; j < n; j++)
  {
    // Which atoms are involved in bond j?
    c = bonds[2*j]; d = bonds[2*j+1];

    // Compute vector from atom c to atom d (using y), s(c,d) = s_d-s_c
    scd[0] = x[3*d+0] - x[3*c+0];
    scd[1] = x[3*d+1] - x[3*c+1];
    scd[2] = x[3*d+2] - x[3*c+2];

    // Loop over the nonzeros in the jth column
    for(int k = matrix->xadj[j]; k < matrix->xadj[j+1]; k++) {
      // Determine the row index of the kth entry
      int i = matrix->adj[k];

      // Which atoms are involved in bond i?
      a = bonds[2*i]; b = bonds[2*i+1];

      // Compute vector from atom a to atom b (using x), r(a,b) = r_b-r_a
      rab[0] = x[3*b+0] - x[3*a+0];
      rab[1] = x[3*b+1] - x[3*a+1];
      rab[2] = x[3*b+2] - x[3*a+2];
```

```
        // Compute the inner product scalar = <r_{a,b},s_{c,d}>
        scalar = rab[0]*scd[0] + rab[1]*scd[1] + rab[2]*scd[2];

        // Fill in the appropriate entry into the matrix
        matrix->val[k] = weights[k]*scalar;
      }
    }
  }
  else
  {
    if ((format == DEN) || (format == BAN) || (format == CSR)) {
      // Dense, banded or compressed sparse row
      printf("MD-SPARSE.C: MAKE_MATRIX: Format not implemented\n");
    }
    else
    {
      // Invalid format
      printf("MD-SPARSE: MAKE_MATRIX: Invalid format\n");
    }
  }
  // End of subroutine
}

//---------------------------------------------------------------------------
void make_matrix_opt(molecule_t *mol, matrix_t *matrix, real **displacements)
{
  /* Constructs the matrix A given by

                A = A(x,y) = Dg(x)*inv(M)*Dg(y)'

     where g is the function which imposes the given bond length constraints.
     The construction is aided by using the table with information about the
     nonzero entries.

     ON ENTRY:

     mol->m        the number of atoms
     mol->n        the number of bonds
     mol->bonds    a sequential list of n bonds
     mol->weights  the weights needed compute the matrix rapidly
       displacements distance of the atoms of the bonds

     matrix        a structure which must be unchanged since a call to
                   INIT_MATRIX

     ON EXIT:

     The entries of the matrix have been computed.
     No other changes are made.
```

```
*/

// ------------------------------------------------------------------------
// Declaration of internal variables
// ------------------------------------------------------------------------

// The number of bonds n
int n;

// Scalar holds the inner product between rab and scd;
real scalar;

// Short cut to the list of weights
real *weights;

// Shortcut to the matrix format
int format;

// ------------------------------------------------------------------------
// Start of instructions
// ------------------------------------------------------------------------

// Extract the number of bonds
n = mol->n;

// Establish shortcut to the list of weights
weights = mol->weights;

// Extract the desired format
format = matrix->format;

if ((format == COO) || (format == CSC))
{
  /* COO (Coordinate) or CSC (compressed sparse column)
     A previous call to INIT_MATRIX must have allocated space for VAL and
     recorded the sparsity pattern using arrays ICN and JCN in the case COO
     format and arrays XADJ and ADJ in the case of CSC format. Regardless
     of format the values of the matrix are entered into the array VAL. */

  // Loop over the columns of the matrix
  for(int j = 0; j < n; j++)
  {

    // Loop over the nonzeros in the jth column
    for(int k = matrix->xadj[j]; k < matrix->xadj[j+1]; k++) {
      // Determine the row index of the kth entry
      int i = matrix->adj[k];

      // Compute the inner product scalar = <r_{a,b},s_{c,d}>
      scalar = displacements[0][i]*displacements[0][j] + displacements[1][i]*displacemen
```

```
        // Fill in the appropriate entry into the matrix
        matrix->val[k] = weights[k]*scalar;
      }
    }
  }
  else
  {
    if ((format == DEN) || (format == BAN) || (format == CSR)) {
      // Dense, banded or compressed sparse row
      printf("MD-SPARSE.C: MAKE_MATRIX: Format not implemented\n");
    }
    else
    {
      // Invalid format
      printf("MD-SPARSE: MAKE_MATRIX: Invalid format\n");
    }
  }
  // End of subroutine
}

//-----------------------------------------------------------------------------
void make_matrix_vec(molecule_t *mol, real *x, matrix_t *matrix,
     real *valVec, int *xadjVec, real *fillLeft, real *fillRight,
     int block_size, int offset, int isFirst, int isLast)
{
  // Alignment hints
  x = __builtin_assume_aligned(x, AVX_ALIGNMENT, 0);

  int a, b, c, d;
  real rab[3];
  real scd[3];
  real scalar;
  int *bonds;
  real *weights;
  int format;

  bonds = mol->bonds;
  weights = mol->weights;
  format = matrix->format;

  if ((format == COO) || (format == CSC))
  {
      valVec[xadjVec[0]] = 0;
      valVec[xadjVec[block_size-1]+2*VLEN+(VLEN-1)] = 0;

      for(int i = 0; i < VLEN*block_size; i++)
    {
        fillLeft[i] = 0;
        fillRight[i] = 0;
      }
```

```
  if (!isFirst)
{
    int j = offset - 1;
    // Which atoms are involved in bond j?
    c = bonds[2*j]; d = bonds[2*j+1];

    // Compute vector from atom c to atom d (using y), s(c,d) = s_d-s_c
    scd[0] = x[3*d+0] - x[3*c+0];
    scd[1] = x[3*d+1] - x[3*c+1];
    scd[2] = x[3*d+2] - x[3*c+2];

    // Left fill-in
    int k = matrix->xadj[j]+2;

    int i = matrix->adj[k];

    // Which atoms are involved in bond i?
    a = bonds[2*i]; b = bonds[2*i+1];

    // Compute vector from atom a to atom b (using x), r(a,b) = r_b - r_a
    rab[0] = x[3*b+0]-x[3*a+0];
    rab[1] = x[3*b+1]-x[3*a+1];
    rab[2] = x[3*b+2]-x[3*a+2];

    // Compute the inner product scalar = <r_{a,b},s_{c,d}>
    scalar = rab[0]*scd[0] + rab[1]*scd[1] + rab[2]*scd[2];

    // Fill in the appropriate entry into the matrix
    fillLeft[0] = weights[k]*scalar;
  }
  for (int l=1; l < VLEN; l++)
{
    int j = l*(block_size+1) - 1 + offset;
    // Which atoms are involved in bond j?
    c = bonds[2*j]; d = bonds[2*j+1];

    // Compute vector from atom c to atom d (using y), s(c,d) = s_d-s_c
    scd[0] = x[3*d+0] - x[3*c+0];
    scd[1] = x[3*d+1] - x[3*c+1];
    scd[2] = x[3*d+2] - x[3*c+2];

    // Left fill-in
    int k = matrix->xadj[j] + 2;

    int i=matrix->adj[k];

    // Which atoms are involved in bond i?
    a = bonds[2*i]; b = bonds[2*i+1];

    // Compute vector from atom a to atom b (using x), r(a,b) = r_b-r_a
    rab[0] = x[3*b+0] - x[3*a+0];
```

```
      rab[1] = x[3*b+1] - x[3*a+1];
      rab[2] = x[3*b+2] - x[3*a+2];

      // Compute the inner product scalar = <r_{a,b},s_{c,d}>
      scalar = rab[0]*scd[0] + rab[1]*scd[1] + rab[2]*scd[2];

      // Fill in the appropriate entry into the matrix
      fillLeft[l] = weights[k]*scalar;

      // Right fill-in
      k = matrix->xadj[j];

      i = matrix->adj[k];

    // Which atoms are involved in bond i?
    a = bonds[2*i]; b = bonds[2*i+1];

    // Compute vector from atom a to atom b (using x), r(a,b) = r_b-r_a
    rab[0] = x[3*b+0] - x[3*a+0];
    rab[1] = x[3*b+1] - x[3*a+1];
    rab[2] = x[3*b+2] - x[3*a+2];

    // Compute the inner product scalar = <r_{a,b},s_{c,d}>
    scalar = rab[0]*scd[0] + rab[1]*scd[1] + rab[2]*scd[2];

    // Fill in the appropriate entry into the matrix
      fillRight[(block_size-1)*VLEN+l-1] = weights[k]*scalar;
    }
    if(!isLast)
  {
      int j = VLEN*(block_size+1) - 1 + offset;
      // Which atoms are involved in bond j?
      c = bonds[2*j]; d = bonds[2*j+1];

      // Compute vector from atom c to atom d (using y), s(c,d) = s_d-s_c
      scd[0] = x[3*d+0] - x[3*c+0];
      scd[1] = x[3*d+1] - x[3*c+1];
      scd[2] = x[3*d+2] - x[3*c+2];

      // Right fill-in
      int k = matrix->xadj[j];

      int i = matrix->adj[k];

    // Which atoms are involved in bond i?
    a = bonds[2*i]; b = bonds[2*i+1];

    // Compute vector from atom a to atom b (using x), r(a,b) = r_b-r_a
    rab[0] = x[3*b+0] - x[3*a+0];
    rab[1] = x[3*b+1] - x[3*a+1];
    rab[2] = x[3*b+2] - x[3*a+2];
```

```
  // Compute the inner product scalar = <r_{a,b},s_{c,d}>
  scalar = rab[0]*scd[0] + rab[1]*scd[1] + rab[2]*scd[2];

  // Fill in the appropriate entry into the matrix
     fillRight[(block_size-1)*VLEN+VLEN-1] = weights[k]*scalar;
  }

  for (int l=0; l < VLEN; l++)
{
  for (int m=0; m < block_size; m++)
  {
       int j = l*(block_size+1) + m + offset;
     // Which atoms are involved in bond j?
     c = bonds[2*j]; d = bonds[2*j+1];

     // Compute vector from atom c to atom d (using y), s(c,d) = s_d-s_c
     scd[0] = x[3*d+0]-x[3*c+0];
     scd[1] = x[3*d+1]-x[3*c+1];
     scd[2] = x[3*d+2]-x[3*c+2];

     // Loop over the nonzeros in the jth column
     for(int k = matrix->xadj[j]; k < matrix->xadj[j+1]; k++)
    {
       // Determine the row index of the kth entry
       int i = matrix->adj[k];

       // Which atoms are involved in bond i?
       a = bonds[2*i]; b = bonds[2*i+1];

       // Compute vector from atom a to atom b (using x), r(a,b) = r_b-r_a
       rab[0] = x[3*b+0] - x[3*a+0];
       rab[1] = x[3*b+1] - x[3*a+1];
       rab[2] = x[3*b+2] - x[3*a+2];

       // Compute the inner product scalar = <r_{a,b},s_{c,d}>
       scalar = rab[0]*scd[0] + rab[1]*scd[1] + rab[2]*scd[2];

           valVec[xadjVec[m]+(i-j+1)*VLEN+l] = weights[k]*scalar;

           /*if((j+1)%(block_size+1)==0){// Schur complement column
              if(i==j){// Schur complement
                 schurMatrix[(j+1)/(block_size+1)][1]=weights[k]*scalar;
              }else if(i<j){// Right fill-in
                 fillRight[(block_size-1)*VLEN+((j+1)/(block_size+1)-1)]=weights[k]*sca
              }else{// Left fill-in
                 fillLeft[(j+1)/(block_size+1)]=weights[k]*scalar;
              }
           }else{// Rest of the matrix
              valVec[xadjVec[j%(block_size+1)]+(i-(j-1))*VLEN+j/(block_size+1)]=weights
           }*/
```

```
        }
        }
    }
  }
  else
  {
    if ((format == DEN) || (format == BAN) || (format == CSR))
    {
      // Dense, banded or compressed sparse row
      printf("MD-SPARSE.C: MAKE_MATRIX: Format not implemented\n");
    } else {
      // Invalid format
      printf("MD-SPARSE: MAKE_MATRIX: Invalid format\n");
    }
  }
  // End of subroutine
}

//---------------------------------------------------------------------------
void make_matrix_vec_opt(molecule_t *mol, matrix_t *matrix,
     real *valVec, int *xadjVec, real *fillLeft, real *fillRight,
     int block_size, int offset, int isFirst, int isLast, real **displacements)
{
  real scalar;
  real *weights;
  int format;

  weights = mol->weights;
  format = matrix->format;

  if ((format == COO) || (format == CSC))
  {
      valVec[xadjVec[0]] = 0;
      valVec[xadjVec[block_size-1]+2*VLEN+(VLEN-1)] = 0;

      for(int i = 0; i < VLEN*block_size; i++)
    {
        fillLeft[i] = 0;
        fillRight[i] = 0;
      }

      if (!isFirst)
    {
        int j = offset - 1;

        // Left fill-in
        int k = matrix->xadj[j]+2;

        int i = matrix->adj[k];

        // Compute the inner product scalar = <r_{a,b},s_{c,d}>
```

```
    scalar = displacements[0][i]*displacements[0][j] + displacements[1][i]*displaceme

    // Fill in the appropriate entry into the matrix
    fillLeft[0] = weights[k]*scalar;
  }
  for (int l=1; l < VLEN; l++)
{
    int j = l*(block_size+1) - 1 + offset;

    // Left fill-in
    int k = matrix->xadj[j] + 2;

    int i=matrix->adj[k];

    // Compute the inner product scalar = <r_{a,b},s_{c,d}>
    scalar = displacements[0][i]*displacements[0][j] + displacements[1][i]*displaceme

    // Fill in the appropriate entry into the matrix
    fillLeft[l] = weights[k]*scalar;

    // Right fill-in
    k = matrix->xadj[j];

    i = matrix->adj[k];

  // Compute the inner product scalar = <r_{a,b},s_{c,d}>
  scalar = displacements[0][i]*displacements[0][j] + displacements[1][i]*displacements

  // Fill in the appropriate entry into the matrix
    fillRight[(block_size-1)*VLEN+l-1] = weights[k]*scalar;
  }
  if(!isLast)
{
    int j = VLEN*(block_size+1) - 1 + offset;

    // Right fill-in
    int k = matrix->xadj[j];

    int i = matrix->adj[k];

  // Compute the inner product scalar = <r_{a,b},s_{c,d}>
  scalar = displacements[0][i]*displacements[0][j] + displacements[1][i]*displacements

  // Fill in the appropriate entry into the matrix
    fillRight[(block_size-1)*VLEN+VLEN-1] = weights[k]*scalar;
  }

  for (int l=0; l < VLEN; l++)
{
  for (int m=0; m < block_size; m++)
  {
```

```
           int j = l*(block_size+1) + m + offset;

         // Loop over the nonzeros in the jth column
         for(int k = matrix->xadj[j]; k < matrix->xadj[j+1]; k++)
        {
           // Determine the row index of the kth entry
           int i = matrix->adj[k];

           // Compute the inner product scalar = <r_{a,b},s_{c,d}>
           scalar = displacements[0][i]*displacements[0][j] + displacements[1][i]*displacer

               valVec[xadjVec[m]+(i-j+1)*VLEN+l] = weights[k]*scalar;
         }
         }
    }
  }
  else
  {
    if ((format == DEN) || (format == BAN) || (format == CSR))
    {
      // Dense, banded or compressed sparse row
      printf("MD-SPARSE.C: MAKE_MATRIX: Format not implemented\n");
    } else {
      // Invalid format
      printf("MD-SPARSE: MAKE_MATRIX: Invalid format\n");
    }
  }
  // End of subroutine
}

//-----------------------------------------------------------------------------
void make_matrix_Svec_opt(molecule_t *mol, matrix_t *matrix, int *xadjVec,
                    real schurMatrix[][3], int block_size, int schur_size, real **displace
{
  // ---------------------------------------------------------------------
  // Declaration of internal variables
  // ---------------------------------------------------------------------

  real scalar;
  real *weights;
  int format;

  // ---------------------------------------------------------------------
  // Start of instructions
  // ---------------------------------------------------------------------

  weights = mol->weights;
  format = matrix->format;

  if ((format == COO) || (format == CSC))
  {
```

```
      for(int i=0;i<block_size+1;i++)
    {
        xadjVec[i] = i*3*VLEN;
      }

      for(int i = 0; i < schur_size; i++)
    {
        schurMatrix[i][0] = 0;
        schurMatrix[i][2] = 0;
      }
      schurMatrix[0][1] = 0;
      schurMatrix[schur_size][1] = 0;

    // Loop over the columns of the matrix
    for (int l = 1; l < schur_size; l++)
    {
        int j = l*(block_size+1) - 1;

        int k = matrix->xadj[j]+1;
      // Determine the row index of the kth entry
      int i = matrix->adj[k];

      // Compute the inner product scalar = <r_{a,b},s_{c,d}>
      scalar = displacements[0][i]*displacements[0][j] + displacements[1][i]*displacements

      // Fill in the appropriate entry into the matrix
        schurMatrix[l][1] = weights[k]*scalar;
    }
  }
  else
  {
    if ((format == DEN) || (format == BAN) || (format == CSR))
    {
      // Dense, banded or compressed sparse row
      printf("MD-SPARSE.C: MAKE_MATRIX: Format not implemented\n");
    }
    else
    {
      // Invalid format
      printf("MD-SPARSE: MAKE_MATRIX: Invalid format\n");
    }
  }
  // End of subroutine
}

//---------------------------------------------------------------------------
void make_matrix_Svec(molecule_t *mol, real *x, matrix_t *matrix, int *xadjVec,
                      real schurMatrix[][3], int block_size, int schur_size)
{
  // Alignment hints
  x = __builtin_assume_aligned(x, AVX_ALIGNMENT, 0);
```

```
// ------------------------------------------------------------------------
// Declaration of internal variables
// ------------------------------------------------------------------------

int a, b, c, d;
real rab[3];
real scd[3];
real scalar;
int *bonds;
real *weights;
int format;

// ------------------------------------------------------------------------
// Start of instructions
// ------------------------------------------------------------------------

bonds = mol->bonds;
weights = mol->weights;
format = matrix->format;

if ((format == COO) || (format == CSC))
{
    for(int i=0;i<block_size+1;i++)
  {
      xadjVec[i] = i*3*VLEN;
    }

    for(int i = 0; i < schur_size; i++)
  {
      schurMatrix[i][0] = 0;
      schurMatrix[i][2] = 0;
    }
    schurMatrix[0][1] = 0;
    schurMatrix[schur_size][1] = 0;

  // Loop over the columns of the matrix
  for (int l = 1; l < schur_size; l++)
  {
      int j = l*(block_size+1) - 1;
    // Which atoms are involved in bond j?
    c = bonds[2*j]; d = bonds[2*j+1];

    // Compute vector from atom c to atom d (using y), s(c,d) = s_d-s_c
    scd[0] = x[3*d+0]-x[3*c+0];
    scd[1] = x[3*d+1]-x[3*c+1];
    scd[2] = x[3*d+2]-x[3*c+2];

      int k = matrix->xadj[j]+1;
    // Determine the row index of the kth entry
    int i = matrix->adj[k];
```

```
      // Which atoms are involved in bond i?
      a = bonds[2*i]; b = bonds[2*i+1];

      // Compute vector from atom a to atom b (using x), r(a,b) = r_b-r_a
      rab[0] = x[3*b+0] - x[3*a+0];
      rab[1] = x[3*b+1] - x[3*a+1];
      rab[2] = x[3*b+2] - x[3*a+2];

      // Compute the inner product scalar = <r_{a,b},s_{c,d}>
      scalar = rab[0]*scd[0] + rab[1]*scd[1] + rab[2]*scd[2];

      // Fill in the appropriate entry into the matrix
         schurMatrix[l][1] = weights[k]*scalar;
    }
  }
  else
  {
    if ((format == DEN) || (format == BAN) || (format == CSR))
    {
      // Dense, banded or compressed sparse row
      printf("MD-SPARSE.C: MAKE_MATRIX: Format not implemented\n");
    }
    else
    {
      // Invalid format
      printf("MD-SPARSE: MAKE_MATRIX: Invalid format\n");
    }
  }
  // End of subroutine
}

//-----------------------------------------------------------------------------
void init_jacobian(molecule_t *mol, int format, matrix_t *Dg)
{
  /* Allocates enough memory to hold the Jacobian Dg() in various formats

     DESCRIPTION OF INTERFACE:

     ON ENTRY:
     mol      pointer to a datastructure descriping the molecule
     format   the requested matrix format for the Jacobian
     Dg       pointer to a datastructure which can hold a matrix
  */

  // ---------------------------------------------------------------------
  // Declaration of internal variables
  // ---------------------------------------------------------------------

  // The number of atoms/bonds
  int m, n;
```

```
// A flag used to test the format specification
int flag;

// The number of nnz in the matrix
int nnz;

// Atomic indices
int a, b;

// ------------------------------------------------------------------------
// Start of instructions
// ------------------------------------------------------------------------

// Extract the number of atoms, bonds
m = mol->m; n = mol->n;

// Initialize the format flag.
flag=1;

// Test if the format specification is acceptable.
switch(format)
{
  case DEN: // Dense format
    // Allocate space for an n by 3m dense matrix!
    Dg->val=(real *)malloc(n*3*m*sizeof(real));
    // Specify the leading dimension of the array for the sake of LAPACK
    Dg->lda=n;
    break;
  case BAN: // Banded format
    printf("MD-SPARSE: INIT_JACOBIAN: Matrix format not supported.\n");
    flag=0;
    break;
  case COO: // Coordianate format
    printf("MD-SPARSE: INIT_JACOBIAN: Matrix format not supported.\n");
    flag=0;
    break;
  case CSR: // Compressed sparse row
    // Compute nnz, 6 elements pr. row for bond constraints
    nnz=6*n;
    // Set the nnz for the matrix
    Dg->nnz=6*n;
    // Allocate space for the row-pointers
    Dg->xadj=(int *)malloc((n+1)*sizeof(int));
    // Allocate space for the column indices
    Dg->adj=(int *)malloc(nnz*sizeof(int));
    // Allocate space for the values of the matrix
    Dg->val=(real *)malloc(nnz*sizeof(real));

    // The sparsity pattern is fixed and will be defined below
```

```
      // Initialize the index pointer into Dg->adj
      int k=0; Dg->xadj[0]=0;

      // Loop over the n rows
      for(int i=0; i<n; i++)
      {
        // Isolate the atoms which partake in bond a and b
        a=mol->bonds[2*i]; b=mol->bonds[2*i+1];
        // Fill in the column indices corresponding to atom b
        Dg->adj[k]=3*b+XX; k++;
        Dg->adj[k]=3*b+YY; k++;
        Dg->adj[k]=3*b+ZZ; k++;
        // Fill in the column indices corresponding to atom a
        Dg->adj[k]=3*a+XX; k++;
        Dg->adj[k]=3*a+YY; k++;
        Dg->adj[k]=3*a+ZZ; k++;
        // Set the pointer to the next row
        Dg->xadj[i+1]=k;
      }
      // Please note that Dg->xadj[n]=6*n at this point.
      break;
    case CSC: // Compressed sparse column
      printf("MD-SPARSE: INIT_JACOBIAN: Matrix format not supported.\n");
      flag=0;
      break;
    default:
      printf("MD-SPARSE: INIT_JACOBIAN: Invalid matrix format specified.\n");
      flag=0;
      break;
  }

  // Was the format acceptable?
  if (flag==1)
  {
    // Set the matrix format
    Dg->format = format;
    // Set the matrix dimension: n rows and 3m columns.
    Dg->m = n; Dg->n = 3*m;
  }

  // Normal return to caller
  return;
}
//---------------------------------------------------------------------------

void make_jacobian(molecule_t *mol, real *x, matrix_t *dg)
{
  /* Builds the Jacobian matrix Dg(x) explicitly

     DESCRIPTION OF INTERFACE
```

```
   ON ENTRY:
   mol      pointer to a datastructure describing the molecule
   x        pointer the real vector x
   a        pointer to a datastructure which can hold the matrix

   ON EXIT:
   mol     unchanged
   x       unchanged
   a       has been updated with an explicit representation of the
           Jacobian Dg(x)

   WARNING:
   A earlier call to INIT_JACOBIAN must have set the format and allocated
   memory for the matrix.
 */

// ------------------------------------------------------------------------
// Declaration of internal variables
// ------------------------------------------------------------------------

// The number of bonds n
int n;

// Atomic indices
int a, b, k;

// Three dimensional vector from atom a to atom b
real r[3];

// ------------------------------------------------------------------------
// Start of instructions
// ------------------------------------------------------------------------

// Extract the number of bonds
n = mol->n;

// Examine the format specification of the matrix
switch(dg->format)
{
  case DEN:
    printf("MD-SPARSE: MAKE_JACOBIAN: Format not supported\n");
    break;
  case BAN:
    printf("MD-SPARSE: MAKE_JACOBIAN: Format not supported\n");
    break;
  case COO:
    printf("MD-SPARSE: MAKE_JACOBIAN: Format not supported\n");
    break;
  case CSR:
    // Initialize the index pointer into dg->val
    k=0;
```

```
      // Loop over the rows
      for(int i=0; i<n; i++) {
        // Determine the atoms involved: atom a and atom b;
        a=mol->bonds[2*i]; b=mol->bonds[2*i+1];
        // Determine the vector from atom a to atom b
        r[0]=x[3*b+XX]-x[3*a+XX];
        r[1]=x[3*b+YY]-x[3*a+YY];
        r[2]=x[3*b+ZZ]-x[3*a+ZZ];
        // Fill in values for the three columns corresponding to atom b
        dg->val[k]=r[0]; k++;
        dg->val[k]=r[1]; k++;
        dg->val[k]=r[2]; k++;
        // Fill in the value for the three column correspoding to atom a
        dg->val[k]=-r[0]; k++;
        dg->val[k]=-r[1]; k++;
        dg->val[k]=-r[2]; k++;
      }
      break;
    case CSC:
      printf("MD-SPARSE: MAKE_JACOBIAN: Format not supported\n");
      break;
    default:
      printf("MD-SPARSE: MAKE_JACOBIAN: Format not supported\n");
      break;
  }
}
//-----------------------------------------------------------------------------

void apply_jacobian(char *trans, real alpha, molecule_t *mol,
                    real * restrict x, real * restrict y, real * restrict z)
{

  /* Compute z = z + alpha * op(Dg(x))*y, without forming Dg(x) explicitly

     I expect this to be faster than explicitly forming the Jacobian using
     MAKE_JACOBIAN and then applying the MATRIX_VECTOR routine from MATRIX.h.

     Use one routine to verify that the other routine is working correctly!

     DESCRIPTION OF INTERFACE

     ON ENTRY:
     trans       a pointer to a string which is either "N" or "T"
                   op(A) = A , if trans="N"
                   op(A) = A', if trans="T", (matrix transposition)
     alpha       a real constant
     mol->bonds  a pointer to a sequential list of bonds
     x, y, z     pointers to real vectors x, y, z

     ON EXIT:
     trans       unchanged
```

```
   alpha       unchanged
   mol->bonds  unchanged
   x, y        unchanged
   z           has been updated to z=z+alpha*op(Dg(x))*y

   WARNING:
   It is the responsibility of the caller to ensure that the operation is
   defined, i.e. that all objects have the correct number of rows/columns.
*/

// ------------------------------------------------------------------------
// Declaration of interface variables
// ------------------------------------------------------------------------

// Auxiliary variable for intermediate calculations
real aux;

// Vector from atom a to atom b
real r[3];

// Atomic indices
int a, b;

// The number of bonds
int n;

// ------------------------------------------------------------------------
// Start of instructions
// ------------------------------------------------------------------------

// Extract the number bonds n
n = mol->n;

// Examine the operator specification
if (strcmp(trans,"T") == 0)
{
  /* op(A)=A', (matrix transposition)

     We loop over the rows of Dg (the columns of Dg') and compute

       z = z+alpha*(ith row of Dg)*y[i]

     This accomplished the goal.
  */

  // Loop over the rows of Dg
  for(int i = 0; i < n; i++)
  {
    // Isolate the atoms involved in bond i
    a = mol->bonds[2*i]; b = mol->bonds[2*i+1];
    // Calculate the vector from atom a to atom b;
```

```
      r[0] = x[3*b+XX] - x[3*a+XX];
      r[1] = x[3*b+YY] - x[3*a+YY];
      r[2] = x[3*b+ZZ] - x[3*a+ZZ];

      // Update the components of z which correspond to atom a
      z[3*a+XX] = z[3*a+XX] - alpha*r[0]*y[i];
      z[3*a+YY] = z[3*a+YY] - alpha*r[1]*y[i];
      z[3*a+ZZ] = z[3*a+ZZ] - alpha*r[2]*y[i];
      // Update the components of z which correspond to atom b
      z[3*b+XX] = z[3*b+XX] + alpha*r[0]*y[i];
      z[3*b+YY] = z[3*b+YY] + alpha*r[1]*y[i];
      z[3*b+ZZ] = z[3*b+ZZ] + alpha*r[2]*y[i];
    }
  }
  else
  {
    if (strcmp(trans,"N") == 0)
    { // op(A) = A
      // Loop over the rows of Dg
      for(int i=0; i<n; i++)
      {
        // Determine the atoms a and b involved in the ith bond
        a = mol->bonds[2*i]; b = mol->bonds[2*i+1];
        // Calculate the vector from atom a to atom b.
        r[0] = x[3*b+XX] - x[3*a+XX];
        r[1] = x[3*b+YY] - x[3*a+YY];
        r[2] = x[3*b+ZZ] - x[3*a+ZZ];
        // Compute the ith component of Dg(x)*y
        aux = 0;
        aux = aux + r[0]*(y[3*b+XX] -y[3*a+XX]);
        aux = aux + r[1]*(y[3*b+YY] -y[3*a+YY]);
        aux = aux + r[2]*(y[3*b+ZZ] -y[3*a+ZZ]);
        // Update the ith component of z
        z[i] = z[i] + alpha*aux;
      }
    } else {
      // Invalid operator specificition
      printf("MD-SPARSE: APPLY_JACOBIAN: Invalid operator specification.\n");
    }
  }

  // End of subroutine
  return;
}
//-----------------------------------------------------------------------------

void apply_jacobian_Svec(real alpha, molecule_t *mol, real * restrict x,
                         real * restrict gSchurVec, real * restrict z,
                        int block_size, int schur_size)
{
  real r[3];
```

```
  int a, b;

  // Loop over the rows of Dg
  for (int j = 1; j < schur_size; j++)
  {
      int i = j*(block_size+1) - 1;
    // Isolate the atoms involved in bond i
    a = mol->bonds[2*i]; b = mol->bonds[2*i+1];
    // Calculate the vector from atom a to atom b;
    r[0] = x[3*b+XX] - x[3*a+XX];
    r[1] = x[3*b+YY] - x[3*a+YY];
    r[2] = x[3*b+ZZ] - x[3*a+ZZ];
    // Update the components of z which correspond to atom a
    z[3*a+XX] = z[3*a+XX] - alpha*r[0]*gSchurVec[j];
    z[3*a+YY] = z[3*a+YY] - alpha*r[1]*gSchurVec[j];
    z[3*a+ZZ] = z[3*a+ZZ] - alpha*r[2]*gSchurVec[j];

    // Update the components of z which correspond to atom b
    z[3*b+XX] = z[3*b+XX] + alpha*r[0]*gSchurVec[j];
    z[3*b+YY] = z[3*b+YY] + alpha*r[1]*gSchurVec[j];
    z[3*b+ZZ] = z[3*b+ZZ] + alpha*r[2]*gSchurVec[j];
  }
  return;
}
//-------------------------------------------------------------------------

void apply_jacobian_vec(real alpha, molecule_t *mol, real * restrict x,
                        real * restrict gVec, real * restrict z,
                       int block_size, int offset)
{
  real r[3];
  int a, b;

   for (int j=0; j < VLEN; j++)
   {
      for (int k = 0; k < block_size; k++)
      {
        int i = j*(block_size+1) + k + offset;
       // Isolate the atoms involved in bond i
       a = mol->bonds[2*i]; b = mol->bonds[2*i+1];
       // Calculate the vector from atom a to atom b;
       r[0] = x[3*b+XX] - x[3*a+XX];
       r[1] = x[3*b+YY] - x[3*a+YY];
       r[2] = x[3*b+ZZ] - x[3*a+ZZ];
       // Update the components of z which correspond to atom a
       z[3*a+XX] = z[3*a+XX] - alpha*r[0]*gVec[k*VLEN+j];
       z[3*a+YY] = z[3*a+YY] - alpha*r[1]*gVec[k*VLEN+j];
       z[3*a+ZZ] = z[3*a+ZZ] - alpha*r[2]*gVec[k*VLEN+j];

       // Update the components of z which correspond to atom b
       z[3*b+XX] = z[3*b+XX] + alpha*r[0]*gVec[k*VLEN+j];
```

```
      z[3*b+YY] = z[3*b+YY] + alpha*r[1]*gVec[k*VLEN+j];
      z[3*b+ZZ] = z[3*b+ZZ] + alpha*r[2]*gVec[k*VLEN+j];
    }
   }
  return;
}
//------------------------------------------------------------------------

void bajt(int n, int *xadj, int *adj, real * restrict val, real * restrict f,
         real alpha, molecule_t *mol, real * restrict x, real * restrict z)
{
  /* BAJT = BACKWARD, APPLY JACOBIAN TRANSPOSED.

     Does the rather specialized linear update

        z:=z + alpha *Dg(x)'*(inv(L')*f)

     by fusing the back sweep with the application of the Jacobian.
     This is fusion of subroutines BACKWARD from CHOL and APPLY_JACOBIAN
     from this module.

  */

  // ------------------------------------------------------------------------
  // Declaration of internal variables
  // ------------------------------------------------------------------------

  // Diagonal value from the matrix
  real diag;

  // Atomic indices
  int a, b;

  // Auxillary vector from atom a to atom b
  real r[3];

  // ------------------------------------------------------------------------
  // Start of instructions
  // ------------------------------------------------------------------------

  for(int i=n-1; i>=0; i--)
  {
    // Isolate the ith diagonal element
    int k = xadj[i]; diag = val[k];
    // Remove the contribution of the variables x[i+1], ..., x[n-1]
    for (int s=k+1; s < xadj[i+1]; s++)
    {
      f[i] = f[i] - val[s]*f[adj[s]];
    }
    // Divide with the diagonal element
    f[i] = f[i]/diag;
```

```
    /* At this point the final value for x[i] has been computed.
       We are now free to apply the ith component to the linear update */
    // Isolate the atoms involved in bond i
    a = mol->bonds[2*i]; b = mol->bonds[2*i+1];
    // Calculate the vector from atom a to atom b;
    r[0] = x[3*b+XX] - x[3*a+XX];
    r[1] = x[3*b+YY] - x[3*a+YY];
    r[2] = x[3*b+ZZ] - x[3*a+ZZ];
    // Update the components of z which correspond to atom a
    z[3*a+XX] = z[3*a+XX] - alpha*r[0]*f[i];
    z[3*a+YY] = z[3*a+YY] - alpha*r[1]*f[i];
    z[3*a+ZZ] = z[3*a+ZZ] - alpha*r[2]*f[i];
    // Update the components of z which correspond to atom b
    z[3*b+XX] = z[3*b+XX] + alpha*r[0]*f[i];
    z[3*b+YY] = z[3*b+YY] + alpha*r[1]*f[i];
    z[3*b+ZZ] = z[3*b+ZZ] + alpha*r[2]*f[i];
  }
}
//------------------------------------------------------------------------

real evaluate_constraint_function(molecule_t *mol, real * restrict x,
                                  real * restrict g)
{
  /* Computes the constaint function x -> g(x) where g(x) = 0 means that
     the constraint are satisfied. Returns the largest relative error. */

  // ------------------------------------------------------------------------
  // Declaration of internal variables
  // ------------------------------------------------------------------------

  // The number of bonds (n)
  int n;

  // Largest relative bond length violation
  real rel;

  // Atomic indices
  int a, b;

  // Square of the current bond length
  real sigma2;

  // The vector from atom a to atom b
  real r[3];

  // Auxiliary variable used to compute the current constraint
  real aux;

  // ------------------------------------------------------------------------
  // Start of instruction
  // ------------------------------------------------------------------------
```

```
  // Extract the number of bonds (n)
  n = mol->n;

  // Initialize the largest relative error
  rel = 0;

  // Loop over the n constraints
  for(int i=0; i<n; i++)
  {
    // Isolate the atoms which partake in bond i
    a = mol->bonds[2*i]; b = mol->bonds[2*i+1];

    // Isolate the square of the length of the ith bond
    sigma2 = mol->sigma2[i];

    // Compute the vector from atom a to atom b
    r[0] = x[3*b+XX] - x[3*a+XX];
    r[1] = x[3*b+YY] - x[3*a+YY];
    r[2] = x[3*b+ZZ] - x[3*a+ZZ];

    // Compute the square of the length of r
    aux = r[0]*r[0] + r[1]*r[1] + r[2]*r[2];

    // Compute the constraint violation
    g[i] = 0.5*(aux - sigma2);

    // Update the relative error
    rel = max(rel, myfabs(g[i])/sigma2);
  }

  // Return the largest relative (square) bond length violation
  return rel;
}

//------------------------------------------------------------------------
real
evaluate_constraint_function_Svec(molecule_t *mol, real * restrict x,
                                  real * restrict gSchurVec,
                                int block_size, int schur_size)
{
  real rel = 0;
  real sigma2, r[3], aux;
  int a, b;

  gSchurVec[0] = 0;
  gSchurVec[schur_size] = 0;

  for (int j = 1; j < schur_size; j++)
  {
    int i = j*(block_size+1) - 1;
```

```
    // Isolate the atoms which partake in bond i
    a = mol->bonds[2*i]; b = mol->bonds[2*i+1];

    // Isolate the square of the length of the ith bond
    sigma2 = mol->sigma2[i];

    // Compute the vector from atom a to atom b
    r[0] = x[3*b+XX] - x[3*a+XX];
    r[1] = x[3*b+YY] - x[3*a+YY];
    r[2] = x[3*b+ZZ] - x[3*a+ZZ];

    // Compute the square of the length of r
    aux = r[0]*r[0] + r[1]*r[1] + r[2]*r[2];

    // Compute the constraint violation
    gSchurVec[j] = 0.5*(aux - sigma2);

    // Update the relative error
    rel = max(rel, myfabs(gSchurVec[j])/sigma2);
  }

  // Return the largest relative (square) bond length violation
  return rel;
}

//------------------------------------------------------------------------
real evaluate_constraint_function_vec(molecule_t *mol, real * restrict x,
                                      real * restrict gVec, int block_size, int offset)
{
  /* Computes the constaint function x -> g(x) where g(x) = 0 means that
     the constraint are satisfied. Returns the largest relative error. */

  // ------------------------------------------------------------------------
  // Declaration of internal variables
  // ------------------------------------------------------------------------

  // Largest relative bond length violation
  real rel;

  // Atomic indices
  int a, b;

  // Square of the current bond length
  real sigma2;

  // The vector from atom a to atom b
  real r[3];

  // Auxiliary variable used to compute the current constraint
  real aux;
```

```
  // ------------------------------------------------------------------------
  // Start of instruction
  // ------------------------------------------------------------------------

  // Initialize the largest relative error
  rel = 0;

  // Loop over the n constraints
   for (int k=0; k < VLEN; k++)
  {
      for (int j=0; j < block_size; j++)
    {
         int i = k*(block_size+1) + j + offset;
       // Isolate the atoms which partake in bond i
       a = mol->bonds[2*i]; b = mol->bonds[2*i+1];

       // Isolate the square of the length of the ith bond
       sigma2 = mol->sigma2[i];

       // Compute the vector from atom a to atom b
       r[0] = x[3*b+XX] - x[3*a+XX];
       r[1] = x[3*b+YY] - x[3*a+YY];
       r[2] = x[3*b+ZZ] - x[3*a+ZZ];

       // Compute the square of the length of r
       aux = r[0]*r[0] + r[1]*r[1] + r[2]*r[2];

         gVec[j*VLEN+k] = 0.5*(aux - sigma2);

       // Update the relative error
       rel = max(rel, myfabs(gVec[j*VLEN+k])/sigma2);
     }
   }
  // Return the largest relative (square) bond length violation
  return rel;
}
//------------------------------------------------------------------------

/* read lines starting with the % character */
static void
skip_comments(char *line, int length, FILE *fp)
{
  while (fgets(line,length,fp) != NULL)
  {
    if (strncmp("%",line,1) != 0)
    {
      break;
    }
  }
}
//------------------------------------------------------------------------
```

```
/* read the next line */
static void
advance_line(char *line, int length, FILE *fp)
{
  if (fgets(line,length,fp) == NULL)
  {
    printf("ERROR: Unexpected End Of File\n\n");
    exit(-1);
  }
}
//------------------------------------------------------------------------

void read_molecule(molecule_t *mol, char *filename)
{
  /* Reads the description of a molecule from a formatted text file */

  // Atoms, bonds
  int m, n;

  // A file pointer
  FILE *fp;

  // Define a text line of 78 characters.
  char line[78];

  // Length of strings.
  size_t l;

  // Open the file for reading
  fp = fopen(filename, "r");
  if (fp == NULL)
  {
    printf("ERROR: cannot open %s file\n\n", filename);
    exit(1);
  }

  // Skip initial comments
  skip_comments(line,78,fp);
  l = strlen(line);
  mol->name = (char *) calloc((l+1), sizeof(char)); strncpy(mol->name,line,l+1);
  remove_new_line(mol->name);
  skip_comments(line,78,fp);
  l = strlen(line);
  mol->abb = (char *) calloc((l+1), sizeof(char)); strncpy(mol->abb,line,l+1);
  remove_new_line(mol->abb);
  skip_comments(line,78,fp);

  // Read the number of atoms
  sscanf(line,"%d",&m); mol->m = m;
  skip_comments(line,78,fp);
```

```
  // Read the number of bonds
  sscanf(line,"%d",&n); mol->n = n;
  skip_comments(line,78,fp);
  // Read the list of atoms
  mol->atoms = (int *) malloc(m*sizeof(int));
  for(int i=0; i < m; i++)
  {
    sscanf(line,"%d",&(mol->atoms[i]));
    advance_line(line,78,fp);
  }
  skip_comments(line,78,fp);

  // Allocate space for the list of bonds
  mol->bonds = (int *) malloc(2*n*sizeof(int));

  // Allocate space for the bond lengths
  // mol->sigma2 = (real *) malloc(n*sizeof(real));
  smalloc_aligned(mol->sigma2, n, AVX_ALIGNMENT);
  smalloc_aligned(mol->sigma , n, AVX_ALIGNMENT);

  // Read the list of bonds and the bond lengths
  for(int i=0; i<n; i++)
  {
#if PRECISION==0
    // Read the bond length as
    sscanf(line,"%d %d %f",&(mol->bonds[2*i]),&(mol->bonds[2*i+1]),
           &(mol->sigma2[i]));
#else
    sscanf(line,"%d %d %lf",&(mol->bonds[2*i]),&(mol->bonds[2*i+1]),
           &(mol->sigma2[i]));
#endif

    mol->sigma[i] = mysqrt(mol->sigma2[i]);
    advance_line(line,78,fp);
  }

  /* This completes the reading of the molecule.
     We now initialize the invmass as well */

  mol->invmass = (real *) malloc(m*sizeof(real));
  for(int i=0; i<m; i++) mol->invmass[i] = 1;

  // Close the input file
  fclose(fp);
}
//------------------------------------------------------------------------

real
compute_atom_distance(real *x, int a, int b)
{
  // Compute the vector from atom a to atom b
```

```
  real rx = x[3*b+XX] - x[3*a+XX];
  real ry = x[3*b+YY] - x[3*a+YY];
  real rz = x[3*b+ZZ] - x[3*a+ZZ];

  // Compute the square of the length of r
  return(mysqrt(rx*rx + ry*ry + rz*rz));
}
//------------------------------------------------------------------------

void print_molecule_coordinates(molecule_t *mol, real *positions)
{
  // Print a molecule nicely

  // Extract the number of atoms and bonds
  int m = mol->m;
  int n = mol->n;

  if (positions != NULL)
  {
    // Loop over the atoms
    for (int i=0; i < m; i++)
    {
      printf("%9f %9f %9f\n", positions[3*i+XX], positions[3*i+YY], positions[3*i+ZZ]);
    }
  }
  printf("\n");
}
//------------------------------------------------------------------------


void print_molecule(molecule_t *mol, real *positions)
{
  // Print a molecule nicely

  // Extract the number of atoms and bonds
  int m = mol->m;
  int n = mol->n;

  // Print the number of vertices
  printf("PRINT_MOLECULE:\n");
  printf(" Name: %s (%s)\n", mol->name, mol->abb);
  printf(" Number of atoms: %d\n", m);
  printf(" Number of bonds: %d\n", n);

  if (positions != NULL)
  {
    printf(" Atoms positions (x/y/z):\n");

    // Loop over the atoms
    for (int i=0; i < m; i++)
    {
```

```
      printf(" %3d -> %9f / %9f / %9f\n",
             i, positions[3*i+XX], positions[3*i+YY], positions[3*i+ZZ]);
    }
  }

  // Loop over the bonds
  printf(" Bond lengths (real vs. theoretical)\n");
  for (int i=0; i < n; i++)
  {
    int a = mol->bonds[2*i];
    int b = mol->bonds[2*i+1];
    printf(" %3d -> %9f vs. %9f  %4.1f%%\n",
           i, compute_atom_distance(positions, a, b), mysqrt(mol->sigma2[i]),
           100.0*(compute_atom_distance(positions, a, b) - mysqrt(mol->sigma2[i]))/mysqrt
  }
  printf("\n");
}
//---------------------------------------------------------------------------

void extract_matrix(int m, matrix_t *a, matrix_t *b, real *work) {

  /* EXTRACT MATRIX

     We extract from the sparse matrix A precisely those entries which
     correspond to the sparsity pattern of the matrix B.

     This subroutine can be used to copy the lower triangular part of a
     sparse SPD matrix A into a sparse matrix B which is large enough to
     accommodate the fill obtained during the factorization of A.

     WARNING: There is no sanity check of the input.

     DESCRIPTION OF THE INTERFACE:

     ON ENTRY:
       m      integer, the dimension of the matrices
       a      pointer to matrix_t, describing the original matrix
       b      pointer to matrix_t, describing the extended matrix
       work   pointer to real array of length at least m. The first m
              entries must be zero on entry

     ON EXIT:
       m      unchanged
       a      all fields of a are unchanged
       b->val has been updated with the correct values drawn from A
       work   the first m entries are zero on exit
   */

  // ---------------------------------------------------------------------
  // Start of instructions
  // ---------------------------------------------------------------------
```

```
  // Loop over the columns of the sparse matrices
  for(int j=0; j<m; j++)
  {
    // Expand the jth column of A into WORK
    for(int k = a->xadj[j]; k < a->xadj[j+1]; k++) work[a->adj[k]] = a->val[k];
    // Pull the entries of WORK into the jth column of B
    for(int k = b->xadj[j]; k < b->xadj[j+1]; k++) b->val[k] = work[b->adj[k]];
    // Clear the nonzero entries of WORK
    for(int k = a->xadj[j]; k < a->xadj[j+1]; k++) work[a->adj[k]] = 0;
  }
}
```

B.6. schur.h

```
/* SCHUR, version 1.0.0

   Author:
    Rubn  Langarita Bentez,
   Student at Zaragoza University
   Email: 621272@unizar.es

   Date: November 20th, 2017

   A module for doing solving linear systems which are tridiagonal
   using Schur complement method and a dense representation.
*/
#ifndef SCHUR_NEWTON
#define SCHUR_NEWTON

void newtonSchur_init(molecule_t *mol, int block_size);

int newtonSchur_kernel(int num, molecule_t *mol, real *x, real tol, int maxit,
                       int *numit, real *res);

void newtonSchur_free();

// Schur complement versions
// dense
int
newtonSchur(int num, molecule_t *mol, real *x, real tol, int maxit, int *numit, real *res

#endif
```

B.7. schur.c

```c
/* SCHUR

   Author:
    Rubn  Langarita Bentez,
   Student at Zaragoza University
   Email: 621272@unizar.es

   A module for doing solving linear systems which are tridiagonal
   using Schur complement method and a dense representation.
*/

// Standard libraries
#include <stdio.h>
#include <stdlib.h>

// Project specific libraries
#include "aux.h"        // access max and min macros
#include "md-sparse.h"
#include "smalloc.h"
#include "precision.h"
#include "md-newton.h"

//---------------------------------------------------------------------------
// DECLARATION OF GLOBAL VARIABLES
//---------------------------------------------------------------------------

// Work space for the NEXT value of x (X PLUS ONE)
static real *xp1;

// Dense representation of A
static real **A;

// Size of the Schur complements matrix
static int schurSize;

// Array for storing the values of g
static real * g;

// Matrices: B is the lower triangular half of A, C is the Cholesky factor.
static matrix_t bmatrix;

// Size of the blocks, to split A according the Schur complement method
static int block_size;

#if MAKE_MATRIX_OPT
   // Vector to store the distance between the atoms of the bonds
   static real **displacements;
#endif

//---------------------------------------------------------------------------
// HELPER FUNCTIONS TO DEBUGGING
```

```
//--------------------------------------------------------------------------

void printMatrix(int msize)
{
  /* Print the matrix A */

  printf("-------------------------------\n");
  for (int i = 0; i < msize; i++)
  {
    for (int j = 0; j < msize; j++)
    {
      printf("%-8.2g", A[i][j]);
    }
    printf("\n");
  }
  printf("----------------------A---------\n");
}
//--------------------------------------------------------------------------
void printG(int Gsize)
{
  /* Print the vector g */

  printf("-------------------------------\n");
  for (int i = 0; i < Gsize; i++)
  {
      printf("%-8.2g", g[i]);
  }
  printf("\n");
  printf("----------------------g---------\n");
}
//--------------------------------------------------------------------------

void makeOnes(int i, int j)
{
  /* Make ones in the diagonal modifying fillins and g */

  for (int k = i; k <= j; k++)
  {
    g[k] /= A[k][k];
    A[k][i-1] /= A[k][k];
    A[k][j+1] /= A[k][k];
    A[k][k] = 1;
  }
}
//--------------------------------------------------------------------------

void makeOnesF(int i, int j)
{
  /* Same as makeOnes but for the special case of the first block */

  for (int k = i; k <= j; k++)
```

```
  {
    g[k] /= A[k][k];
    A[k][j+1] /= A[k][k];
    A[k][k] = 1;
  }
}
//---------------------------------------------------------------------------

void makeOnesL(int i, int j)
{
  /* Same as makeOnes but for the special case of the last block */

  for (int k = i; k <= j; k++)
  {
    g[k]/=A[k][k];
    A[k][i-1]/=A[k][k];
    A[k][k]=1;
  }
}
//---------------------------------------------------------------------------

void downward(int i, int j)
{
  /* Apply Gaussian elimination from the first element to the last
     for a tridiagonal system */

  for (int k = i; k <= j; k++)
  {
    real factor = A[k+1][k]/A[k][k];
    A[k+1][i-1] -= A[k][i-1]*factor;
    A[k+1][k+1] -= A[k][k+1]*factor;
    g[k+1] -= g[k]*factor;
    A[k+1][k] = 0;
  }
}
//---------------------------------------------------------------------------

void downwardF(int i, int j)
{
  /* Same as downward but for the special case of the first block */

  for (int k = i; k <= j; k++)
  {
    real factor = A[k+1][k]/A[k][k];
    A[k+1][k+1] -= A[k][k+1]*factor;
    g[k+1] -= g[k]*factor;
    A[k+1][k] = 0;
  }
}
//---------------------------------------------------------------------------
```

```
void upward(int i, int j)
{
  /* Apply Gaussian elimination from the last element to the first
     for a tridiagonal system */

  for (int k = i; k >= j; k--)
  {
    real factor = A[k-1][k]/A[k][k];
    A[k-1][i+1] -= A[k][i+1]*factor;
    A[k-1][j-1] -= A[k][j-1]*factor;
    g[k-1] -= g[k]*factor;
    A[k-1][k] = 0;
  }
}
//-------------------------------------------------------------------------

void upwardF(int i, int j)
{
  /* Same as upward but for the special case of the first block */

  for (int k = i; k > j; k--)
  {
    real factor = A[k-1][k]/A[k][k];
    A[k-1][i+1] -= A[k][i+1]*factor;
    g[k-1] -= g[k]*factor;
    A[k-1][k] = 0;
  }
}
//-------------------------------------------------------------------------

void upwardL(int i, int j)
{
  /* Same as upward but for the special case of the last block */

  for (int k = i; k >= j; k--)
  {
    real factor = A[k-1][k]/A[k][k];
    A[k-1][j-1] -= A[k][j-1]*factor;
    g[k-1] -= g[k]*factor;
    A[k-1][k] = 0;
  }
}
//-------------------------------------------------------------------------

void computeFirstBlock(int i, int j)
{
  /* Same as compteBlock but for the special case of the first block */

  downwardF(i, j);
  upwardF(j, i);
  makeOnesF(i, j);
```

```
}
//---------------------------------------------------------------------------

void computeLastBlock(int i, int j)
{
  /* Same as compteBlock but for the special case of the last block */

  downward(i, j - 1);
  upwardL(j, i);
  makeOnesL(i, j);
}
//---------------------------------------------------------------------------

void computeBlock(int i, int j)
{
  /* Helper function that apply Gaussian elimination to solve the system */

  downward(i, j);
  upward(j, i);
  makeOnes(i, j);
}
//---------------------------------------------------------------------------

void step1(int Asize)
{
  /* Calculate the fillins and the Schur complements of the system

     DESCRIPTION OF INTERFACE:

     ON ENTRY (Global variables):
         A               Dense representation of A
         g               Array for storing the values of g
         Asize            Size of A
         block_size      Size of the blocks, to split A according the Schur
                           complement method

    ON EXIT (Global variables):
         A               fillins and Schur complements calculated
                           in their appropriate place
         g               Modified according to the modifications of A

  */

  // Index
  int i;

  // Compute the first block, special case because there are not left fillin
  computeFirstBlock(0, block_size - 1);

  // Loop over the blocks that are not the first nor the last
  for (i = block_size + 1; i < Asize - block_size; i += block_size + 1)
```

```
  {
    computeBlock(i, i + block_size - 1);
  }
  // Compute the last block, special case because there are not right fillin
  computeLastBlock(i, Asize - 1);
}
//--------------------------------------------------------------------------

void step2()
{
  /* Solve the Schur complements system using Gaussian elimination */

  // Indices
  int index, index1;

  // Apply Gaussian elimination downwards
  for (int i = 0; i < schurSize - 1; i++)
  {
    index = i*(block_size+1) + block_size;
    index1 = index + block_size + 1;
    real factor = A[index1][index]/A[index][index];
    A[index1][index1] -= A[index][index1]*factor;
    g[index1] -= g[index]*factor;
    A[index1][index] = 0;
  }

  // Apply Gaussian elimination upward
  for (int i = schurSize - 1; i > 0; i--)
  {
    index = i*(block_size+1) + block_size;
    index1 = index - block_size - 1;
    real factor = A[index1][index]/A[index][index];
    A[index1][index1] -= A[index][index1]*factor;
    g[index1] -= g[index]*factor;
    A[index1][index] = 0;
  }

  // Make ones in the diagonal modifying g
  for (int i = 0; i < schurSize; i++)
  {
    index = i*(block_size+1) + block_size;
    g[index]/=A[index][index];
    A[index][index] = 1;
  }
}
//--------------------------------------------------------------------------

void makeZerosSchur(int index)
{
  /* Apply Gaussian elimination to the fillins of a
      determinate Schur complement */
```

```
  // Loop over the superior part of the fillin
  for(int i = index - block_size; i < index; i++)
  {
    g[i] -= g[index]*A[i][index];
    A[i][index] -= A[index][index]*A[i][index];
  }

  // Loop over the interior part of the fillin
  for(int i = index + 1; i <= index + block_size; i++)
  {
    g[i] -= g[index]*A[i][index];
    A[i][index] -= A[index][index]*A[i][index];
  }
}
//-------------------------------------------------------------------------

void step3()
{
  /* Apply Gaussian elimination to the fillins */

  // Indices
  int index;

  // Loop over the diagonal of the Schur complements matrix, except the last
  for (int i = 0; i < schurSize; i++)
  {
    index = i*(block_size+1) + block_size;
    makeZerosSchur(index);
  }
}
//-------------------------------------------------------------------------

static void
schur_function(int Asize, int *xadj, int *adj, real * restrict val)
{
  /* SCHUR_FUNCTION: Dense representation

    Solve a linear system which is tridiagonal using Schur complement
    method and a dense representation.

    DESCRIPTION OF THE INTERFACE

    ON ENTRY:
      Asize        the dimension of the matrix
      adj          pointer to the compressed adjacency lists
      xadj         XADJ[j] is the index in ADJ of the first element
                   of the jth column of the matrix
      val          pointer to the list on nonzero elements of the matrix

    ON ENTRY (GLOBAL):
```

```
      g            vector g, independent term

    ON EXIT (GLOBAL):
      g            the vector will be modified with the solution
                   of the system

  */

  // Calculate the size of the Schur complements matrix
  schurSize = Asize/(block_size+1);

  // Alloc memory for the dense representation of A
  A = (real**)calloc(Asize, sizeof(real*));
  for (int i = 0; i < Asize; i++)
    A[i] = (real*)calloc(Asize, sizeof(real));

  // Fill A with the values of the sparse representation
  for (int i = 0; i < Asize; i++)
  {
    for (int j = xadj[i]; j < xadj[i+1]; j++)
    {
      A[adj[j]][i] = val[j];
    }
  }

  // Apply the 3 steps needed to solve the system
  step1(Asize); // Calculate fillins and Schur complements matrix
  step2(); // Solve the Schur complements system
  step3(); // Apply Gaussian elimination to the fillins

  // Release the memory allocated
  for (int i = 0; i < Asize; i++) free(A[i]);
  free(A);
}
//---------------------------------------------------------------------------

void
newtonSchur_init(molecule_t *mol, int block_size_par)
{
  /* NEWTONSchur: Dense representation

     Attempts to solve the constraint equations corresponding to NUM copies
     of the same molecule with bond constraints using Newton's method.

     DESCRIPTION OF INTERFACE:

     ON ENTRY:
        mol->m          integer, the number of atoms
        mol->n          integer, the number of bonds
        mol->bonds      pointer to integer, the sequential bond list
        mol->bond_graph pointer to graph, the LOWER triangular bond graph
```

```
    ON EXIT:
        mol             all fields unchanged

    RETURNS:
        the number of equations solved to the requested tolerance using at
        most MAXIT iterations.

  */

  // ------------------------------------------------------------------------
  // Declaration of interval variables
  // ------------------------------------------------------------------------

  // The number of atoms (m) and the number of bonds (n)
  int m, n;

  // Format specification
  int format = CSC;

  // ------------------------------------------------------------------------
  // Start of instructions
  // ------------------------------------------------------------------------

  // Copy the parameters to the global variables
  block_size = block_size_par;

  // Extract the number of atoms (m) and the number of bonds (n)
  m = mol->m; n = mol->n;

  // Allocate space for the NEXT value of X (X PLUS ONE)
  smalloc_aligned(xp1, 3*m, AVX_ALIGNMENT);

  // Allocate space for values of g
  smalloc_aligned(g, n, AVX_ALIGNMENT);

  // Initialize the matrix using the specificed format
  init_matrix(format, mol->bond_graph, &bmatrix);

#if MAKE_MATRIX_OPT
  // Allocate space for the displacement vector
  displacements = alloc_displacements(mol);
#endif
}
//---------------------------------------------------------------------------

int
newtonSchur_kernel(int num, molecule_t *mol, real *x, real tol, int maxit,
                   int *numit, real *res)
{
  /* NEWTONSchur: Dense representation
```

```
    Attempts to solve the constraint equations corresponding to NUM copies
    of the same molecule with bond constraints using Newton's method.

    DESCRIPTION OF INTERFACE:

    ON ENTRY:
       num             integer, the number of molecules
       mol->m          integer, the number of atoms
       mol->n          integer, the number of bonds
       mol->bonds      pointer to integer, the sequential bond list
       mol->bond_graph pointer to graph, the LOWER triangular bond graph
       x               pointer to real, the list of initial positions
       tol             real, the convergence tolerance
       maxit           integer, the maximum number of iterations
       res             pointer to real array of length at least NUM

    ON EXIT:
       mol             all fields unchanged
       x               contains the last iterate for all equations
       tol             unchanged
       maxit           unchanged
       res             res[k] is the final relative constraint violation
                       for the kth equation.

    RETURNS:
       the number of equations solved to the requested tolerance using at
       most MAXIT iterations.

  */

  // ------------------------------------------------------------------------
  // Declaration of interval variables
  // ------------------------------------------------------------------------

  // The return code. Here we assume total failure!
  int rc = 0;

// #pragma omp parallel reduction(+:rc)
// {

  // The number of atoms (m) and the number of bonds (n)
  int m, n;

  // Local pointer into the spatial data
  real *lx;

  // Constant needed during the linear updates
  real alpha = -1;

  // Maximum relative bond length violation
```

```
  real tau;

  // ------------------------------------------------------------------------
  // Start of instructions
  // ------------------------------------------------------------------------

  // Extract the number of atoms (m) and the number of bonds (n)
  m = mol->m; n = mol->n;

  // Process the molecules one at a time
  //#pragma omp for
  for (int k=0; k < num; k++)
  {
    // Establish a local pointer to the spatial data
    lx = &x[3*m*k];

    // Copy lx into xp1
    #pragma GCC ivdep
    for (int j=0; j < 3*m; j++) xp1[j] = lx[j];

    // Compute g(x)
    tau = evaluate_constraint_function(mol, lx, g);
#if TAU_OUTPUT
    printf("\n");
    printf("\tit. 0: tau=%e\n", tau);
    // ddm(n, 1, g, n, "%16.8e", " constraint violation\n");
#endif

    // Do exactly MAXIT Newton steps. This will be modified in the future
    for ((*numit)=0; (*numit < maxit) && (tol < tau); (*numit)++)
    {
      /* We compute xp1 (read : X PLUS ONE) as follows

        xp1 = x - Dg(x)'*(inv(Dg(x)*Dg(x)')*g(x))

       and then we manually copy xp1 into x. */

      // Make the lower triangular part of matrix A(x,x)
#if MAKE_MATRIX_OPT
     compute_displacements(mol, lx, displacements);
      make_matrix_opt(mol, &bmatrix, displacements);
#else
      make_matrix(mol, lx, &bmatrix);
#endif

      // Solve the system using the Schur method
      schur_function(n, bmatrix.xadj, bmatrix.adj, bmatrix.val);

      // Do the linear update
      apply_jacobian("T", alpha, mol, lx, g, xp1);
```

```
      // Manually copy XP1 into X
      #pragma GCC ivdep
      for (int j=0; j<3*m; j++) lx[j] = xp1[j];

      // Compute g(x)
      tau = evaluate_constraint_function(mol, lx, g);
#if TAU_OUTPUT
      printf("\tit.%2d: tau=%e\n", (*numit)+1, tau);
      // ddm(n, 1, g, n, "%16.8e", " constraint violation\n");
#endif
    }

    // Adjust the return code
    if (tau < tol) rc++; // Increment the number of equations solved.

    // Save the final relative constraint violation for the kth equation
    res[k] = tau;
  }

// }
// pragma omp parallel

  // End of subroutine
  return rc;
}
//-----------------------------------------------------------------------------

void
newtonSchur_free()
{
  /* NEWTONSchur: Dense representation

     Attempts to solve the constraint equations corresponding to NUM copies
     of the same molecule with bond constraints using Newton's method.

     DESCRIPTION OF INTERFACE:

     ON ENTRY:

     ON EXIT:

  */

  //-----------------------------------------------------------------------
  // Release the memory directly allocated by the subroutine.
  //-----------------------------------------------------------------------
  free(xp1); free(g);

  // Release the memory allocated by siblings
  free_matrix(&bmatrix);
```

```
#if MAKE_MATRIX_OPT
  free_displacements(displacements);
#endif

  // End of subroutine
}
//---------------------------------------------------------------------------

int
newtonSchur(int num, molecule_t *mol, real *x, real tol, int maxit,
            int *numit, real *res, int block_size_par)
{
  /* NEWTONSchur: Dense representation

     Attempts to solve the constraint equations corresponding to NUM copies
     of the same molecule with bond constraints using Newton's method.

     DESCRIPTION OF INTERFACE:

     ON ENTRY:
        num             integer, the number of molecules
        mol->m          integer, the number of atoms
        mol->n          integer, the number of bonds
        mol->bonds      pointer to integer, the sequential bond list
        mol->bond_graph pointer to graph, the LOWER triangular bond graph
        x               pointer to real, the list of initial positions
        tol             real, the convergence tolerance
        maxit           integer, the maximum number of iterations
        res             pointer to real array of length at least NUM

     ON EXIT:
        mol             all fields unchanged
        x               contains the last iterate for all equations
        tol             unchanged
        maxit           unchanged
        res             res[k] is the final relative constraint violation
                        for the kth equation.

     RETURNS:
        the number of equations solved to the requested tolerance using at
        most MAXIT iterations.

  */

  // -------------------------------------------------------------------------
  // Declaration of interval variables
  // -------------------------------------------------------------------------

  // The return code. Here we assume total failure!
  int rc = 0;
```

```
// #pragma omp parallel reduction(+:rc)
// {

  // The number of atoms (m) and the number of bonds (n)
  int m, n;

  // Format specification
  int format = CSC;

  // Local pointer into the spatial data
  real *lx;

  // Constant needed during the linear updates
  real alpha = -1;

  // Maximum relative bond length violation
  real tau;

  // ------------------------------------------------------------------------
  // Start of instructions
  // ------------------------------------------------------------------------

  // Copy the parameters to the global variables
  block_size = block_size_par;

  // Extract the number of atoms (m) and the number of bonds (n)
  m = mol->m; n = mol->n;

  // Allocate space for values of g
  smalloc_aligned(g, n, AVX_ALIGNMENT);

  // Initialize the matrix using the specificed format
  init_matrix(format, mol->bond_graph, &bmatrix);

  // Allocate space for the NEXT value of X (X PLUS ONE)
  smalloc_aligned(xp1, 3*m, AVX_ALIGNMENT);

#if MAKE_MATRIX_OPT
  // Allocate space for the displacement vector
  displacements = alloc_displacements(mol);
#endif

  // Process the molecules one at a time
  //#pragma omp for
  for (int k=0; k < num; k++)
  {
    // Establish a local pointer to the spatial data
    lx = &x[3*m*k];

    // Copy lx into xp1
    #pragma GCC ivdep
```

```
    for (int j=0; j < 3*m; j++) xp1[j] = lx[j];

    // Compute g(x)
    tau = evaluate_constraint_function(mol, lx, g);
#if TAU_OUTPUT
    printf("\n");
    printf("\tit. 0: tau=%e\n", tau);
    // ddm(n, 1, g, n, "%16.8e", " constraint violation\n");
#endif

    // Do exactly MAXIT Newton steps. This will be modified in the future
    for ((*numit)=0; (*numit < maxit) && (tol < tau); (*numit)++)
    {
      /* We compute xp1 (read : X PLUS ONE) as follows

        xp1 = x - Dg(x)'*(inv(Dg(x)*Dg(x)')*g(x))

       and then we manually copy xp1 into x. */

      // Make the lower triangular part of matrix A(x,x)
#if MAKE_MATRIX_OPT
     compute_displacements(mol, lx, displacements);
      make_matrix_opt(mol, &bmatrix, displacements);
#else
      make_matrix(mol, lx, &bmatrix);
#endif

      // Solve the system using the Schur method
      schur_function(n, bmatrix.xadj, bmatrix.adj, bmatrix.val);

      // Do the linear update
      apply_jacobian("T", alpha, mol, lx, g, xp1);

      // Manually copy XP1 into X
      #pragma GCC ivdep
      for (int j=0; j<3*m; j++) lx[j] = xp1[j];

      // Compute g(x)
      tau = evaluate_constraint_function(mol, lx, g);
#if TAU_OUTPUT
      printf("\tit.%2d: tau=%e\n", (*numit)+1, tau);
      // ddm(n, 1, g, n, "%16.8e", " constraint violation\n");
#endif
    }

    // Adjust the return code
    if (tau < tol) rc++; // Increment the number of equations solved.

    // Save the final relative constraint violation for the kth equation
    res[k] = tau;
  }
```

```
  //---------------------------------------------------------------------
  // Release the memory directly allocated by the subroutine.
  //---------------------------------------------------------------------
  free(xp1); free(g);

  // Release the memory allocated by siblings
  free_matrix(&bmatrix);

#if MAKE_MATRIX_OPT
  free_displacements(displacements);
#endif

// }
// pragma omp parallel

  // End of subroutine
  return rc;
}
//---------------------------------------------------------------------
```

B.8. schurSparse.h

```
/* SCHUR_SPARSE, version 1.0.0

   Author:
    Rubn  Langarita Bentez,
   Student at Zaragoza University
   Email: 621272@unizar.es

   Date: November 20th, 2017

   A module for doing solving linear systems which are sparse and tridiagonal
   using Schur complement method.
*/
#ifndef SCHUR_SPARSE_NEWTON
#define SCHUR_SPARSE_NEWTON

void newtonSchurSparse_init(molecule_t *mol, int block_size);

int newtonSchurSparse_kernel(int num, molecule_t *mol, real *x, real tol, int maxit,
                             int *numit, real *res);

void newtonSchurSparse_free();

int newtonSchurSparse(int num, molecule_t *mol, real *x, real tol, int maxit, int *numit,


#endif
```

B.9. schurSparse.c

```
/* SCHUR

   Author:
    Rubn  Langarita Bentez,
   Student at Zaragoza University
   Email: 621272@unizar.es

   A module for doing solving linear systems which are tridiagonal
   using Schur complement method and a dense representation.
*/

// Standard libraries
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Project specific libraries
#include "aux.h"        // access max and min macros
#include "md-sparse.h"
#include "smalloc.h"
#include "precision.h"
#include "md-newton.h"

/* Set to !=0 to solve as a tridiagonal system.
    It does not work with no tridiagonal systems*/
#define TRIDIAGONAL 1

//------------------------------------------------------------------------
// DECLARATION OF GLOBAL VARIABLES
//------------------------------------------------------------------------

// Work space for the NEXT value of x (X PLUS ONE)
static real *xp1;

// Variables for the schur function
// Schur complements matrix
static real *schurMatrix[3];
// real **schurMatrix;

// Size of the Schur complements matrix
static int schurSize;

// To store the fillins
static real **fillin;
```

```
// Array for storing the values of g
static real *g;

// B is the lower triangular half of A
static matrix_t bmatrix;

// Size of the blocks in the Schur complement method
static int block_size = 0;

// Index in ADJ of the first element of the jth column of the matrix
static int *xadj;

// Pointer to the compressed adjacency lists
static int *adj;

// Pointer to the list on nonzero elements of the matrix
static real * val;

#if MAKE_MATRIX_OPT
   // Vector to store the distance between the atoms of the bonds
   static real **displacements;
#endif

// Auxiliary vectors to store B(L) and D(U)
#if !TRIDIAGONAL
static real *bVectorUp;
static real *bVectorDown;
#endif

//------------------------------------------------------------------------
// HELPER FUNCTIONS TO DEBUGGING
//------------------------------------------------------------------------

void printSparseMatrix(int size)
{
   /* Print the matrix A */
  real mat[size][size];

  for (int i = 0; i < size; i++)
  {
    for (int j = 0; j < size; j++)
    {
      mat[i][j] = 0;
    }
  }
  for (int i = 0; i < size; i++)
  {
    for (int j = xadj[i]; j < xadj[i+1]; j++)
    {
      mat[adj[j]][i] = val[j];
    }
```

```
  }
  printf("--------------------------------\n");
  for (int i = 0; i < size; i++)
  {
    for (int j = 0; j < size; j++)
    {
      printf("%-8.2g", mat[i][j]);
    }
    printf("\n");
  }
  printf("--------------------------------\n");
}
//-------------------------------------------------------------------------

void printSchurMatrix()
{
   /* Print Schur complements matrix */

  printf("--------------------------------\n");
  for(int i = 0; i < schurSize - 1; i++)
  {
    printf("%-8.2g", schurMatrix[0][i]);
  }
  printf("\n");
  for(int i = 0; i < schurSize; i++)
  {
    printf("%-8.2g", schurMatrix[1][i]);
  }
  printf("\n");
  for(int i = 0; i < schurSize - 1; i++)
  {
    printf("%-8.2g", schurMatrix[2][i]);
  }
  printf("\n");
  printf("----------------------schurMatrix---------\n");
}
//-------------------------------------------------------------------------

void printFill(real * fill)
{
   /* Print a specific fillin */

  printf("--------------------------------\n");
  for(int i = 0; i < block_size; i++)
  {
      printf("%-8.2g", fill[i]);
  }
  printf("\n");
  printf("----------------------fill---------\n");
}
//-------------------------------------------------------------------------
```

```
void printFills()
{
   /* Print all the fillins */

  printf("--------------------------------\n");
  for(int i = 0; i < schurSize; i++)
   {
    for(int j = 0; j < block_size*2; j++)
    {
      printf("%-8.2g", fillin[i][j]);
    }
    printf("\n");
  }
  printf("\n");
  printf("----------------------fills---------\n");
}
//---------------------------------------------------------------------------

void printSparseG(int size)
{
   /* Print the vector g */

  printf("--------------------------------\n");
  for(int i = 0; i < size; i++)
  {
    printf("%-8.2g", g[i]);
  }
  printf("\n");
  printf("----------------------g---------\n");
}
//---------------------------------------------------------------------------

static void
factorization(int start)
{
   /* Factorize a block from start to start+block_size.
       It does not work with no tridiagonal matrices */
  real factor;

  for(int i = start; i < start + block_size - 1; i++)
  {
    factor = val[xadj[i+1]-1]/val[xadj[i+1]-2];
    val[xadj[i+1]-1] = factor;
    val[xadj[i+1]+1] -= val[xadj[i+1]]*factor;
  }
}
//---------------------------------------------------------------------------

static void
forwardLU(real * fill, int start)
```

```
{
   /* Solves Lx = f using forward substitution */

  for(int i = 1; i < block_size; i++)
  {
    fill[i] -= val[xadj[i+start]-1]*fill[i-1];
  }
}
//---------------------------------------------------------------------------

static void
backwardLU(real * fill, int start)
{
   /* Solves Ux = f using backward substitution */

  fill[block_size-1] /= val[xadj[start+block_size-1]+1];
  for(int i = block_size - 1; i > 0; i--)
  {
    fill[i-1] = (fill[i-1] - val[xadj[start+i]]*fill[i])/val[xadj[start+i]-2];
  }
}
//---------------------------------------------------------------------------

static void
solveLU(real * fill, int start)
{
   /* Solves LUx = f using backward and forward substitution */

  forwardLU(fill, start);
  backwardLU(fill, start);
}

#if !TRIDIAGONAL
//---------------------------------------------------------------------------

static void
calculateSchur(real *schur, real *bVector, real *fill)
{
   /* Calculate the scalar product of bVector and fill
       Use it only for no tridiagonal matrices

     DESCRIPTION OF INTERFACE:

     ON ENTRY:
         bVector        First vector
         fill           Second vector

    ON EXIT:
         schur          Result of the scalar product of bVector and fill

  */
```

```
  real sum = 0;
  for(int i = 0; i < block_size; i++)
  {
    sum += bVector[i]*fill[i];
  }
  *schur -= sum;
}
#endif
//------------------------------------------------------------------------

static void
computeFirstBlockSparse()
{
   /* Same as computeBlockSparse but for the special case of the first block */

  fillin[0][block_size-1] = val[xadj[block_size]]; // @TODO fill properly for no tridiagon

  factorization(0);

  solveLU(fillin[0], 0);
  solveLU(&g[0], 0);

#if TRIDIAGONAL
  schurMatrix[1][0] -= val[xadj[block_size-1]+2]*fillin[0][block_size-1];
  g[block_size-1+1] -= val[xadj[block_size-1]+2]*g[block_size-1];
#else
  bVectorDown[block_size-1] = val[xadj[block_size-1]+1];// @TODO fill properly for no tri
  calculateSchur(&schurMatrix[1][0], bVectorDown, fillin[0]);
  calculateSchur(&g[block_size-1+1], bVectorDown, &g[0]);
#endif
}
//------------------------------------------------------------------------

static void
computeLastBlockSparse(int start, real *fillLeft)
{
   /* Same as computeBlockSparse but for the special case of the last block */

  fillLeft[0] = val[xadj[start-1]+2]; // @TODO fill properly for no tridiagonal matrices

  factorization(start);
  solveLU(fillLeft, start);
  solveLU(&g[start], start);

#if TRIDIAGONAL
  schurMatrix[1][schurSize-1] -= val[xadj[start]]*fillLeft[0];
  g[start-1] -= val[xadj[start]]*g[start];
#else
  bVectorUp[0] = val[xadj[start]]; // @TODO fill properly for no tridiagonal matrices
  calculateSchur(&schurMatrix[1][schurSize-1], bVectorUp, fillLeft);
```

```
  calculateSchur(&g[start-1], bVectorUp, &g[start]);
#endif
}
//-------------------------------------------------------------------------

static void
computeBlockSparse(int start, int end, real *fillLeft, real *fillRight)
{
   /* Calculate fillins and schur complements for a given block of the matrix.
       It only work with tridiagonal systems.

     DESCRIPTION OF INTERFACE:

     ON ENTRY:
        start          Index of the beginning of the block
        end            Index of the end of the block
            fillLeft       Pointer to the left fillin of the block
        fillRight      Pointer to the right fillin of the block

      ON ENTRY AS GLOBAL VARIABLES:
            xadj           XADJ[j] is the index in ADJ of the first element
                                 of the jth column of the matrix
            val            Pointer to the list on nonzero elements of the matrix
            block_size     Size of the blocks
            g              Vector g, independent term
            schurMatrix    Schur complements matrix
            bVectorUp      Pointer to D(U) vector
            bVectorDown    Pointer to B(L) vector

    ON EXIT:
         fillLeft
           fillRight      Fillins updated according to the schur method

      ON EXIT AS GLOBAL VARIABLES:
          schurMatrix    Entries corresponding to the block updated
                                  according to the schur method

  */

  fillRight[block_size-1] = val[xadj[end+1]];// @TODO fill properly for no tridiagonal mat
  fillLeft[0] = val[xadj[start-1]+2];

  factorization(start);

  solveLU(fillLeft, start);
  solveLU(fillRight, start);
  solveLU(&g[start], start);

  int index = start*(1.0/(block_size+1)) - 1;
#if TRIDIAGONAL
  schurMatrix[1][index] -= val[xadj[start]]*fillLeft[0];
```

```
  schurMatrix[2][index] -= val[xadj[start]]*fillRight[0];
  g[start-1] -= val[xadj[start]]*g[start];

  schurMatrix[0][index] -= val[xadj[end]+2]*fillLeft[block_size-1];
  schurMatrix[1][index+1] -= val[xadj[end]+2]*fillRight[block_size-1];
  g[end+1] -= val[xadj[end]+2]*g[end];
#else
  bVectorUp[0] = val[xadj[start]]; // @TODO fill properly for no tridiagonal matrices
  bVectorDown[block_size-1] = val[xadj[end]+2];
  calculateSchur(&schurMatrix[1][index], bVectorUp, fillLeft);
  calculateSchur(&schurMatrix[2][index], bVectorUp, fillRight);
  calculateSchur(&g[start-1], bVectorUp, &g[start]);

  calculateSchur(&schurMatrix[0][index], bVectorDown, fillLeft);
  calculateSchur(&schurMatrix[1][index+1], bVectorDown, fillRight);
  calculateSchur(&g[end+1], bVectorDown, &g[start]);
#endif
}
//-----------------------------------------------------------------------------

void step1Sparse(int Asize)
{
   /* Calculate the fillins and the Schur complements of the system

     DESCRIPTION OF INTERFACE:

     ON ENTRY:
         Asize           Size of A

     ON ENTRY (Global variables):
           adj            Pointer to the compressed adjacency lists
           xadj           XADJ[j] is the index in ADJ of the first element
                                of the jth column of the matrix
           val            Pointer to the list on nonzero
                                    elements of the matrix
         g              Array for storing the values of g
         block_size     Size of the blocks, to split A according the Schur
                        complement method

    ON EXIT (Global variables):
         fillin               Modified according to the schur method
             schurMatrix      Modified according to the schur method
         g              Modified according to the modifications of A

  */
   int i;

  computeFirstBlockSparse();
  for(i = block_size + 1; i < Asize - block_size; i += block_size + 1)
  {
    int index = i*(1.0/(block_size+1));
```

```
    computeBlockSparse(i, i + block_size - 1, &fillin[index-1][block_size], fillin[index])
  }
  computeLastBlockSparse(i, &fillin[schurSize-1][block_size]);
}
//-------------------------------------------------------------------------

static void
factorizationSchur()
{
   /* Factorize the schur complements matrix */

  for(int i = 1; i < schurSize; i++)
  {
    real factor = schurMatrix[0][i-1]/schurMatrix[1][i-1];
    schurMatrix[0][i-1] = factor;
    schurMatrix[1][i] -= schurMatrix[2][i-1]*factor;
  }
}
//-------------------------------------------------------------------------

static void
forwardLUSchur()
{
   /* Solves Lx = f using forward substitution */

  for(int i = 1; i < schurSize; i++)
  {
    int index = i*(block_size+1) + block_size;
    g[index] -= schurMatrix[0][i-1]*g[index-block_size-1];
  }
}
//-------------------------------------------------------------------------

static void
backwardLUSchur()
{
   /* Solves Ux = f using backward substitution */

  g[(schurSize-1)*(block_size+1)+block_size] /= schurMatrix[1][schurSize-1];
  for(int i = schurSize - 2; i >= 0; i--)
  {
    int index = i*(block_size + 1) + block_size;
    g[index] = (g[index] - schurMatrix[2][i]*g[index + block_size + 1])/schurMatrix[1][i];
  }
}
//-------------------------------------------------------------------------

void step2Sparse()
{
   /* Solve the schur complements matrix.
       Factorize the schur complements matrix and solves the system LUx = f
```

```
      using backward and forward substitution. */

  factorizationSchur();
  forwardLUSchur();
  backwardLUSchur();
}
//---------------------------------------------------------------------------

static void
solveFill(real *fillLeft, real *fillRight, real *pg)
{
   /* Apply Gaussian elimination to the fillins of a given block */

  real gUp = pg[-1];
   real gDown = pg[block_size];
  for(int i = 0; i < block_size; i++)
  {
    pg[i] -= gUp*fillLeft[i] + gDown*fillRight[i];
  }
}
//---------------------------------------------------------------------------

static void
solveFillLast(real *fillLeft, real *pg)
{
   /* Same as solveFill but for the special case of the last block */

  real gUp = pg[-1];
  for(int i = 0; i < block_size; i++)
  {
    pg[i] -= gUp*fillLeft[i];
  }
}
//---------------------------------------------------------------------------

static void
solveFillFirst(real *fillRight, real *pg)
{
   /* Same as solveFill but for the special case of the first block */

   real gDown = pg[block_size];
  for(int i = 0; i < block_size; i++)
  {
    pg[i] -= gDown*fillRight[i];
  }
}
//---------------------------------------------------------------------------

static void
step3Sparse()
{
```

```
   /* Apply Gaussian elimination to all the fillins */

   solveFillFirst(fillin[0], &g[0]);
  for(int i = 1; i < schurSize; i++)
  {
    int index = i*(block_size + 1);
    solveFill(&fillin[i-1][block_size], fillin[i], &g[index]);
  }
   solveFillLast(&fillin[schurSize-1][block_size], &g[schurSize*(block_size+1)]);
}
//---------------------------------------------------------------------------

static void
schur_function_sparse(int Asize, int *_xadj, int *_adj, real * restrict _val)
{
   /* SCHUR_FUNCTION_SPARSE: Sparse representation

      Solve a linear system which is tridiagonal using Schur complement
      method and a sparse representation.

      DESCRIPTION OF THE INTERFACE

      ON ENTRY:
         Asize        The dimension of the matrix
         _block_size  Size of the blocks, to split A according
                             the Schur complement method
         _adj         Pointer to the compressed adjacency lists
         _xadj        XADJ[j] is the index in ADJ of the first element
                             of the jth column of the matrix
         _val         Pointer to the list on nonzero elements of the matrix
         f            Vector g, independent term

      ON EXIT:
         f            The vector will be modified with the solution
                             of the system

   */

   // Copy the parameters to the global variables
   xadj = _xadj;
   adj = _adj;
   val = _val;

   // Calculate the size of the Schur complements matrix
   schurSize = Asize/(block_size + 1);

   // Alloc space for D(U) and B(L) auxiliary vectors
#if !TRIDIAGONAL
  bVectorUp = (real*)calloc(block_size, sizeof(real));
  bVectorDown = (real*)calloc(block_size, sizeof(real));
#endif
```

```
   /* Set to 0 the superior and inferior diagonals
       of the schur complements matrix */
  for(int j = 0; j < schurSize - 1; j++)
  {
    schurMatrix[0][j] = 0;
    schurMatrix[2][j] = 0;
  }

   // Fill the diagonal of the schur complements matrix
   for(int i = 0; i < schurSize; i++)
  {
    int index = i*(block_size+1) + block_size;
    schurMatrix[1][i] = val[xadj[index]+1];
  }

   // Set to 0 the fillins
  for(int i = 0; i < schurSize; i++)
  {
    for(int j =0; j < block_size*2; j++)
    {
      fillin[i][j] = 0;
    }
  }

   // Apply the 3 steps needed to solve the system
  step1Sparse(Asize);  // Calculate fillins and Schur complements matrix
  step2Sparse();     // Solve the Schur complements system
  step3Sparse();     // Apply Gaussian elimination to the fillins

   // Release the memory allocated
#if !TRIDIAGONAL
  free(bVectorUp);
  free(bVectorDown);
#endif
}
//---------------------------------------------------------------------------

void newtonSchurSparse_init(molecule_t *mol, int block_size_par)
{
  /* NEWTONSchurSparse: Sparse representation

     Attempts to solve the constraint equations corresponding to NUM copies
     of the same molecule with bond constraints using Newton's method.

     DESCRIPTION OF INTERFACE:

     ON ENTRY:
        num             integer, the number of molecules
        mol
        x               pointer to real, the list of initial positions
```

```
     tol            real, the convergence tolerance
     maxit          integer, the maximum number of iterations

  ON EXIT:
     mol            all fields unchanged

  RETURNS:
     the number of equations solved to the requested tolerance using at
     most MAXIT iterations.
*/

// ------------------------------------------------------------------------
// Declaration of interval variables
// ------------------------------------------------------------------------

// The number of atoms (m) and the number of bonds (n)
int n, m;

// Format specification
int format = CSC;

// ------------------------------------------------------------------------
// Start of instructions
// ------------------------------------------------------------------------

// Copy the parameters to the global variables
block_size = block_size_par;

// Extract the number of atoms (m) and the number of bonds (n)
n = mol->n; m = mol->m;

// Allocate space for the NEXT value of X (X PLUS ONE)
smalloc_aligned(xp1, 3*m, AVX_ALIGNMENT);

// Allocate space for values of g
smalloc_aligned(g, n, AVX_ALIGNMENT);

// Initialize the matrix using the specificed format
init_matrix(format, mol->bond_graph, &bmatrix);

// Prepare variables for the schur function
schurSize = n/(block_size_par+1);

// schurMatrix = (real**)malloc(3*sizeof(real*));
 schurMatrix[0] = (real*)calloc(schurSize, sizeof(real));
 schurMatrix[1] = (real*)calloc(schurSize, sizeof(real));
schurMatrix[2] = (real*)calloc(schurSize, sizeof(real));

fillin = (real**)malloc(schurSize*sizeof(real*));
for (int i=0; i < schurSize; i++)
  fillin[i] = (real*)calloc(block_size*2, sizeof(real));
```

```
#if MAKE_MATRIX_OPT
  // Allocate space for the displacement vector
  displacements = alloc_displacements(mol);
#endif
}
//-------------------------------------------------------------------------

void newtonSchurSparse_free()
{
  /* NEWTONSchurSparse: Sparse representation

     Attempts to solve the constraint equations corresponding to NUM copies
     of the same molecule with bond constraints using Newton's method.

     DESCRIPTION OF INTERFACE:

     ON ENTRY:

     ON EXIT:

  */

  // ------------------------------------------------------------------------
  // Declaration of interval variables
  // ------------------------------------------------------------------------

  // ------------------------------------------------------------------------
  // Start of instructions
  // ------------------------------------------------------------------------

  //-------------------------------------------------------------------------
  // Release the memory directly allocated by the subroutine.
  //-------------------------------------------------------------------------
  free(xp1); free(g);

  // Release the memory allocated by siblings
  free_matrix(&bmatrix);

   free(schurMatrix[0]);
   free(schurMatrix[1]);
   free(schurMatrix[2]);

  for (int i=0; i < schurSize; i++) free(fillin[i]);
  free(fillin);

#if MAKE_MATRIX_OPT
  free_displacements(displacements);
#endif

}
```

```
//------------------------------------------------------------------------

int newtonSchurSparse_kernel(int num, molecule_t *mol, real *x, real tol, int maxit,
                             int *numit, real *res)
{
  /* NEWTONSchurSparse: Sparse representation

     Attempts to solve the constraint equations corresponding to NUM copies
     of the same molecule with bond constraints using Newton's method.

     DESCRIPTION OF INTERFACE:

     ON ENTRY:
        num             integer, the number of molecules
        mol->m          integer, the number of atoms
        mol->n          integer, the number of bonds
        mol->bonds      pointer to integer, the sequential bond list
        mol->bond_graph pointer to graph, the LOWER triangular bond graph
        x               pointer to real, the list of initial positions
        tol             real, the convergence tolerance
        maxit           integer, the maximum number of iterations
        numit           pointer to real
        res             pointer to real array of length at least NUM

    ON EXIT:
        mol             all fields unchanged
        x               contains the last iterate for all equations
        tol             unchanged
        maxit           unchanged
        numit           number of algorithm iterations executed
        res             res[k] is the final relative constraint violation
                        for the kth equation.

    RETURNS:
        the number of equations solved to the requested tolerance using at
        most MAXIT iterations.
  */

  // ------------------------------------------------------------------------
  // Declaration of interval variables
  // ------------------------------------------------------------------------

  // The return code. Here we assume total failure!
  int rc = 0;

  // The number of atoms (m) and the number of bonds (n)
  int m, n;

  // Local pointer into the spatial data
  real *lx;
```

```
  // Constant needed during the linear updates
  real alpha = -1;

  // Maximum relative bond length violation
  real tau;

#if DISPLAY_TIME_SCHUR
   struct timeval start,end;
   double total = 0;
#endif

  // ------------------------------------------------------------------------
  // Start of instructions
  // ------------------------------------------------------------------------

  // Extract the number of atoms (m) and the number of bonds (n)
  m = mol->m; n = mol->n;

  // Process the molecules one at a time
//#pragma omp for
  for (int k = 0; k < num; k++)
  {
    // Establish a local pointer to the spatial data
    lx = &x[3*m*k];

    // Copy lx into xp1
    #pragma GCC ivdep
    for (int j = 0; j < 3*m; j++) xp1[j] = lx[j];

    // Compute g(x)
    tau = evaluate_constraint_function(mol, lx, g);
#if TAU_OUTPUT
    printf("\n");
    printf("\tit. 0: tau=%e\n", tau);
    // ddm(n, 1, g, n, "%16.8e", " constraint violation\n");
#endif

    for ((*numit)=0; (*numit < maxit) && (tol < tau); (*numit)++)
    {
#if PERF
      start_perf();
#endif

      // Make the lower triangular part of matrix A(x,x)
#if MAKE_MATRIX_OPT
         compute_displacements(mol, lx, displacements);
      make_matrix_opt(mol, &bmatrix, displacements);
#else
      make_matrix(mol, lx, &bmatrix);
#endif
```

```
#if PERF
      stop_perf();
#endif

#if DISPLAY_TIME_SCHUR
      gettimeofday(&start, 0);
#endif

      // Solve the system using the Schur method
      schur_function_sparse(n, bmatrix.xadj, bmatrix.adj, bmatrix.val);

#if DISPLAY_TIME_SCHUR
      gettimeofday(&end, 0);
      total += ((end.tv_sec - start.tv_sec) * 1000000u +
                (end.tv_usec - start.tv_usec)) / 1.e6;
#endif

      // Do the linear update
      apply_jacobian("T", alpha, mol, lx, g, xp1);

      // Manually copy XP1 into X
      #pragma GCC ivdep
      for (int j=0; j < 3*m; j++) lx[j] = xp1[j];

      // Compute g(x)
      tau = evaluate_constraint_function(mol, lx, g);
#if TAU_OUTPUT
      printf("\tit.%2d: tau=%e\n", (*numit)+1, tau);
      // ddm(n, 1, g, n, "%16.8e", " constraint violation\n");
#endif
    }

    // Adjust the return code
    if (tau < tol) rc++; // Increment the number of equations solved.

    // Save the final relative constraint violation for the kth equation
    res[k] = tau;
  }

#if DISPLAY_TIME_SCHUR
   printf("Tiempo secuencial: %f\n", total);
#endif

#if PERF
   print_perf();
   printf("-------------------------------------------------------\n\n");
#endif

  return rc;
}
//---------------------------------------------------------------------------
```

```
int newtonSchurSparse(int num, molecule_t *mol, real *x, real tol, int maxit,
                      int *numit, real *res, int block_size_par)
{
  /* NEWTONSchurSparse: Sparse representation

     Attempts to solve the constraint equations corresponding to NUM copies
     of the same molecule with bond constraints using Newton's method.

     DESCRIPTION OF INTERFACE:

     ON ENTRY:
        num             integer, the number of molecules
        mol->m          integer, the number of atoms
        mol->n          integer, the number of bonds
        mol->bonds      pointer to integer, the sequential bond list
        mol->bond_graph pointer to graph, the LOWER triangular bond graph
        x               pointer to real, the list of initial positions
        tol             real, the convergence tolerance
        maxit           integer, the maximum number of iterations
        numit           pointer to real
        res             pointer to real array of length at least NUM

    ON EXIT:
        mol             all fields unchanged
        x               contains the last iterate for all equations
        tol             unchanged
        maxit           unchanged
        numit           number of algorithm iterations executed
        res             res[k] is the final relative constraint violation
                        for the kth equation.

    RETURNS:
        the number of equations solved to the requested tolerance using at
        most MAXIT iterations.
  */

  // ---------------------------------------------------------------------------
  // Declaration of interval variables
  // ---------------------------------------------------------------------------

  // The return code. Here we assume total failure!
  int rc = 0;

  // The number of atoms (m) and the number of bonds (n)
  int m, n;

  // Format specification
  int format = CSC;

  // Local pointer into the spatial data
```

```
  real *lx;

  // Constant needed during the linear updates
  real alpha = -1;

  // Maximum relative bond length violation
  real tau;

#if DISPLAY_TIME_SCHUR
   struct timeval start, end;
   double total = 0;
#endif

  // ------------------------------------------------------------------------
  // Start of instructions
  // ------------------------------------------------------------------------

  // Copy the parameters to the global variables
  block_size = block_size_par;

  // printf("[newtonSchurSparse] block_size=%d\n", block_size);

  // Extract the number of atoms (m) and the number of bonds (n)
  m = mol->m; n = mol->n;

  // Allocate space for values of g
  smalloc_aligned(g, n, AVX_ALIGNMENT);

  // Initialize the matrix using the specificed format
  init_matrix(format, mol->bond_graph, &bmatrix);

  // Allocate space for the NEXT value of X (X PLUS ONE)
  smalloc_aligned(xp1, 3*m, AVX_ALIGNMENT);

  // Prepare variables for the schur function
  schurSize = n/(block_size+1);

   schurMatrix[0] = (real*)calloc(schurSize, sizeof(real));
   schurMatrix[1] = (real*)calloc(schurSize, sizeof(real));
  schurMatrix[2] = (real*)calloc(schurSize, sizeof(real));

  fillin = (real**)malloc(schurSize*sizeof(real*));
  for (int i=0; i < schurSize; i++)
    fillin[i] = (real*)calloc(block_size*2, sizeof(real));

#if MAKE_MATRIX_OPT
  // Allocate space for the displacement vector
  displacements = alloc_displacements(mol);
#endif

  // Process the molecules one at a time
```

```
//#pragma omp for
  for (int k=0; k < num; k++)
  {
    // Establish a local pointer to the spatial data
    lx = &x[3*m*k];

    // Copy lx into xp1
    #pragma GCC ivdep
    for (int j=0; j<3*m; j++) xp1[j] = lx[j];

    // Compute g(x)
    tau = evaluate_constraint_function(mol, lx, g);
#if TAU_OUTPUT
    printf("\n");
    printf("\tit. 0: tau=%e\n", tau);
    // ddm(n, 1, g, n, "%16.8e", " constraint violation\n");
#endif

    for ((*numit)=0; (*numit < maxit) && (tol < tau); (*numit)++)
    {
#if PERF
      start_perf();
#endif

      // Make the lower triangular part of matrix A(x,x)
#if MAKE_MATRIX_OPT
         compute_displacements(mol, lx, displacements);
      make_matrix_opt(mol, &bmatrix, displacements);
#else
      make_matrix(mol, lx, &bmatrix);
#endif

#if PERF
      stop_perf();
#endif

#if DISPLAY_TIME_SCHUR
      gettimeofday(&start, 0);
#endif

      // Solve the system using the Schur method
      schur_function_sparse(n, bmatrix.xadj, bmatrix.adj, bmatrix.val);

#if DISPLAY_TIME_SCHUR
      gettimeofday(&end, 0);
      total += ((end.tv_sec - start.tv_sec) * 1000000u +
                (end.tv_usec - start.tv_usec)) / 1.e6;
#endif

      // Do the linear update
      apply_jacobian("T", alpha, mol, lx, g, xp1);
```

```
      // Manually copy XP1 into X
      #pragma GCC ivdep
      for (int j=0; j < 3*m; j++) lx[j] = xp1[j];

      // Compute g(x)
      tau = evaluate_constraint_function(mol, lx, g);
#if TAU_OUTPUT
      printf("\tit.%2d: tau=%e\n", (*numit)+1, tau);
      // ddm(n, 1, g, n, "%16.8e", " constraint violation\n");
#endif
    }

    // Adjust the return code
    if (tau < tol) rc++; // Increment the number of equations solved.

    // Save the final relative constraint violation for the kth equation
    res[k] = tau;
  }
  //---------------------------------------------------------------------------
  // Release the memory directly allocated by the subroutine.
  //---------------------------------------------------------------------------
  free(xp1); free(g);

  // Release the memory allocated by siblings
  free_matrix(&bmatrix);

   free(schurMatrix[0]);
   free(schurMatrix[1]);
   free(schurMatrix[2]);

  for (int i=0; i < schurSize; i++) free(fillin[i]);
  free(fillin);

#if MAKE_MATRIX_OPT
  free_displacements(displacements);
#endif

#if DISPLAY_TIME_SCHUR
   printf("Tiempo secuencial: %f\n", total);
#endif

#if PERF
   print_perf();
   printf("-------------------------------------------------------\n\n");
#endif

  return rc;
}
//---------------------------------------------------------------------------
```

B.10. schurSparseVec.h

```
/* SCHUR_SPARSE, version 1.0.0

   Author:
    Rubn  Langarita Bentez,
   Student at Zaragoza University
   Email: 621272@unizar.es

   Date: November 20th, 2017

   A module for doing solving linear systems which are sparse and tridiagonal
   using Schur complement method and vectorization.
*/
#ifndef SCHUR_SPARSE_PAR_NEWTON
#define SCHUR_SPARSE_PAR_NEWTON

int
newtonSchurSparseVec(int num, molecule_t *mol, real *x, real tol, int maxit, int *numit,

int
newtonSchurSparsePar(int num, molecule_t *mol, real *x, real tol, int maxit, int *numit,

void
newtonSchurSparseVec_init(molecule_t *mol, int block_size);

void
newtonSchurSparseVec_free();

int
newtonSchurSparseVec_kernel(int num, molecule_t *mol, real *x, real tol, int maxit,
                  int *numit, real *res, int block_size);

void newtonSchurSparsePar_init(molecule_t *mol, int block_size);

int newtonSchurSparsePar_kernel(int num, molecule_t *mol, real *x, real tol, int maxit,
    int *numit, real *res, int block_size);

void newtonSchurSparsePar_free();

#endif
```

B.11. schurSparseVec.c

```
/* SCHUR

   Author:
    Rubn  Langarita Bentez,
```

```
   Student at Zaragoza University
   Email: 621272@unizar.es

    A module for doing solving linear systems which are sparse and tridiagonal
    using Schur complement method and vectorization.
*/

// Standard libraries
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <omp.h>

// Project specific libraries
#include "precision.h"
#include "vectorization.h"
#include "md-newton.h"
#include "smalloc.h"

real *fillLeftVec;
real *fillRightVec;
int *xadjVec;
real *valVec;
real *gVec;
real **displacements;
real *g;
matrix_t b;
real *xp1;

real *fillLeftPar[NTHREADS];
real *fillRightPar[NTHREADS];
real *valVecPar[NTHREADS];
real *gVecPar[NTHREADS];
pthread_mutex_t mutex[NTHREADS+1];

//---------------------------------------------------------------------------

/* Factorize all blocks using vectorization.
    It does not work with no tridiagonal matrices */
static void
factorizationVec(int block_size, real * restrict valV)
{
  real factor;

  for(int i = 0; i < block_size - 1; i++)
  {
      #pragma GCC ivdep
      for (int j = 0; j < VLEN; j++)
    {
        factor = valV[xadjVec[i] + 2*VLEN + j] / valV[xadjVec[i] + VLEN + j];
```

```
        valV[xadjVec[i ] + 2*VLEN + j] = factor;
        valV[xadjVec[i+1] +  VLEN + j] -= valV[xadjVec[i+1]+j]*factor;
      }
  }
}
//---------------------------------------------------------------------------

static void
forwardLUVec(real * fill, int block_size, int * restrict xadjV, real * restrict valV)
{
   /* Solves Lx = f using forward substitution */

  for (int i = 1; i < block_size; i++)
  {
      #pragma GCC ivdep
      for (int j = 0; j < VLEN; j++)
    {
        fill[i*VLEN + j] -= valV[xadjV[i-1] + 2*VLEN + j]*fill[(i-1)*VLEN + j];
      }
  }
}
//---------------------------------------------------------------------------

static void
backwardLUVec(real * fill, int block_size, int * restrict xadjV, real * restrict valV)
{
   /* Solves Ux = f using backward substitution */

   #pragma GCC ivdep
   for (int i = 0; i < VLEN; i++)
  {
      fill[(block_size-1)*VLEN + i] /= valV[xadjV[block_size-1] + VLEN + i];
   }
   for (int i = block_size - 2; i >= 0; i--)
  {
      #pragma GCC ivdep
      for (int j = 0; j < VLEN; j++)
    {
         fill[i*VLEN + j] =
      (fill[i*VLEN + j] - valV[xadjV[i+1] + j]*fill[(i+1)*VLEN + j])/valV[xadjV[i] + VLEN
      }
  }
}
//---------------------------------------------------------------------------

/* Solves LUx = f using backward and forward substitution */
static void
solveLUVec(real * fill, int block_size, int * restrict xadjV, real * restrict valV)
{
  forwardLUVec(fill, block_size, xadjV, valV);
  backwardLUVec(fill, block_size, xadjV, valV);
```

```
}
//---------------------------------------------------------------------------

static void
step1SparseVec(int block_size, real * restrict gSchurVec, real schurMatrix[][3])
{
   /* Calculate fillins and schur complements for a given block of the matrix.
       It only work with tridiagonal systems.

    DESCRIPTION OF INTERFACE:

      ON ENTRY:
           block_size     Size of the blocks
           gSchurVec     Vector g, independent term for the schur matrix
           schurMatrix    Schur complements matrix

      ON ENTRY AS GLOBAL VARIABLES:
           fillLeftVec       Pointer to the left fillin of the block
           fillRightVec      Pointer to the right fillin of the block
           xadjVec         XADJ[j] is the index in ADJ of the first element
                                of the jth column of the matrix
           valVec          Pointer to the list on nonzero elements of the matrix
           gVec           Vector g, independent term for the blocks

      ON EXIT AS GLOBAL VARIABLES:
           fillLeftVec
           fillRightVec      Fillins updated according to the step1
          schurMatrix    Entries corresponding to the block updated
                                 according to the step1
           gVec
           gSchurVec     Vector g modified acording to the step1

  */

  factorizationVec(block_size, valVec);

  solveLUVec(fillLeftVec, block_size, xadjVec, valVec);
  solveLUVec(fillRightVec, block_size, xadjVec, valVec);
  solveLUVec(gVec, block_size, xadjVec, valVec);

   #pragma GCC ivdep
   for (int i = 0; i < VLEN; i++)
  {
     schurMatrix[i][1] -= valVec[xadjVec[0] + i]*fillLeftVec[i];
     schurMatrix[i][2] -= valVec[xadjVec[0] + i]*fillRightVec[i];
     gSchurVec[i]     -= valVec[xadjVec[0] + i]*gVec[i];
   }

   #pragma GCC ivdep
   for (int i = 0; i < VLEN; i++)
  {
```

```
    schurMatrix[i ][0] -= valVec[xadjVec[block_size-1] + 2*VLEN + i]* fillLeftVec[(block_
    schurMatrix[i+1][1] -= valVec[xadjVec[block_size-1] + 2*VLEN + i]* fillRightVec[(bloc
    gSchurVec[i+1]    -= valVec[xadjVec[block_size-1] + 2*VLEN + i]* gVec[(block_size-1)*
  }
}
//---------------------------------------------------------------------------

static void
step1SparsePar(int block_size, real * restrict valVP, real * restrict gVP, real * restric
               real schurMatrix[][3], real * restrict fillLeftP, real * restrict fillRight
               pthread_mutex_t *mutexSchurDown)
{
  /* Calculate fillins and schur complements for a given block of the matrix.
      It only work with tridiagonal systems.

   DESCRIPTION OF INTERFACE:

     ON ENTRY:
          fillLeftP       Pointer to the left fillin of the block
          fillRightP      Pointer to the right fillin of the block
          xadjVec         XADJ[j] is the index in ADJ of the first element
                                of the jth column of the matrix
          valVP          Pointer to the list on nonzero elements of the matrix
          block_size      Size of the blocks
          gVP            Vector g, independent term for the blocks
          gSchurVec      Vector g, independent term for the schur matrix
          schurMatrix     Schur complements matrix

     ON EXIT
          fillLeftP
          fillRightP      Fillins updated according to the step1
        schurMatrix     Entries corresponding to the block updated
                                according to the step1
          gVP
          gSchurVec      Vector g modified acording to the step1

  */

  factorizationVec(block_size, valVP);

  solveLUVec(fillLeftP, block_size, xadjVec, valVP);
  solveLUVec(fillRightP, block_size, xadjVec, valVP);
  solveLUVec(gVP, block_size, xadjVec, valVP);

   pthread_mutex_lock(mutexSchurUp);
   #pragma GCC ivdep
   for (int i = 0; i < VLEN; i++)
  {
    schurMatrix[i][1] -= valVP[xadjVec[0]+i]*fillLeftP[i];
    schurMatrix[i][2] -= valVP[xadjVec[0]+i]*fillRightP[i];
    gSchurVec[i] -= valVP[xadjVec[0]+i]*gVP[i];
```

```
   }
   pthread_mutex_unlock(mutexSchurUp);

   pthread_mutex_lock(mutexSchurDown);
   #pragma GCC ivdep
   for (int i = 0; i < VLEN; i++)
  {
     schurMatrix[i][0] -= valVP[xadjVec[block_size-1] + 2*VLEN + i]*fillLeftP[(block_size-
     schurMatrix[i+1][1] -= valVP[xadjVec[block_size-1] + 2*VLEN + i]*fillRightP[(block_si
     gSchurVec[i+1] -= valVP[xadjVec[block_size-1]+2*VLEN+i]*gVP[(block_size-1)*VLEN + i];
   }
   pthread_mutex_unlock(mutexSchurDown);
}
//---------------------------------------------------------------------------

/* Factorize the schur complements matrix */
static void
factorizationSchurVec(real schurMatrix[][3], int schur_size)
{
  real factor;

  for(int i = 1; i < schur_size - 1; i++)
  {
    factor = schurMatrix[i][0]/schurMatrix[i][1];
    schurMatrix[i][0] = factor;
    schurMatrix[i+1][1] -= schurMatrix[i][2]*factor;
  }
}
//---------------------------------------------------------------------------

/* Solves Lx = f using forward substitution */
static void
forwardLUSchurVec(real * restrict gSchurVec, real schurMatrix[][3], int schur_size)
{
  for (int i = 2; i < schur_size; i++)
  {
    gSchurVec[i] -= schurMatrix[i-1][0]*gSchurVec[i-1];
  }
}
//---------------------------------------------------------------------------

/* Solves Ux = f using backward substitution */
static void
backwardLUSchurVec(real * restrict gSchurVec, real schurMatrix[][3], int schur_size)
{
  gSchurVec[schur_size - 1] /= schurMatrix[schur_size - 1][1];
  for (int i = schur_size - 2; i > 0; i--)
  {
    gSchurVec[i] = (gSchurVec[i] - schurMatrix[i][2]*gSchurVec[i+1])/schurMatrix[i][1];
  }
}
```

```
//------------------------------------------------------------------------

static void
step2SparseVec(real * restrict gSchurVec, real schurMatrix[][3], int schur_size)
{
   /* Solve the schur complements matrix.
       Factorize the schur complements matrix and solves the system LUx = f
       using backward and forward substitution. */

  factorizationSchurVec(schurMatrix, schur_size);
  forwardLUSchurVec(gSchurVec, schurMatrix, schur_size);
  backwardLUSchurVec(gSchurVec, schurMatrix, schur_size);
}
//------------------------------------------------------------------------

static void
step3SparseVec(int block_size, real * restrict gV, real * restrict gSchurVec,
              real * restrict fillLeftV, real * restrict fillRightV)
{
   /* Apply Gaussian elimination to all the fillins */
   for (int i = 0; i < block_size; i++)
  {
      #pragma GCC ivdep
      for (int j = 0; j < VLEN; j++)
    {
        gV[i*VLEN+j] -= gSchurVec[j]*fillLeftV[i*VLEN+j] + gSchurVec[j+1]*fillRightV[i*VL
      }
   }
}
//------------------------------------------------------------------------

void
newtonSchurSparseVec_init(molecule_t *mol, int block_size)
{
  // The number of atoms (m) and the number of bonds (n)
  int m, n;

  // Format specification
  int format = CSC;

  // Extract the number of atoms (m) and the number of bonds (n)
  m = mol->m; n = mol->n;

  // Allocate space for values of g
  smalloc_aligned(g, n, AVX_ALIGNMENT);

  // Initialize the matrix using the specificed format
  init_matrix(format, mol->bond_graph, &b);

  // Allocate space for the NEXT value of X (X PLUS ONE)
  smalloc_aligned(xp1, 3*m, AVX_ALIGNMENT);
```

```
  // Allocate space for schur variables
  smalloc_aligned(fillLeftVec, block_size*VLEN, AVX_ALIGNMENT);
  smalloc_aligned(fillRightVec, block_size*VLEN, AVX_ALIGNMENT);
  smalloc_aligned(xadjVec, block_size+1, AVX_ALIGNMENT);
  smalloc_aligned(valVec, b.nnz + 2 - 3*(VLEN-1), AVX_ALIGNMENT);
  smalloc_aligned(gVec, block_size*VLEN, AVX_ALIGNMENT);

#if MAKE_MATRIX_OPT
  // Allocate space for the displacement vector
  displacements = alloc_displacements(mol);
#endif
}
//------------------------------------------------------------------------

void
newtonSchurSparseVec_free()
{
   free_matrix(&b);
   free(g);
   free(xp1);
   free(fillLeftVec);
   free(fillRightVec);
   free(xadjVec);
   free(valVec);
   free(gVec);

#if MAKE_MATRIX_OPT
   free_displacements(displacements);
#endif
}
//------------------------------------------------------------------------

int
newtonSchurSparseVec_kernel(int num, molecule_t *mol, real *x, real tol, int maxit,
                            int *numit, real *res, int block_size)
{
  /* NEWTONSchurSparseVec: Sparse representation and vectorized

     Attempts to solve the constraint equations corresponding to NUM copies
     of the same molecule with bond constraints using Newton's method and
       using vectorization.

     DESCRIPTION OF INTERFACE:

     ON ENTRY:
        num             integer, the number of molecules
        mol->m          integer, the number of atoms
        mol->n          integer, the number of bonds
        mol->bonds      pointer to integer, the sequential bond list
        mol->bond_graph pointer to graph, the LOWER triangular bond graph
```

```
        x               pointer to real, the list of initial positions
        tol             real, the convergence tolerance
        maxit           integer, the maximum number of iterations
        numit           pointer to real
        res             pointer to real array of length at least NUM

    ON EXIT:
        mol             all fields unchanged
        x               contains the last iterate for all equations
        tol             unchanged
        maxit           unchanged
        numit           number of algorithm iterations executed
        res             res[k] is the final relative constraint violation
                        for the kth equation.

    RETURNS:
        the number of equations solved to the requested tolerance using at
        most MAXIT iterations.
  */

  // ------------------------------------------------------------------------
  // Declaration of interval variables
  // ------------------------------------------------------------------------

  // The return code. Here we assume total failure!
  int rc = 0;

  // The number of atoms (m) and the number of bonds (n)
  int m;

  // Local pointer into the spatial data
  real *lx;

  // Constant needed during the linear updates
  real alpha = -1;

  // Maximum relative bond length violation
  real tau;

  // schur variables
  real schurMatrix[VLEN+1][3] __attribute__ ((aligned (AVX_ALIGNMENT)));
  real gSchurVec[VLEN+1] __attribute__ ((aligned (AVX_ALIGNMENT)));

  // Extract the number of atoms (m) and the number of bonds (n)
  m = mol->m;

#if DISPLAY_TIME_SCHUR
   struct timeval start,end;
   double total = 0;
#endif
```

```
  // ------------------------------------------------------------------------
  // Start of instructions
  // ------------------------------------------------------------------------

  // Process the molecules one at a time
  for (int k=0; k<num; k++)
  {
    // Establish a local pointer to the spatial data
    lx = &x[3*m*k];

    // Copy lx into xp1
     memcpy(xp1, lx, 3*m*sizeof(real));

    // Compute g(x)
    tau = evaluate_constraint_function(mol, lx, g);
#if TAU_OUTPUT
    printf("\n");
    printf("\tit. 0: tau=%e\n", tau);
    // ddm(n, 1, g, n, "%16.8e", " constraint violation\n");
#endif

    for ((*numit)=0; (*numit < maxit) && (tol < tau); (*numit)++)
    {
#if PERF
      start_perf();
#endif
      // Make the lower triangular part of matrix A(x,x)
#if MAKE_MATRIX_OPT
         compute_displacements(mol, lx, displacements);
      make_matrix_opt(mol, &b, displacements);
#else
      make_matrix(mol, lx, &b);
#endif
#if PERF
      stop_perf();
#endif

      fillVec(b.xadj, b.val, g, xadjVec, valVec, gVec, gSchurVec, schurMatrix,
             fillLeftVec, fillRightVec, block_size);

#if DISPLAY_TIME_SCHUR
      gettimeofday(&start,0);
#endif

      step1SparseVec(block_size, gSchurVec, schurMatrix);
      step2SparseVec(gSchurVec, schurMatrix, VLEN);
      step3SparseVec(block_size, gVec, gSchurVec, fillLeftVec, fillRightVec);

#if DISPLAY_TIME_SCHUR
      gettimeofday(&end, 0);
      total += ((end.tv_sec - start.tv_sec) * 1000000u +
```

```
              (end.tv_usec - start.tv_usec)) / 1.e6;
#endif

      // Do the linear update
      apply_jacobian_Svec(alpha, mol, lx, gSchurVec, xp1, block_size, VLEN);
      apply_jacobian_vec(alpha, mol, lx, gVec, xp1, block_size, 0);

      // Copy XP1 into X
      memcpy(lx, xp1, 3*m*sizeof(real));

      // Compute g(x)
      tau = evaluate_constraint_function(mol, lx, g);
#if TAU_OUTPUT
      printf("\tit.%2d: tau=%e\n", (*numit)+1, tau);
      // ddm(n, 1, g, n, "%16.8e", " constraint violation\n");
#endif
    }

    // Adjust the return code
    if (tau < tol) rc++; // Increment the number of equations solved.

    // Save the final relative constraint violation for the kth equation
    res[k] = tau;
  }


#if DISPLAY_TIME_SCHUR
   printf("Tiempo vectorial: %f\n", total);
#endif

#if PERF
   print_perf();
   printf("-------------------------------------------------------\n\n");
#endif

  return rc;
}
//---------------------------------------------------------------------------

int
newtonSchurSparseVec(int num, molecule_t *mol, real *x, real tol, int maxit,
                     int *numit, real *res, int block_size)
{
  /* NEWTONSchurSparseVec: Sparse representation and vectorized

     Attempts to solve the constraint equations corresponding to NUM copies
     of the same molecule with bond constraints using Newton's method and
       using vectorization.

     DESCRIPTION OF INTERFACE:
```

```
     ON ENTRY:
        num             integer, the number of molecules
        mol->m          integer, the number of atoms
        mol->n          integer, the number of bonds
        mol->bonds      pointer to integer, the sequential bond list
        mol->bond_graph pointer to graph, the LOWER triangular bond graph
        x               pointer to real, the list of initial positions
        tol             real, the convergence tolerance
        maxit           integer, the maximum number of iterations
        numit           pointer to real
        res             pointer to real array of length at least NUM

     ON EXIT:
        mol             all fields unchanged
        x               contains the last iterate for all equations
        tol             unchanged
        maxit           unchanged
        numit           number of algorithm iterations executed
        res             res[k] is the final relative constraint violation
                        for the kth equation.

     RETURNS:
        the number of equations solved to the requested tolerance using at
        most MAXIT iterations.
   */

   // ------------------------------------------------------------------------
   // Declaration of interval variables
   // ------------------------------------------------------------------------

   // The return code. Here we assume total failure!
   int rc = 0;

   newtonSchurSparseVec_init(mol, block_size);
   rc = newtonSchurSparseVec_kernel(num, mol, x, tol, maxit, numit, res, block_size);
   newtonSchurSparseVec_free();

   return rc;
}
//--------------------------------------------------------------------------

void
newtonSchurSparsePar_init(molecule_t *mol, int block_size)
{
   // The number of atoms (m) and the number of bonds (n)
   int m, n;

   // Format specification
   int format = CSC;

    init_matrix(format, mol->bond_graph, &b);
```

```
   // Extract the number of atoms (m) and the number of bonds (n)
   m = mol->m; n = mol->n;

  // Allocate space for values of g
  smalloc_aligned(g, n, AVX_ALIGNMENT);

  // Allocate space for the NEXT value of X (X PLUS ONE)
  smalloc_aligned(xp1, 3*m, AVX_ALIGNMENT);

  xadjVec = (int*)malloc((block_size+1)*sizeof(real));

  for(int t = 0; t < NTHREADS; t++)
  {
    smalloc_aligned(fillLeftPar[t], block_size*VLEN, AVX_ALIGNMENT);
     smalloc_aligned(fillRightPar[t], block_size*VLEN, AVX_ALIGNMENT);
     smalloc_aligned(valVecPar[t], b.nnz + 2 - 3*(VLEN*NTHREADS-1), AVX_ALIGNMENT);
     smalloc_aligned(gVecPar[t], block_size*VLEN, AVX_ALIGNMENT);
  }

#if MAKE_MATRIX_OPT
  // Allocate space for the displacement vector
  displacements = alloc_displacements(mol);
#endif

   for (int l = 0; l < NTHREADS + 1; l++)
   {
      pthread_mutex_init(&mutex[l], 0);
   }
}
//---------------------------------------------------------------------------

int
newtonSchurSparsePar_kernel(int num, molecule_t *mol, real *x, real tol, int maxit,
                            int *numit, real *res, int block_size)
{
  /* NEWTONSchurSparsePar: Sparse representation parallel version

     Attempts to solve the constraint equations corresponding to NUM copies
     of the same molecule with bond constraints using Newton's method.

     DESCRIPTION OF INTERFACE:

     ON ENTRY:
        num             integer, the number of molecules
        mol->m          integer, the number of atoms
        mol->n          integer, the number of bonds
        mol->bonds      pointer to integer, the sequential bond list
        mol->bond_graph pointer to graph, the LOWER triangular bond graph
        x               pointer to real, the list of initial positions
        tol             real, the convergence tolerance
```

```
        maxit           integer, the maximum number of iterations
        numit           pointer to real
        res             pointer to real array of length at least NUM

    ON EXIT:
        mol             all fields unchanged
        x               contains the last iterate for all equations
        tol             unchanged
        maxit           unchanged
        numit           number of algorithm iterations executed
        res             res[k] is the final relative constraint violation
                        for the kth equation.

    RETURNS:
        the number of equations solved to the requested tolerance using at
        most MAXIT iterations.
  */

  // ------------------------------------------------------------------------
  // Declaration of interval variables
  // ------------------------------------------------------------------------

  // The return code. Here we assume total failure!
  int rc = 0;

  // The number of atoms (m) and the number of bonds (n)
  // int m;
  int m;

  // Local pointer into the spatial data
  real *lx;

  // Constant needed during the linear updates
  real alpha = -1;

  // Maximum relative bond length violation
  real tau;

  // schur variables
   real schurMatrix[(VLEN+1)*NTHREADS][3] __attribute__ ((aligned (AVX_ALIGNMENT)));
   real gSchurVec[VLEN*NTHREADS + 1] __attribute__ ((aligned (AVX_ALIGNMENT)));

   // Extract the number of atoms (m) and the number of bonds (n)
   m = mol->m;

#if DISPLAY_TIME_SCHUR
   struct timeval start,end;
   double total = 0;
#endif

  // ------------------------------------------------------------------------
```

```
  // Start of instructions
  // ------------------------------------------------------------------------

  // set the number of threads
  omp_set_num_threads(NTHREADS);

  // Process the molecules one at a time
  for (int k=0; k < num; k++)
  {
    // Establish a local pointer to the spatial data
    lx = &x[3*m*k];

    // Copy lx into xp1
     memcpy(xp1, lx, 3*m*sizeof(real));

    // Compute gSchurVec(x)
    tau = evaluate_constraint_function_Svec(mol, lx, gSchurVec, block_size, VLEN*NTHREADS)

    // Compute tau
    #pragma omp parallel for schedule(static,1) default(shared) reduction(max:tau)
    for(int l=0; l < NTHREADS; l++)
    {
      int offset = l*(block_size*VLEN + (VLEN-1)) + l;
      tau = evaluate_constraint_function_vec(mol, lx, gVecPar[l], block_size, offset);
    }
#if TAU_OUTPUT
    printf("\n");
    printf("\tit. 0: tau=%e\n", tau);
    // ddm(n, 1, g, n, "%16.8e", " constraint violation\n");
#endif

    for ((*numit)=0; (*numit < maxit) && (tol < tau); (*numit)++)
    {
#if MAKE_MATRIX_OPT
        compute_displacements(mol, lx, displacements);
      make_matrix_Svec_opt(mol, &b, xadjVec, schurMatrix, block_size, VLEN*NTHREADS, displ
#else
      make_matrix_Svec(mol, lx, &b, xadjVec, schurMatrix, block_size, VLEN*NTHREADS);
#endif

#if DISPLAY_TIME_SCHUR
      gettimeofday(&start,0);
#endif

       // reset value from previous iteration
      tau = 0;

      #pragma omp parallel for schedule(static,1) default(shared) reduction(max:tau)
      for(int l = 0; l < NTHREADS; l++)
      {
        real *schurMatrixAux;
```

```
        int isFirst = 0, isLast = 0;
        if (l == 0) isFirst = 1;
        if (l == NTHREADS - 1) isLast = 1;

        int offset = l*(block_size*VLEN + (VLEN-1)) + l;

#if MAKE_MATRIX_OPT
      compute_displacements(mol, lx, displacements);
        make_matrix_vec_opt(mol, &b, valVecPar[l], xadjVec, fillLeftPar[l], fillRightPar[l
#else
        make_matrix_vec(mol, lx, &b, valVecPar[l], xadjVec, fillLeftPar[l], fillRightPar[l
#endif

        schurMatrixAux = schurMatrix[l*VLEN];
        step1SparsePar(block_size, valVecPar[l], gVecPar[l], &gSchurVec[l*VLEN],
                       (real (*)[3]) schurMatrixAux, fillLeftPar[l],
                       fillRightPar[l], &mutex[l], &mutex[l+1]);
      }

      step2SparseVec(gSchurVec, schurMatrix, VLEN*NTHREADS);

      #pragma omp parallel for schedule(static,1) default(shared)
      for(int l = 0; l < NTHREADS; l++)
      {
        step3SparseVec(block_size, gVecPar[l], &gSchurVec[l*VLEN], fillLeftPar[l], fillRig
        int offset = l*(block_size*VLEN + (VLEN-1)) + l;
        apply_jacobian_vec(alpha, mol, lx, gVecPar[l], xp1, block_size, offset);
      }

  #if DISPLAY_TIME_SCHUR
      gettimeofday(&end,0);
      total += ((end.tv_sec - start.tv_sec) * 1000000u +
         (end.tv_usec - start.tv_usec)) / 1.e6;
  #endif

      // Do the linear update
      apply_jacobian_Svec(alpha, mol, lx, gSchurVec, xp1, block_size, VLEN*NTHREADS);

      // Copy XP1 into X
      memcpy(lx, xp1, 3*m*sizeof(real));

      tau = evaluate_constraint_function_Svec(mol, lx, gSchurVec, block_size, VLEN*NTHREAD

      #pragma omp parallel for schedule(static,1) default(shared) reduction(max:tau)
      for(int l = 0; l < NTHREADS; l++)
      {
        int offset = l*(block_size*VLEN + (VLEN-1)) + l;
        tau = evaluate_constraint_function_vec(mol, lx, gVecPar[l], block_size, offset);
#if TAU_OUTPUT
        printf("\tit.%2d: tau=%e (thread %d)\n", (*numit)+1, tau, omp_get_thread_num());
        // ddm(n, 1, g, n, "%16.8e", " constraint violation\n");
```

```
#endif
      }
#if TAU_OUTPUT
      printf("\tit.%2d: tau=%e\n", (*numit)+1, tau);
      // ddm(n, 1, g, n, "%16.8e", " constraint violation\n");
#endif
    }

    // Adjust the return code
    if (tau < tol) rc++; // Increment the number of equations solved.

    // Save the final relative constraint violation for the kth equation
    res[k] = tau;
  }

#if DISPLAY_TIME_SCHUR
   printf("Tiempo paralelo: %f\n", total);
#endif

  return rc;
}
//---------------------------------------------------------------------------

void
newtonSchurSparsePar_free()
{
   free_matrix(&b);

   free(g);
   free(xp1);
   free(xadjVec);

   for(int t = 0; t < NTHREADS; t++)
   {
      free(fillLeftPar[t]);
      free(fillRightPar[t]);
      free(valVecPar[t]);
      free(gVecPar[t]);
   }

#if MAKE_MATRIX_OPT
   free_displacements(displacements);
#endif

   for (int l = 0; l < NTHREADS + 1; l++)
      pthread_mutex_destroy(&mutex[l]);

}
//---------------------------------------------------------------------------

int
```

```
newtonSchurSparsePar(int num, molecule_t *mol, real *x, real tol, int maxit,
                     int *numit, real *res, int block_size)
{
  /* NEWTONSchurSparsePar: Sparse representation parallel version

     Attempts to solve the constraint equations corresponding to NUM copies
     of the same molecule with bond constraints using Newton's method.

     DESCRIPTION OF INTERFACE:

     ON ENTRY:
        num             integer, the number of molecules
        mol->m          integer, the number of atoms
        mol->n          integer, the number of bonds
        mol->bonds      pointer to integer, the sequential bond list
        mol->bond_graph pointer to graph, the LOWER triangular bond graph
        x               pointer to real, the list of initial positions
        tol             real, the convergence tolerance
        maxit           integer, the maximum number of iterations
        numit           pointer to real
        res             pointer to real array of length at least NUM

    ON EXIT:
        mol             all fields unchanged
        x               contains the last iterate for all equations
        tol             unchanged
        maxit           unchanged
        numit           number of algorithm iterations executed
        res             res[k] is the final relative constraint violation
                        for the kth equation.

    RETURNS:
        the number of equations solved to the requested tolerance using at
        most MAXIT iterations.
  */

  // ---------------------------------------------------------------------
  // Declaration of interval variables
  // ---------------------------------------------------------------------

  // The return code. Here we assume total failure!
  int rc = 0;

  newtonSchurSparsePar_init(mol, block_size);
  rc = newtonSchurSparsePar_kernel(num, mol, x, tol, maxit, numit, res, block_size);
  newtonSchurSparsePar_free();

  return rc;
}
//---------------------------------------------------------------------
```

B.12. shake.h

```
//------------------------------------------------------------------------
//-- DUMMY STATEMENT 78 CHARACTERS LONG TO ENSURE THE LINES ARE NOT TOO LONG -
//------------------------------------------------------------------------

#ifndef SHAKE
#define SHAKE

// Modules written specifically for the MD project
#include "md-sparse.h"


// Standard libraries
#include <math.h>
#include <malloc.h>

// ------------------------------------------------------------------------
// COMPILE TIME CONSTANTS
// ------------------------------------------------------------------------

//------------------------------------------------------------------------
// DATA STRUCTURES
//------------------------------------------------------------------------

//------------------------------------------------------------------------
// SUBROUTINE PROTOTYPES WITH BRIEF DESCRIPTION
//------------------------------------------------------------------------

void shake_init(molecule_t *mol);

int shake_kernel(int num, molecule_t *mol, real *x, real *x_initial, real tol,
              int maxit, int *numit, real *res);

void shake_free();

// shake algorithm
int
shake(int num, molecule_t *mol, real *x, real *x_initial, real tol, int maxit, int *numit

#endif // SHAKE
```

B.13. shake.c

```
//------------------------------------------------------------------------
//-- DUMMY STATEMENT 78 CHARACTERS LONG TO ENSURE THE LINES ARE NOT TOO LONG -
//------------------------------------------------------------------------
```

```c
/* SHAKE

   Author:
    Rubn  Langarita Bentez,
   Student at Zaragoza University
   Email: 621272@unizar.es

*/

#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <pthread.h>
#include <time.h>
#include <sys/time.h>

#include "md-newton.h"
#include "smalloc.h"
#include "aux.h"
#include "md-sparse.h"
#include "shake.h"
#include "perf.h"

//--------------------------------------------------------------------------
// DECLARATION OF GLOBAL VARIABLES
//--------------------------------------------------------------------------

static real *initialDisplacements; // Tables of positions
static real *halfOfReducedMass; // 1/(2/mi+2/mj) for each constraint

//--------------------------------------------------------------------------

void
shake_init(molecule_t *mol)
{
  int nConstrains = mol->n, nAtoms = mol->m; // Number of constraints and atoms

  // Allocate space
  // tables of positions
  smalloc_aligned(initialDisplacements, 3*nAtoms, AVX_ALIGNMENT);
  // 1/(2/mi+2/mj) for each constraint
  smalloc_aligned(halfOfReducedMass, nConstrains, AVX_ALIGNMENT);

  // Initialize halfOfReducedMass table
   for (int j = 0; j < nConstrains; j++)
   {
      int a = mol->bonds[2*j];
      int b = mol->bonds[2*j+1]; // Atoms involved in the constraint
      halfOfReducedMass[j] = 1 / (2*mol->invmass[a] + 2*mol->invmass[b]);
   }
}
```

```
//------------------------------------------------------------------------

int shake_kernel(int num, molecule_t *mol, real *x, real *x_initial, real tol,
              int maxit, int *numit, real *res)
{
     /*
        num                        Number of molecules to process
        mol->bonds             Atoms involved in each constraint
        mol->invmass           Inverse of each atom mass
        mol->sigma2            Objective value for each constraint
        x                          Actual positions to correct (input and output)
        x_initial                Positions on the previous step
        tol                        Tolerance for the constraints
        maxit                    Maximum iterations permitted
        rel                        Relative error for each constraint (output)
     */

  // The return code. Here we assume total failure!
  int rc = 0;
  int nConstrains = mol->n, nAtoms = mol->m; // Number of constraints and atoms
   real *lx, *lx_initial;

   int error = 0, converged = 0; // Stop condition for the loop

  // Initialize the largest relative error
   real rel = 0;

  // Process the molecules one at a time
  for (int k=0; k < num; k++)
  {
      error = 0;
      converged = 0;

      lx = &x[3*nAtoms*k]; // Establish local pointers
      lx_initial = &x_initial[3*nAtoms*k];

    // Initialize the initial displacements
      for (int j = 0; j < nConstrains; j++)
      {
         int a = mol->bonds[2*j];
         int b = mol->bonds[2*j+1]; // Atoms involved in the constraint

         initialDisplacements[3*j+XX] = lx_initial[3*a+XX] - lx_initial[3*b+XX];
         initialDisplacements[3*j+YY] = lx_initial[3*a+YY] - lx_initial[3*b+YY];
         initialDisplacements[3*j+ZZ] = lx_initial[3*a+ZZ] - lx_initial[3*b+ZZ];
      }

#if TAU_OUTPUT
        printf("\n");
#endif
```

```
// Stop when maxit reached or a stop condition
for (*numit=0; (*numit < maxit*10) && (error == 0) && (converged == 0); (*numit)++)
{
     converged = 1;
  // reset constraint violation value
  rel = 0;

     // Loop over constraints
     for (int j = 0; j < nConstrains; j++)
     {
        int a = mol->bonds[2*j];
        int b = mol->bonds[2*j+1]; // Atoms involved in the constraint

        real rx = initialDisplacements[3*j+XX]; // Displacement vector of the constrai
        real ry = initialDisplacements[3*j+YY];
        real rz = initialDisplacements[3*j+ZZ];

        real rPrimeX = lx[3*a+XX] - lx[3*b+XX]; // Displacement vector of the constrai
        real rPrimeY = lx[3*a+YY] - lx[3*b+YY];
        real rPrimeZ = lx[3*a+ZZ] - lx[3*b+ZZ];

    // Displacement squared
        real rPrime2 = rPrimeX*rPrimeX + rPrimeY*rPrimeY + rPrimeZ*rPrimeZ;

    // Objective value for the constraint
        real constraint2 = mol->sigma2[j];
        real diff = constraint2 - rPrime2;
        real convergence = 0.5*myfabs(diff)/constraint2;

    // Check if it is necesary to correct the positions
        if (convergence > tol)
        {
           converged = 0;

      // Scalar product of actual and previous displacements
           real rXrPrime = rPrimeX*rx + rPrimeY*ry + rPrimeZ*rz;

      // Check if the correction is possible
           if (rXrPrime < constraint2 * tol)
           {
              error = 1;
        // printf("\tshake it. %2d: rXrPrime=%e (%e)\n", *numit, rXrPrime, constraint2
           }
           else
           {
              real lagranceMultiplier = diff * halfOfReducedMass[j] / rXrPrime; // Calc

              lx[3*a+XX] += rx*lagranceMultiplier*mol->invmass[a];
              lx[3*a+YY] += ry*lagranceMultiplier*mol->invmass[a];
              lx[3*a+ZZ] += rz*lagranceMultiplier*mol->invmass[a];
```

```
                lx[3*b+XX] -= rx*lagranceMultiplier*mol->invmass[b];
                lx[3*b+YY] -= ry*lagranceMultiplier*mol->invmass[b];
                lx[3*b+ZZ] -= rz*lagranceMultiplier*mol->invmass[b];
              }
            }

            // Update the relative error
          rel = max(rel, convergence);
          // printf("[%d] convergence=%e, rel=%e, \n", i, convergence, rel);
           }
#if TAU_OUTPUT
      //printf("\tit. %2d: tau=%e\n", *numit, rel);
      printf("%e,\n", rel);
#endif
    }
    // Increment the number of equations solved.
    if (rel < tol) rc++;
    // if (error == 0) rc++;
      res[k] = rel;
  }

  return rc;
}
//---------------------------------------------------------------------------

void
shake_free()
{
  //---------------------------------------------------------------------------
  // Release the memory directly allocated by the subroutine and its siblings
  //---------------------------------------------------------------------------

  free(initialDisplacements);
  free(halfOfReducedMass);
}
//---------------------------------------------------------------------------

int shake(int num, molecule_t *mol, real *x, real *x_initial, real tol,
        int maxit, int *numit, real *res)
{
     /*
        num                       Number of molecules to process
        mol->bonds            Atoms involved in each constraint
        mol->invmass          Inverse of each atom mass
        mol->sigma2           Objective value for each constraint
        x                         Actual positions to correct (input and output)
        x_initial               Positions on the previous step
        tol                       Tolerance for the constraints
        maxit                   Maximum iterations permitted
        rel                       Relative error for each constraint (output)
     */
```

```
  int rc = 0;

  shake_init(mol);
  rc = shake_kernel(num, mol, x, x_initial, tol, maxit /* maxit_shake */, numit, res);
  shake_free();
  return rc;
}
```

B.14. vectorization.h

```
//------------------------------------------------------------------------
//-- DUMMY STATEMENT 78 CHARACTERS LONG TO ENSURE THE LINES ARE NOT TOO LONG -
//------------------------------------------------------------------------

/* VECTORIZATION */

#ifndef VECTORIZATION_GUARD
#define VECTORIZATION_GUARD

#include "precision.h"

// Define the alignment used for SIMD operations
#define SSE_ALIGNMENT 16
#define AVX_ALIGNMENT 32
#define AVX512_ALIGNMENT 64

// Assume AVX (256 bits, 32 bytes)
#ifndef VLEN
   #if PRECISION
      #define VLEN   4
   #else
      #define VLEN   8
   #endif
#endif

// Sort structs to vectorize with the schur method
void fillVec(int *xadj, real *val, real *g, int *xadjVec, real *valVec,
   real *gVec, real *gSchurVec, real schurMatrix[][3], real *fillLeft,
   real * fillRight, int block_size);

void fillPar_share(int *xadj, real *val, real *g, int *xadjVec,
   real *gSchurVec, real schurMatrix[][3], int block_size);

void fillPar_private(int *xadj, real *val, real *g, int *xadjVec, real *valVec,
   real *gVec, real *fillLeft, real * fillRight, int block_size,
   int isFirst, int isLast);

// Undo changes of fillVec in vector g
```

```
void unFillVec(real *g, real *gVec, real *gSchurVec, int block_size);

void unFillPar_share(real *g, real *gSchurVec, int block_size);

void unFillPar_private(real *g, real *gVec, int block_size);

#endif
```

B.15. vectorization.c

```
//------------------------------------------------------------------------------
//-- DUMMY STATEMENT 78 CHARACTERS LONG TO ENSURE THE LINES ARE NOT TOO LONG -
//------------------------------------------------------------------------------

/* VECTORIZATION */

#include <stdlib.h>
#include <stdio.h>
#include "md-newton.h"
#include "vectorization.h"

// Sort structs to vectorize with the schur method
void
fillVec(int *xadj, real *val, real *g, int *xadjVec, real *valVec, real *gVec,
        real *gSchurVec, real schurMatrix[][3], real *fillLeft, real *fillRight, int bloc
{
   for(int i = 0; i < block_size + 1; i++)
  {
      xadjVec[i] = i*3*VLEN;
   }

   valVec[xadjVec[0]      ] = 0;
   valVec[xadjVec[0] + VLEN] = val[xadj[0]  ];
   valVec[xadjVec[0] + 2*VLEN] = val[xadj[0] + 1];

   for(int j = 1; j < block_size; j++)
  {
      valVec[xadjVec[j]      ] = val[xadj[j]  ];
      valVec[xadjVec[j] + VLEN] = val[xadj[j] + 1];
      valVec[xadjVec[j] + 2*VLEN] = val[xadj[j] + 2];
   }

   for(int i = 1; i < VLEN - 1; i++)
  {
      for(int j = 0; j < block_size; j++)
    {
        valVec[xadjVec[j] +       i] = val[xadj[j + i*(block_size+1)] ];
        valVec[xadjVec[j] + VLEN + i] = val[xadj[j + i*(block_size+1)] + 1];
        valVec[xadjVec[j] + 2*VLEN + i] = val[xadj[j + i*(block_size+1)] + 2];
```

```
  }
}

 for(int j = 0; j < block_size - 1; j++)
{
    valVec[xadjVec[j] +      (VLEN-1)] =  val[xadj[j + (VLEN-1)*(block_size+1)] ];
    valVec[xadjVec[j] + VLEN +(VLEN-1)] = val[xadj[j + (VLEN-1)*(block_size+1)] + 1];
    valVec[xadjVec[j] + 2*VLEN +(VLEN-1)] = val[xadj[j + (VLEN-1)*(block_size+1)] + 2];
}

 valVec[xadjVec[block_size-1] +      (VLEN-1)] =  val[xadj[(block_size-1) + (VLEN-1)*(bl
 valVec[xadjVec[block_size-1] + VLEN + (VLEN-1)] = val[xadj[(block_size-1) + (VLEN-1)*(b
 valVec[xadjVec[block_size-1] + 2*VLEN + (VLEN-1)] = 0;

 for(int i = 0; i < VLEN; i++)
{
    for(int j = 0; j < block_size; j++)
  {
       gVec[j*VLEN + i] = g[i*(block_size+1) + j];
    }
}

 gSchurVec[0] = 0;
for(int i = 1; i < VLEN; i++)
{
    gSchurVec[i] = g[i*(block_size+1) - 1];
 }
 gSchurVec[VLEN] = 0;

 for(int i = 0; i < VLEN; i++)
{
    schurMatrix[i][0] = 0;
    schurMatrix[i][2] = 0;
}
 schurMatrix[0][1] = 0;
 for(int i = 1; i < VLEN; i++)
{
    schurMatrix[i][1] = val[xadj[i*(block_size+1) - 1] + 1];
 }
 schurMatrix[VLEN][1] = 0;

 fillLeft[0] = 0;
 for(int i = 1; i < VLEN; i++)
{
    fillLeft[i] = val[xadj[i*(block_size+1) - 1] + 2];
 }
 for(int i = VLEN; i < VLEN*block_size; i++)
{
    fillLeft[i] = 0;
}
```

```
   for(int i = 0; i < VLEN*(block_size-1); i++)
  {
     fillRight[i] = 0;
   }
   for(int i = 1; i < VLEN; i++)
  {
     fillRight[(block_size-1)*VLEN + (i-1)] = val[xadj[i*(block_size+1) - 1]];
   }
   fillRight[(block_size-1)*VLEN + (VLEN-1)] = 0;
}

void
fillPar_share(int *xadj, real *val, real *g, int *xadjVec, real *gSchurVec,
            real schurMatrix[][3], int block_size)
{
   for(int i = 0; i < block_size + 1; i++)
  {
     xadjVec[i] = i*3*VLEN;
   }

   gSchurVec[0] = 0;
   for(int i = 1; i < NTHREADS*VLEN; i++)
  {
     gSchurVec[i] = g[i*(block_size+1) - 1];
   }
   gSchurVec[NTHREADS*VLEN] = 0;

   for(int i = 0; i < NTHREADS*VLEN; i++)
  {
     schurMatrix[i][0] = 0;
     schurMatrix[i][2] = 0;
   }
   schurMatrix[0][1] = 0;
   for(int i = 1; i < NTHREADS*VLEN; i++)
  {
     schurMatrix[i][1] = val[xadj[i*(block_size+1) - 1] + 1];
   }
   schurMatrix[NTHREADS*VLEN][1] = 0;
}

void
fillPar_private(int *xadj, real *val, real *g, int *xadjVec, real *valVec,
              real *gVec, real *fillLeft, real * fillRight, int block_size,
                int isFirst, int isLast)
{
   if (isFirst)
  {
     valVec[xadjVec[0]      ] = 0;
     valVec[xadjVec[0] + VLEN] = val[xadj[0]  ];
     valVec[xadjVec[0] + 2*VLEN] = val[xadj[0] + 1];
   }
```

```
else
{
    valVec[xadjVec[0]      ] = val[xadj[0]  ];
    valVec[xadjVec[0] + VLEN] = val[xadj[0] + 1];
    valVec[xadjVec[0] + 2*VLEN] = val[xadj[0] + 2];
 }

 for(int j = 1; j < block_size; j++)
{
    valVec[xadjVec[j]      ] = val[xadj[j]  ];
    valVec[xadjVec[j] + VLEN] = val[xadj[j] + 1];
    valVec[xadjVec[j] + 2*VLEN] = val[xadj[j] + 2];
 }

 for(int i = 1; i < VLEN - 1; i++)
{
    for(int j = 0; j < block_size; j++)
  {
       valVec[xadjVec[j]       + i] = val[xadj[j + i*(block_size+1)] ];
       valVec[xadjVec[j] + VLEN + i] = val[xadj[j + i*(block_size+1)] + 1];
       valVec[xadjVec[j] + 2*VLEN + i] = val[xadj[j + i*(block_size+1)] + 2];
    }
}

 for(int j=0;j<block_size-1;j++)
{
    valVec[xadjVec[j] +       (VLEN-1)] = val[xadj[j + (VLEN-1)*(block_size+1)] ];
    valVec[xadjVec[j] + VLEN + (VLEN-1)] = val[xadj[j + (VLEN-1)*(block_size+1)] + 1];
    valVec[xadjVec[j] + 2*VLEN + (VLEN-1)] = val[xadj[j + (VLEN-1)*(block_size+1)] + 2];
 }
 valVec[xadjVec[block_size-1] +    (VLEN-1)] = val[xadj[(block_size-1) + (VLEN-1)*(bloc
 valVec[xadjVec[block_size-1] + VLEN + (VLEN-1)] = val[xadj[(block_size-1) + (VLEN-1)*(

 if (isLast)
  valVec[xadjVec[block_size-1]+2*VLEN+(VLEN-1)] = 0;
 else
  valVec[xadjVec[block_size-1]+2*VLEN+(VLEN-1)] = val[xadj[(block_size-1) + (VLEN-1)*(bl

 for(int i = 0; i < VLEN; i++)
{
    for(int j = 0; j < block_size; j++)
  {
       gVec[j*VLEN + i] = g[i*(block_size+1) + j];
   }
 }

 if (isFirst)
{
    fillLeft[0] = 0;
    for(int i = 1; i < VLEN; i++)
  {
```

```
        fillLeft[i] = val[xadj[i*(block_size+1) - 1] + 2];
      }
   }
  else
  {
      for(int i = 0; i < VLEN; i++)
    {
        fillLeft[i] = val[xadj[i*(block_size+1) - 1] + 2];
      }
   }
   for(int i = VLEN; i < VLEN*block_size; i++)
  {
      fillLeft[i]=0;
   }

   for(int i = 0; i < VLEN*(block_size-1); i++)
  {
      fillRight[i] = 0;
   }
   for(int i = 0; i < VLEN-1; i++)
  {
      fillRight[(block_size-1)*VLEN + i] = val[xadj[(i+1)*(block_size+1) - 1]];
   }
   if (isLast)
    fillRight[(block_size-1)*VLEN + (VLEN-1)] = 0;
   else
    fillRight[(block_size-1)*VLEN + (VLEN-1)] = val[xadj[VLEN*(block_size+1) - 1]];
}

// Undo changes of fillVec in vector g
void
unFillVec(real *g, real *gVec, real *gSchurVec, int block_size)
{
   for(int i = 0; i < block_size; i++)
  {
      for(int j = 0; j < VLEN; j++)
    {
        g[j*(block_size+1) + i] = gVec[i*VLEN + j];
      }
   }
   for(int i = 1; i < VLEN; i++)
  {
      g[i*(block_size+1) - 1] = gSchurVec[i];
   }
}

void
unFillPar_share(real *g, real *gSchurVec, int block_size)
{
   for(int i = 1; i < VLEN*NTHREADS; i++)
  {
```

```
      g[i*(block_size+1) - 1] = gSchurVec[i];
   }
}

void
unFillPar_private(real *g, real *gVec, int block_size)
{
   for(int i = 0; i < block_size; i++)
  {
      for(int j = 0; j < VLEN; j++)
    {
        g[j*(block_size+1) + i] = gVec[i*VLEN + j];
      }
   }
}
```

B.16. precision.c

```
#ifndef PRECISION_GUARD
#define PRECISION_GUARD
  #ifndef PRECISION
    #define PRECISION 1
    #define real double
    #define mysqrt sqrt
    #define myfabs fabs
  #else
    #if PRECISION == 0
      #define real float
      #define mysqrt sqrtf
      #define myfabs fabsf
    #else
      #define real double
      #define mysqrt sqrt
      #define myfabs fabs
    #endif
  #endif
#endif // PRECISION_GUARD
```

Anexo C

Artículo XXIX Jornadas de Paralelismo (JP2018)

Se ha presentado un resumen de este Trabajo Fin de Grado como contribución a las XXIX Jornadas de Paralelismo (JP2018). Las JP2018 se celebrarán junto a las III Jornadas de Computación Empotrada y Reconfigurable como parte de las Jornadas SARTECO, organizadas por la Sociedad de Arquitectura y Tecnología de Computadores (SARTECO). Este evento de ámbito nacional es el principal punto de encuentro de la comunidad investigadora asociada a SARTECO.

Método Paralelo para Resolver Ecuaciones de Ligadura en Moléculas Lineales

Rubén Langarita-Benítez[1],
Carl Christian Kjelgaard Mikkelsen[2], Jesús Alastruey-Benedé[3],
Pablo Ibáñez-Marín[3] y Pablo García-Risueño[4]

Resumen— **La dinámica molecular es una técnica de simulación por computador que estudia la evolución en el tiempo de un sistema de partículas. Las herramientas que realizan estas simulaciones son esenciales en campos como la biomedicina o la industria química. Por ejemplo, la dinámica molecular se ha aplicado con éxito al diseño de nuevos fármacos o al análisis de materiales. En este campo, imponer ligaduras, es decir, fijar las longitudes de los enlaces atómicos, es una práctica habitual que permite aumentar el paso temporal de simulación (*time step*). De esta forma se pueden acelerar los experimentos o simular mayores intervalos temporales. Los algoritmos más usados para imponer ligaduras convergen linealmente y están basados en aproximaciones que afectan a su estabilidad numérica. Además, no se conoce una implementación paralela eficiente de los mismos.**

En este trabajo se presenta la implementación paralela de un nuevo algoritmo para imponer ligaduras, ILVES, que converge de forma cuadrática. Para acotar la complejidad del proyecto, se ha abordado la resolución de las ecuaciones de ligadura de una molécula compuesta por una cadena lineal de átomos. Utilizamos el método del complemento de Schur para dividir el sistema de ecuaciones en varios subsistemas susceptibles de ser resueltos de forma vectorial y paralela. Se han implementado tres versiones: base, vectorial y paralela, que resuelven las ecuaciones de ligadura hasta el límite de precisión máquina de forma eficiente. Para una tolerancia de 10^{-12}, la aceleración de las versiones vectorial y paralela con respecto a SHAKE es de 4.2 y 6.1 respectivamente.

Palabras clave— **Dinámica Molecular; GROMACS; Imposición de ligaduras; SHAKE; LINCS; ILVES;**

I. Introducción

La dinámica molecular es una técnica que permite simular con un computador el comportamiento a lo largo del tiempo de un sistema de átomos [1] [2] [3]. Esta herramienta se usa para realizar experimentos sin necesidad de disponer de las sustancias reales. Las simulaciones permiten explorar eficientemente diversos escenarios para que después se encaren los experimentos reales en laboratorio con mayor probabilidad de éxito.

Restringir los grados de libertad de los enlaces atómicos permite aumentar el paso temporal de simulación, de forma que pueden acelerarse los experimentos o simular mayores intervalos temporales. A este proceso se le llama imposición de ligaduras.

[1]Universidad de Zaragoza, e-mail: 621272@unizar.es
[2]Department of Computing Science and HPC2N, Umeå University, e-mail: spock@cs.umu.se
[3]Instituto Universitario de Investigación en Ingeniería de Aragón (I3A), Universidad de Zaragoza, e-mail: {jalastru,imarin}@unizar.es
[4]Institut für Physikalische Chemie, Universität Hamburg, e-mail: garcia.risueno@gmail.com

Los métodos más usados para resolver los sistemas de ecuaciones resultantes tras la imposición de ligaduras son SHAKE [4], RATTLE [5] y LINCS [6]. Esto métodos convergen linealmente y están basados en aproximaciones que afectan a su estabilidad numérica. A pesar de múltiples esfuerzos, no se conoce una implementación paralela eficiente de los mismos.

El objetivo de este trabajo consiste en implementar una versión paralela de un nuevo algoritmo, llamado ILVES, para resolver las ecuaciones de ligadura. Como primera aproximación, se va a abordar el caso de un sistema molecular compuesto por una cadena lineal de átomos, que se podría utilizar para simular moléculas compuestas por un esqueleto de átomos pesados cuyas ramificaciones son átomos de hidrógeno. Hay algunas investigaciones que han usado moléculas lineales y circulares para aplicar algoritmos de imposición de ligaduras [7]. Además, esta aproximación servirá como base para implementar el algoritmo sobre una molécula real.

Hemos utilizado el método del complemento de Schur para dividir el sistema de ecuaciones en varios subsistemas que pueden resolverse de forma vectorial y paralela. Se han implementado tres versiones: escalar, vectorial y paralela, que resuelven las ecuaciones de ligadura hasta el límite de precisión máquina de forma eficiente. La aceleración de las versiones vectorial y paralela con respecto a SHAKE es de 4.2 y 6.1 respectivamente, para una tolerancia de 10^{-12}.

El resto de este artículo está organizado como sigue. La sección II realiza una descripción general de la dinámica molecular y de los algoritmos para la imposición de ligaduras. En la sección III proponemos utilizar el método de Schur para realizar una implementación de ILVES que permite su ejecución vectorial y paralela. La sección VI describe la metodología utilizada y detalla los resultados obtenidos. Finalmente, la sección VII concluye.

II. Algoritmos para la Imposición de Ligaduras

En primer lugar se presentan los conceptos básicos de los algoritmos de imposición de ligaduras. A continuación, se describen los algoritmos SHAKE y LINCS, dos métodos de imposición de ligaduras implementados en la popular herramienta de dinámica molecular GROMACS. Por último, se presenta un nuevo algoritmo de imposición de ligaduras, llamado ILVES, que es el objeto de este trabajo.

A. Conceptos Básicos

Para llevar a cabo las simulaciones de dinámica molecular se calculan las posiciones de los átomos en cada paso temporal de la simulación. Para obtener las coordenadas de los átomos al final de un paso temporal, primero se calculan las fuerzas que se ejercen sobre los átomos, y después se mueven los átomos de acuerdo a esas fuerzas. La ecuación que relaciona la fuerza ejercida sobre un átomo α con la aceleración que produce sobre él es la siguiente:

$$\vec{F}_\alpha = m_\alpha \cdot \vec{a}_\alpha, \tag{1}$$

siendo m_α la masa de α y $\vec{a}_\alpha$ la aceleración de α.

Entre las fuerzas simuladas se encuentran, por ejemplo, las producidas entre los átomos debido a sus cargas eléctricas. Para este trabajo son de interés las fuerzas ejercidas entre los átomos que forman un enlace. Por ejemplo, en una molécula de agua (H_2O), las fuerzas que se ejercen entre el átomo de oxígeno y los dos átomos de hidrógeno, tal y como se ve en la Figura 1.

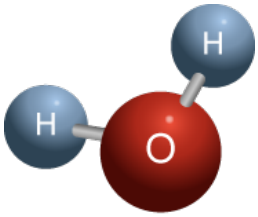

Fig. 1. Enlaces atómicos en una molécula de agua (H_2O).

La distancia de un enlace no es fija, los átomos vibran ligeramente. El período de estas vibraciones es del orden de femtosegundos (fs = 10^{-15} s) y determina el paso temporal de la simulación. Si el paso temporal es demasiado grande respecto al periodo de vibración, la simulación podría ser incorrecta. El paso temporal suele establecerse en un quinto del mínimo periodo de vibración.

La duración del paso temporal puede aumentarse si se imponen ligaduras, es decir, si se restringen los grados de libertad internos asociados a las vibraciones de alta frecuencia de los enlaces atómicos. Imponer una ligadura a un enlace entre dos átomos consiste en fijar una distancia constante entre dichos átomos igual a la longitud del enlace químico. La ligadura de un enlace k entre dos átomos a y b se define como:

$$\sigma_{k(a,b)} = |\mathbf{r}_a - \mathbf{r}_b|^2 - (d_{a,b})^2 = 0, \tag{2}$$

donde $\mathbf{r}_\mathrm{a}$ y $\mathbf{r}_\mathrm{b}$ son los vectores posición de los átomos a y b. Esta ligadura expresa que la distancia entre estos átomos debe ser igual a $d_{a,b}$.

La imposición de ligaduras añade nuevos términos al sistema de ecuaciones clásico de Newton para un sistema molecular:

$$\vec{F}_\alpha + \vec{F}_{ligaduras} = \vec{F}_\alpha - \sum \lambda \frac{\partial \sigma_k}{\partial x_\alpha} = m_\alpha \cdot \vec{a}_\alpha, \tag{3}$$

donde $\vec{F}_\alpha$ es la fuerza ejercida sobre el átomo α debida a las interacciones entre átomos, a la que llamaremos fuerza externa, y $-\sum_{k=1}^{N} \lambda_k \frac{\partial \sigma_k}{\partial r_\alpha}$ es la fuerza sobre el átomo α que aparece debida a las ligaduras, y los distintos λ_k son los multiplicadores de Lagrange. Cada uno de los multiplicadores de Lagrange esta asociado a una ligadura y determina la fuerza que se le aplica a los átomos de esa ligadura. Para que las distancias entre átomos sean las correctas ($d_{a,b}$), se realiza un procedimiento llamado imposición de ligaduras, que consiste en la aplicación las fuerzas artificiales que se originan con los multiplicadores de Lagrange (fuerzas de ligadura) [8].

Las herramientas de dinámica molecular suelen resolver este sistema en dos fases. Poniendo como ejemplo una molécula de agua, la situación inicial sería la de la Figura 2.

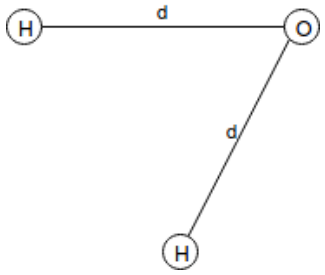

Fig. 2. Posición de los átomos de una molécula de agua al comienzo de un paso temporal de simulación. d es la longitud del enlace entre un átomo de hidrógeno y uno de oxígeno.

En una primera fase, los átomos se mueven de acuerdo a las fuerzas externas sin tener en cuenta las fuerzas de ligadura. Al acabar esta fase, la distancia entre los átomos de un enlace puede ser distinta a la longitud de dicho enlace. Por ejemplo, para una molécula de agua, el resultado podría ser el de la Figura 3.

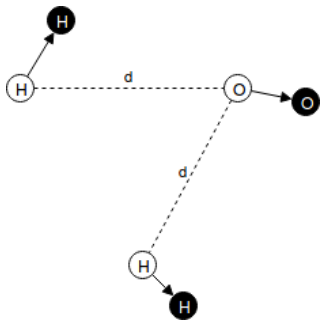

Fig. 3. Posición de los átomos de una molécula de H_2O después de aplicar las fuerzas externas.

En una segunda fase se corrigen las posiciones de los átomos para que se cumplan las ligaduras, es decir, para que las distancias entre los átomos unidos por un enlace sea igual a la longitud de dicho enlace. Después de realizar este proceso, las posiciones de los átomos de la molécula de agua podrían ser las mostradas en la Figura 4.

Las vibraciones de alta frecuencia descritas anteriormente ya no se simularían, de hecho, no son ne-

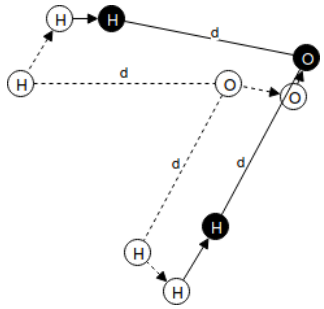

Fig. 4. Posición de los átomos de una molécula de agua después de modificar sus coordenadas para que se cumplan las ligaduras.

cesarias para el objetivo de la simulación, ya que no aportan información relevante.

El proceso de imposición de ligaduras es un problema complejo, ya que la corrección de las posiciones de dos átomos para hacer cumplir una ligadura afecta al resto de ligaduras asociadas a esos átomos.

En la ecuación 4 se pueden observar las condiciones necesarias para que tres átomos (a, b y c) cumplan las restricciones impuestas por dos ligaduras: a-b y b-c.

$$\begin{aligned} \delta^{(a,b)}(\mathbf{r}) &:= |\mathbf{r}_a - \mathbf{r}_b|^2 - (d_{a,b})^2 = 0 \\ \delta^{(b,c)}(\mathbf{r}) &:= |\mathbf{r}_b - \mathbf{r}_c|^2 - (d_{b,c})^2 = 0\,, \end{aligned} \tag{4}$$

siendo $\delta^{(i,j)}(\mathbf{r})$ la ligadura que actúa sobre los átomos i y j, r_i las coordenadas x, y, z del átomo i, y $d_{i,j}$ la distancia impuesta por la ligadura entre los átomos i y j (longitud de enlace).

B. SHAKE

SHAKE [4] fue el primer algoritmo implementado para la imposición de ligaduras. Este algoritmo establece un sistema de n ecuaciones con n incógnitas, siendo n el número de ligaduras. Al resolver el sistema, se obtienen los multiplicadores de Lagrange asociados a cada una de las ligaduras. Los átomos se mueven de acuerdo a estos multiplicadores de Lagrange. Este proceso realiza tantas iteraciones como sean necesarias hasta conseguir minimizar el error hasta el punto establecido al principio de la simulación. Estas iteraciones no se deben confundir con los pasos temporales. En cada paso temporal se simula un periodo de tiempo, mientras que entre iteraciones el valor del tiempo no varía, solo se intenta reducir el error dentro de un mismo paso temporal.

SHAKE resuelve el sistema de forma iterativa: primero se calcula el multiplicador de Lagrange para la primera ligadura y se corrigen las posiciones de los átomos de esa ligadura, después hace lo mismo con la segunda ligadura, la tercera y así sucesivamente.

Cuando se aplican estas correcciones, los átomos se mueven en la dirección del enlace en el paso temporal anterior. Esto se puede apreciar en la Figura 5, la dirección de la corrección $\vec{c}$ es la misma que la dirección del enlace en el paso temporal anterior $\vec{d}$.

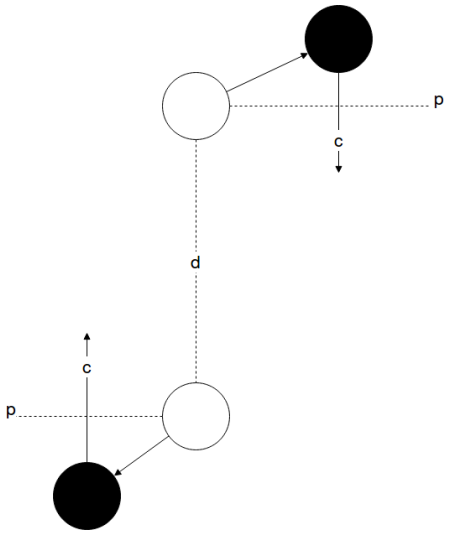

Fig. 5. Dirección de la corrección en SHAKE y en LINCS. Los círculos blancos indican la posición de los átomos en el paso temporal anterior y los negros en el actual pero sin haberse aplicado la corrección. La línea d indica el enlace en el paso temporal anterior. Las flechas c indican la dirección en la que tendrá lugar la corrección, que es paralela a d. En LINCS p es la línea sobre la que se parará el átomo después del primer paso (esto se verá en la sección de LINCS).

C. LINCS

Posteriormente se propuso LINCS [6] (Linear Constraint Solver). LINCS consta de dos fases. Al igual que en SHAKE, cuando se corrigen las posiciones de los átomos se mueven en la dirección del enlace en el anterior paso temporal. En la primera fase, cada átomo se mueve hasta la perpendicular con el enlace en el anterior paso temporal en el punto donde estaba el átomo, en la Figura 5 la línea p. En la segunda fase se realiza otra corrección siguiendo la misma dirección que en la primera, hasta que la distancia entre los átomos sea igual a la longitud del enlace.

En ambas fases se multiplican una serie de matrices. Además, en cierto punto del algoritmo es necesario invertir una matriz. Para evitar esta costosa operación, se usa la siguiente aproximación:

$$(I - B)^{-1} \simeq I + B^1 + B^2 + B^3 + B^4 + B^5 + ... \tag{5}$$

Normalmente se calcula hasta la cuarta u octava potencia. Este es la parte del algoritmo que más tiempo consume.

D. ILVES

Este trabajo se enmarca dentro de un proyecto cuyo objetivo final es integrar un nuevo algoritmo de imposición de ligaduras en GROMACS, una de las aplicaciones más usadas de dinámica molecular.

El algoritmo se llama ILVES [9]. Existen dos versiones de ILVES: ILVES-S basada en SHAKE e ILVES-L basada en LINCS. ILVES-S, la versión que se ha implementado en este trabajo, resuelve el mismo sistema de ecuaciones que SHAKE, pero de for-

ma directa en lugar de iterativa. SHAKE corrige las posiciones de los átomos cada vez que calcula un multiplicador de Lagrange, en cambio, ILVES-S resuelve el sistema y aplica las correcciones de todos los multiplicadores a la vez. Tanto ILVES-S como SHAKE tienen que resolver el sistema varias veces para conseguir minimizar el error hasta el punto deseado.

Una implementación escalar y secuencial de ILVES fue desarrollada por María Astón Serrano Gracia [10]. En dicho trabajo se evaluaba la precisión y convergencia de ILVES, así como una primera estimación de sus prestaciones. El siguiente gran objetivo era resolver el sistema de ecuaciones resultante de forma paralela. Para conseguirlo, y dada su dificultad, se definieron varios hitos incrementales. En primer lugar se paralelizó la resolución de ILVES para muchas moléculas pequeñas e idénticas, que es el caso del disolvente en el que se lleva a cabo la simulación [11]. En este trabajo se presenta la ejecución del siguiente hito: vectorizar y paralelizar ILVES para una cadena lineal de átomos sin ramificaciones. En las siguientes tareas se pretende paralelizar el algoritmo para resolver el sistema de ecuaciones resultante al imponer ligaduras a una cadena de átomos con ramificaciones y a una molécula arbitraria. El último objetivo es integrar en GROMACS el algoritmo desarrollado.

III. Implementación Eficiente de ILVES-S

Abordamos el caso de una molécula compuesta por una cadena lineal de átomos (ver Figura 6). Este caso especial servirá como base para la implementación de versiones posteriores.

El lenguaje de programación ha sido *C*, debido a que se ha partido del código escrito en la fase anterior del proyecto[11]. Se ha usado el compilador *gcc* versión 7.1, junto con *Make* para facilitar la compilación.

ILVES-S establece un sistema lineal de n ecuaciones con n incógnitas, siendo n el número de ligaduras. Al tratarse de una molécula lineal, si se tienen m átomos, n será igual a $m-1$.

En el sistema de ecuaciones a resolver aparece un elemento no nulo en la posición i, j si los enlaces i y j poseen algún átomo en común. Por lo tanto, el sistema lineal equivalente ($Ax = b$) será tridiagonal, es decir, los únicos elementos no nulos de la matriz A están en la diagonal principal y sus adyacentes.

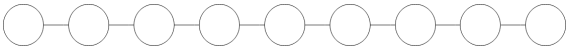

Fig. 6. Representación de una cadena lineal de átomos.

Para resolver un sistema de ecuaciones se puede, por ejemplo, utilizar el método de eliminación gaussiana o se puede invertir la matriz y multiplicar por el vector de términos independientes, estos métodos son muy costosos computacionalmente. Se ha optado por el método del complemento de Schur, ya que este método es propicio para ser paralelizado.

IV. Método del complemento de Schur

El método del complemento de Schur divide la matriz en submatrices más pequeñas, de manera que entre submatrices quede una fila y una columna que no pertenezcan a ninguna submatriz, tal y como se puede ver en la Figura 7.

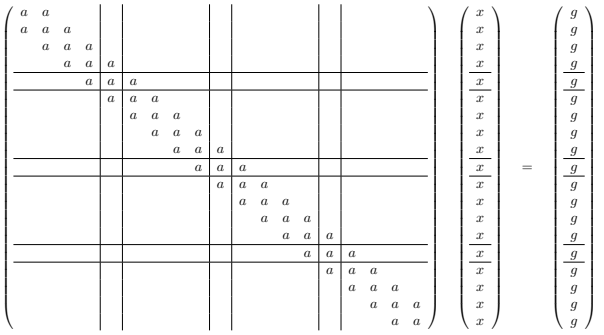

Fig. 7. Aspecto del sistema de ecuaciones inicial. Los ceros han sido representados con espacios en blanco y los elementos no nulos con a. La matriz se ha dividido en submatrices más pequeñas de dimensión 4×4. Entre las submatrices queda una fila y una columna que no pertenece a ninguna submatriz. En la figura se repite la misma letra para designar distintos elementos aunque tengan diferente valor, se hace por simplificar y se aplica también a las demás figuras.

Primero se resuelven los sistemas correspondientes a las submatrices. Cada una de las submatrices se resuelve mediante factorización LU. Sin embargo, por sencillez se va a usar eliminación gaussiana en la descripción del funcionamiento del método de Schur. A continuación, se explica el procedimiento para una submatriz.

El objetivo es que la submatriz termine siendo la matriz identidad. Se empieza eliminando la diagonal inferior. En la Figura 8 se parte del estado inicial de una submatriz y se muestra cómo eliminar el elemento de la diagonal inferior de la fila 2.

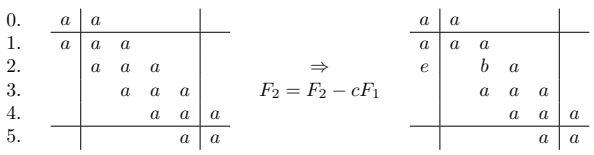

Fig. 8. Eliminación de los elementos de la diagonal inferior.

A la fila 2 se le resta la fila 1 multiplicada por un factor c para eliminar el elemento de la diagonal inferior. Aparece un elemento e en la columna izquierda debido a la a de la misma columna de la fila superior. Este proceso se repite con las filas restantes, hasta eliminar los elementos de la diagonal inferior.

A continuación hay que eliminar los elementos de la diagonal superior. En la Figura 9 se parte del estado en el que todos los elementos de la diagonal inferior han sido eliminados. Obsérvese que han ido apareciendo elementos a lo largo de la columna izquierda durante la eliminación de la diagonal inferior debido a la influencia de las filas superiores. Por el mismo motivo, al eliminar el elemento de la diagonal inferior de la fila 5 también ha aparecido un valor en la esquina inferior izquierda y se ha modificado

el valor de la esquina inferior derecha de dicha fila. Las S se denominan términos del complemento de Schur, del que se hablará más adelante. En la parte derecha de la Figura 9 se muestra como se elimina el elemento de la diagonal superior de la fila 3.

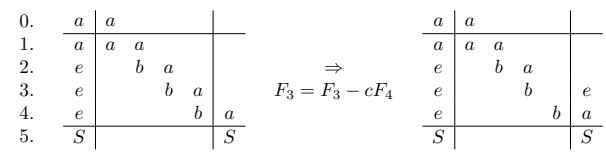

Fig. 9. Eliminación de los elementos de la diagonal superior.

A la fila 3 se le ha restado la fila 4 multiplicada por un factor c de manera que el elemento de la diagonal superior queda eliminado. Aparece un elemento e en la columna derecha al igual que ha sucedido al eliminar los elementos de la diagonal inferior en la columna izquierda. El elemento e de la columna izquierda de la fila 3 también queda modificado. Esto se repite con las filas restantes hasta eliminar todos los elementos de la diagonal superior.

En la Figura 10 se parte del estado en el que se han eliminado las diagonales inferior y superior. Ha aparecido un elemento en la esquina superior derecha y se ha modificado el valor de la esquina superior izquierda, estos valor también pertenecen al complemento de Schur. Obsérvese que el elementos de la esquina superior izquierda y el de la esquina inferior derecha también sufrirán modificaciones debido a las submatrices contiguas. En la Figura 10 se muestra cómo se modifica la fila 1 para que el elemento de la diagonal sea igual a 1.

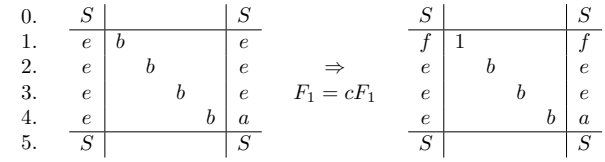

Fig. 10. Escalado de los elementos de la diagonal principal.

La fila 1 se multiplica por un factor c de manera que el elemento de la diagonal pase a ser 1. Los elementos e de las columnas laterales quedan modificados y pasan a ser f. Esta acción se repite para todas las filas de la submatriz. El resultado se muestra en la Figura 11.

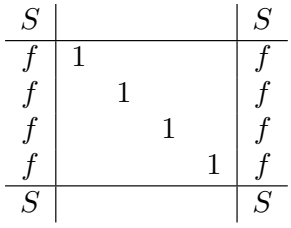

Fig. 11. Estado de la submatriz y de las filas y columnas contiguas después de resolver el sistema de ecuaciones asociado a la submatriz.

Los elementos de la diagonal pasan a ser 1, y la submatriz es la matriz identidad. En las columnas laterales han quedado elementos f, a los que se les denomina *fill-ins*.

Una vez aplicado este procedimiento a todas las submatrices, el resultado es el de la Figura 12.

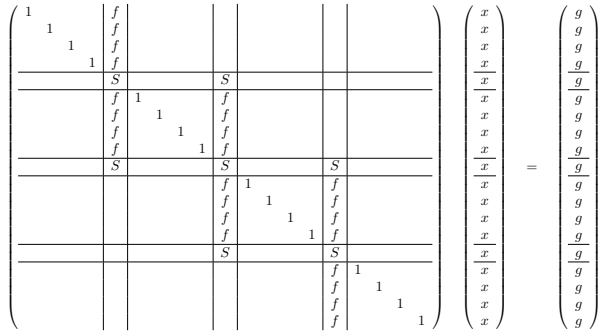

Fig. 12. Representación del sistema de ecuaciones después de transformar las submatrices. Aparecen los *fill-ins* y los valores pertenecientes al complemento de Schur (f y S respectivamente).

A continuación se forma un sistema tridiagonal con las ecuaciones de las filas entre las submatrices (en la Figura 12, aquellas filas que contienen S), que se llama complemento de Schur, tal y como se ve en la Figura 13. Este sistema se resuelve también mediante factorización LU. Si el complemento de Schur tuviera muchas ecuaciones podría resolverse a su vez mediante el método del complemento de Schur, ya que tiene las mismas características que el sistema inicial. En nuestro caso no merece la pena hacer esto debido al pequeño tamaño del complemento de Schur. Tras la resolución del complemento de Schur queda el sistema de ecuaciones mostrado en la Figura 14.

$$\begin{pmatrix} S & S & \\ S & S & S \\ & S & S \end{pmatrix} \begin{pmatrix} x \\ x \\ x \end{pmatrix} = \begin{pmatrix} g \\ g \\ g \end{pmatrix}$$

Fig. 13. Complemento de Schur.

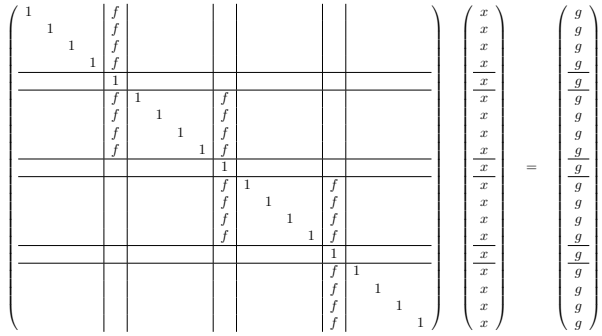

Fig. 14. Representación del sistema de ecuaciones después de resolver el complemento de Schur.

Por último, se eliminan los *fill-ins* de cada submatriz mediante eliminación gaussiana usando las filas del complemento de Schur que acaba de ser resuelto. Cada ecuación resuelta del complemento de Schur servirá para eliminar los *fill-ins* que tenga encima y debajo.

Se parte de la matriz que aparece en la Figura 14. En la figura 15 se observa cómo eliminar un *fill-in*.

El *fill-in* de la cuarta fila de la columna derecha ha sido eliminado. Este procedimiento se repetirá para

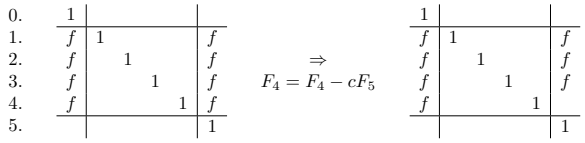

Fig. 15. Eliminación de *fill-in*.

los *fill-ins* restantes de ambas columnas. Con todos los *fill-ins* eliminados, la matriz se convierte en la matriz identidad, y por lo tanto los términos independientes son la solución del sistema.

La ventaja de este algoritmo es que el trabajo puede ser paralelizado, de forma que cada hilo se encargue de una submatriz distinta, a excepción de la resolución del complemento de Schur.

V. Implementación

La implementación del algoritmo no se ha realizado de forma directa, sino que se ha llevado a cabo una aproximación sistemática e incremental. Primero se ha desarrollado una versión secuencial, utilizando eliminación gaussiana para resolver los sistemas de las submatrices y una representación densa de la matriz. Esta versión es muy lenta y costosa en memoria, pero es la más fácil de implementar, de forma que se usa como referencia para la verificación de resultados.

A continuación, se ha implementado una versión secuencial utilizando la factorización LU para resolver los sistemas de las submatrices y una representación dispersa de la matriz. Una vez verificada su corrección, esta versión ha servido como referencia para compararla con las demás, tanto en tiempo como en precisión. Por último, se han desarrollado las versiones vectorial y paralela, que son el objetivo final del trabajo.

Previo a la resolución del sistema ha sido necesario generar la matriz y el vector de términos independientes. Después de resolver el sistema, la solución se utilizará para corregir las posiciones de los átomos con otra función. Este proceso se repetirá hasta conseguir minimizar el error hasta el punto deseado. A estas repeticiones las llamaremos iteraciones. No se deben confundir con los pasos temporales. En cada paso temporal se simula un periodo de tiempo, mientras que entre iteraciones el valor del tiempo no varía, solo se intenta reducir el error dentro de un mismo paso temporal.

A. Versión secuencial

En esta sección se analiza la versión secuencial con representación dispersa y utilizando la factorización LU para resolver las submatrices. Se han creado tres funciones, una por cada paso del método de Schur:

1. Resolver los sistemas de ecuaciones de las submatrices
2. Resolver el complemento de Schur
3. Eliminar los *fill-ins*

En el primer paso, durante la resolución de las submatrices, los resultados necesarios para los pasos posteriores son el vector de términos independientes, los

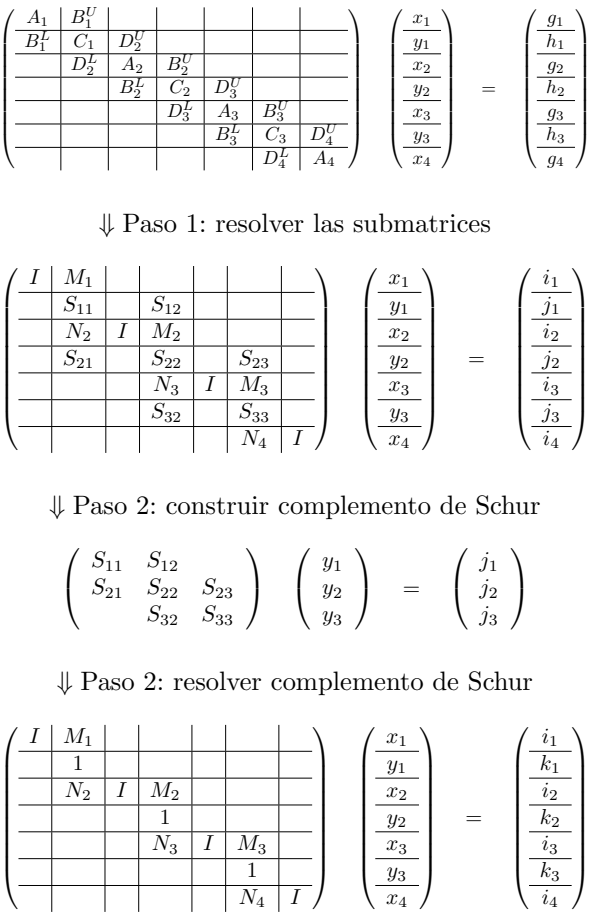

Fig. 16. Representación del sistema de ecuaciones después de cada paso. La figura superior es el estado inicial, es el mismo sistema de ecuaciones que el de la figura 7, los bloques A son de tamaño 4×4, los bloques B^U y D^L son de tamaño 4×1, los bloques B^L y D^U son de tamaño 1×4 y los bloques de C de tamaño 1×1. La segunda figura representa el sistema de ecuaciones de la figura 12, es la representación del sistema después de haber resuelto las submatrices, los bloques I representan la matriz identidad 4×4, los bloques M y N son los fillins y los bloques S son los complementos de Schur. La tercera figura representa el sistema de ecuaciones formado por los complementos de Schur extraídos de la segunda figura. La última figura representa el sistema de ecuaciones de la figura 14, es el sistema después de haber resuelto el sistema de complementos de Schur.

fill-ins y los complementos de Schur. Las submatrices van a terminar siendo matrices identidad, por lo que no hace falta calcular su valor. Las ecuaciones 6-10 definen la manera de calcularlos. Estas ecuaciones hacen referencia a los bloques definidos en los sistemas de ecuaciones de la figura 16, en concreto a los dos primeros.

$$M_i = A_i^{-1} B_i^U \tag{6}$$

$$N_i = A_i^{-1} D_i^L \tag{7}$$

$$i_i = A_i^{-1} g_i \tag{8}$$

$$Sij = \begin{cases} i = j & C_i - M_i B_i^L - N_{i+1} D_{i+1}^U \\ i > j & -N_i B_i^L \\ i < j & -M_i D_j^U \end{cases} \tag{9}$$

$$j_1 = h_1 - i_i B_i^L - i_{i+1} D_{i+1}^U \quad (10)$$

Para calcular todos los componentes es necesario calcular las inversas de las submatrices. Como ya se ha dicho, esto es muy costoso computacionalmente, por lo que se optará por realizar una factorización LU.

Una vez calculados todos los componentes, en el segundo paso del algoritmo es necesario tratar el complemento de Schur como si fuese un sistema independiente. Hay que extraer de la matriz original el complemento de Schur a una nueva matriz, el resultado sería el tercer sistema de ecuaciones de la figura 16. Se puede observar que el nuevo sistema esta formado por una matriz tridiagonal. El sistema se resolverá mediante factorización LU, ya que no merece la pena paralelizar debido al pequeño tamaño de la matriz. En este caso solo es necesario factorizar el complemento de Schur y aplicar $forward$ y *backward* sobre los términos independientes correspondientes. Una vez resuelto el sistema de complementos de Schur, las filas volverán a insertarse en el sistema original, el resultado será el cuarto sistema de ecuaciones de la figura 16.

Por último es necesario eliminar los *fill-ins*. Al hacer desparecer los *fill-ins* los términos independientes se modificarán. A cada término independiente hay que aplicarle la influencia de las filas con las que se han eliminado los *fill-ins*. Los términos independientes son los únicos datos relevantes, ya que después de eliminar los *fill-ins*, la matriz A se convertirá en la matriz identidad y por lo tanto los términos independientes serán la solución del sistema.

B. Versión vectorial

Para que un código sea vectorizable, se deben cumplir dos condiciones. La primera, es que las operaciones a realizar se repitan de forma regular sin dependencias entre ellas. En el caso del método del complemento de Schur, el algoritmo que se aplica a todas las submatrices es el mismo. Por ejemplo, a la hora de factorizar las submatrices, con una longitud vectorial de 4 se podrían factorizar 4 submatrices simultáneamente. La segunda condición, es que los datos sobre los que se realizan las operaciones estén contiguos en memoria. Los procesadores actuales soportan operaciones vectoriales para cargar y guardar datos no contiguos en memoria, pero son más lentas que si se reorganizan los datos. En el caso de factorizar las submatrices, se necesita que los elementos i-ésimos de las submatrices que se van a factorizar estén contiguos en memoria. Para este propósito se han creado unas estructuras que contienen la misma información que en la versión secuencial, pero con los datos reordenados.

Para la versión secuencial se necesitaban dos estructuras: una para la matriz y otra para los términos independientes. En la versión vectorial han sido necesarias tres estructuras para la matriz y dos para los términos independientes. Entre paréntesis se indica el nombre de las estructuras en el programa. La matriz se ha divido en submatrices ($valVec$), *fill-ins* ($fillLeft$ y $fillRight$) y el complemento de Schur ($schurMatrix$). Los términos independientes se han dividido en los que corresponden a las submatrices ($gVec$) y en los que corresponden al complemento de Schur ($gSchurVec$).

$valVec$, $fillLeft$, $fillRight$ y $gVec$ han sido reorganizados de manera que todos los elementos i-ésimos estén contiguos en memoria. No hace falta reorganizar las estructuras correspondientes al complemento de Schur ($schurMatrix$ y $gSchurVec$), ya que las operaciones que se realizan con ellas (factorización, forward y backward en el paso 2) no van a ser vectorizadas.

Como ya se ha dicho, antes de la resolución del sistema se deben generar la matriz y los términos independientes, y posteriormente, los resultados se usan para corregir las posiciones de los átomos. En una primera aproximación para vectorizar el algoritmo, se han reorganizado los datos justo antes y después de la resolución del sistema, pasando de las estructuras secuenciales a vectoriales y viceversa en funciones específicas. Sin embargo, en versiones posteriores se han modificado la función que construye la matriz, la que construye el vector de términos independientes y la que corrige las posición de los átomos para que escriban o lean directamente de las estructuras vectoriales.

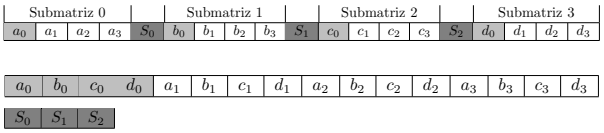

Fig. 17. Ejemplo de reorganización de los términos independientes con 19 elementos y con una longitud vectorial de 4. Arriba se observa la estructura utilizada para la versión secuencial. Abajo se observan las dos estructuras usadas en la versión vectorial: $gVec$ y $gSchurVec$ respectivamente.

Para vectorizar el código hace falta que el bucle más interno itere sobre las submatrices, de manera que el compilador detecte que esos elementos están contiguos en memoria y pueda vectorizar. En la figura 18 se muestran las versiones secuencial y vectorial de la función de factorización. En el código correspondiente a la versión vectorial se puede apreciar que el bucle más interno itera sobre las submatrices. El compilador es capaz de vectorizar este bucle automáticamente, detecta que en el bucle interno los operandos y los resultados están contiguos en memoria.

C. Versión paralela

Por último, se ha implementado la versión paralela. La idea es dividir la matriz en submatrices de manera que estas puedan ser procesadas en paralelo. Para la versión vectorial, la matriz se dividía en $VLEN$ submatrices. Para la versión paralela, la matriz se dividirá en el número de hilos que se vayan a utilizar multiplicado por $VLEN$. De esta manera cada hilo tendrá que procesar $VLEN$ submatrices.

```
for(int i = start; i < start + block_size - 1; i++) {
    double factor =
         val[xadj[i+1]-1]/val[xadj[i+1]-2];
    val[xadj[i+1]-1] = factor;
    val[xadj[i+1]+1] -= val[xadj[i+1]]*factor;
}
```

```
for(int i = 0; i < block_size - 1; i++) {
    #pragma GCC ivdep
    for (int j = 0; j < VLEN; j++) {
        double factor =
             valVec[xadjVec[i]+2*VLEN+j]/valVec[xadjVec[i]+VLEN+j];
        valVec[xadjVec[i]+2*VLEN+j] = factor;
        valVec[xadjVec[i+1]+VLEN+j] -=
             valVec[xadjVec[i+1]+j]*factor;
    }
}
```

Fig. 18. Código para factorizar una submatriz. Arriba la versión secuencial, abajo la vectorial.

Por ejemplo, para una cadena de 2048 átomos, el número de ligaduras (y por tanto la dimensión de la matriz) será de 2047. Con una longitud vectorial de 4 y usando 8 hilos, habría que dividir la matriz en 32 submatrices. El tamaño del complemento de Schur sería de 31 filas y el tamaño de las submatrices de 63 filas. Las 32 submatrices se repartirían entre los 8 hilos, a cada uno de los cuales le tocarían 4 submatrices. A su vez, cada uno de los hilos vectorizará los cálculos de las 4 submatrices que le toquen.

Para la versión paralela cada hilo tendrá sus propias estructuras $valVec$, $fillLeft$, $fillRight$ y $gVec$. Las estructuras $schurMatrix$ y $gSchurMatrix$ serán compartidas, debido a que el complemento de Schur es compartido por todos. Esto permite paralelizar la generación del sistema y la corrección de posiciones, cada hilo se encargará de generar su parte del sistema (submatriz y términos independientes) y corregir las posiciones que le toquen. El hilo principal es el encargado de resolver el complemento de Schur, ya que esta sección de código no esta paralelizada.

Durante la resolución de los sistemas formados por las submatrices, dos submatrices contiguas comparten el término de Schur que se queda entre ellas. Para que no se produzcan inconsistencias en los datos, la sección de código que modifica los términos de Schur de la diagonal se protege con un mutex para cada término.

Se ha usado OpenMP para paralelizar el algoritmo. La división de trabajo se ha hecho de forma estática. La zona paralela consta de un bucle, para el que cada una de sus iteraciones será ejecutada por un hilo. El resultado sería el de la figura 19.

```
#pragma omp parallel for schedule(static,1)
     default(shared)
for(int l=0; l < NTHREADS; l++)
```

Fig. 19. Directiva de OpenMP para paralelizar un bucle.

VI. Experimentos Numéricos

En esta sección se describe la metodología utilizada y se presentan los resultados de las pruebas realizadas.

A. Metodología

Los experimentos se han realizado en un sistema con procesador Intel Core i5-4570 (microarquitectura Haswell) a 3.20 GHz con 4 núcleos y 12 GB de memoria. Este procesador soporta la extensión vectorial AVX2, cuya longitud vectorial es de 4 elementos si se trabaja con números de coma flotante de doble precisión.

Para medir el error de una ligadura entre dos átomos a y b, $e^{(a,b)}$, se ha calculado la diferencia entre el cuadrado de la distancia de los dos átomos, $|\vec{r}_a - \vec{r}_b|^2$, y el cuadrado de la longitud del enlace atómico, $(d_{a,b})^2$, es decir, la que debería haber para que se cumpliese la ligadura.

$$e^{(a,b)} := |\vec{r}_a - \vec{r}_b|^2 - (d_{a,b})^2 \tag{11}$$

El error de un paso temporal es el máximo de los errores de todas las ligaduras. La tolerancia se define como el umbral máximo de error permitido en cada paso de la simulación. Se realizarán tantas iteraciones como sean necesarias hasta conseguir reducir el error por debajo de la tolerancia indicada en la configuración del experimento.

El número de iteraciones se refiere al número de veces que se ha resuelto el sistema de ecuaciones para corregir las posiciones de los átomos.

Las pruebas se han realizado con moléculas de 2048 átomos y simulando 100 pasos temporales. Cada átomo se mueve de forma aleatoria (esto simularía las fuerzas externas) y se aplica el algoritmo de imposición de ligaduras para corregir las posiciones, esto se repite para cada paso temporal.

B. Resultados

Para mostrar los resultados obtenidos con los nuevos algoritmos se han realizado dos experimentos. En el primero se ha medido el error en función del número de iteraciones. En el segundo se mide el tiempo de ejecución de cada algoritmo en función de la tolerancia.

La figura 20 muestra el error (eje Y en escala logarítmica) en función de el número de iteraciones (eje X) para los algoritmos SHAKE e ILVES-S. ILVES-S necesita unas pocas iteraciones para conseguir un error mínimo que viene impuesto por la precisión de la máquina, mientras que SHAKE requiere más de 100 iteraciones. Este resultado ya se puso de manifiesto en el trabajo de M.A. Serrano [10]. La convergencia de ILVES-S es cuadrática mientras que la de SHAKE es lineal. Pese a esto, en el trabajo citado se observó que el tiempo de ejecución de aquella primera versión de ILVES-S era similar al tiempo de SHAKE o incluso mayor en algunos casos. El resultado era debido a un tiempo de ejecución mucho mayor para cada iteración de ILVES-S respecto a las de SHAKE.

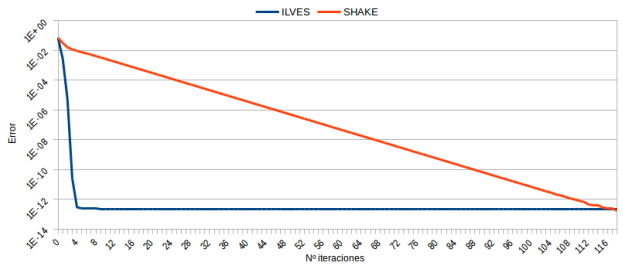

Fig. 20. Error dependiendo del número de iteraciones.

En el segundo experimento se ha medido el tiempo de ejecución que requieren las nuevas versiones de ILVES-S. La Figura 21 muestra los tiempos de ejecución de SHAKE y la versión base de ILVES-S (eje Y) para distintos valores de tolerancia (eje X).

Al reducir la tolerancia se requiere un número mayor de iteraciones para conseguirla y, por lo tanto, un mayor tiempo de ejecución. En el tiempo de ILVES-S se puede observar una forma en escalera. Esto es debido a que una reducción de la tolerancia no implica necesariamente un aumento de iteraciones. En los puntos de la gráfica en los que se produce un aumento del tiempo es debido a que el algoritmo necesita una iteración más para conseguir reducir el error. El tiempo de ejecución de SHAKE es siempre mayor que el de ILVES-S, y la diferencia aumenta cuanto más se desea reducir el valor de la tolerancia. Con una tolerancia de 10^{-12}, la versión base de ILVES-S es 3.46 veces más rápido que SHAKE. Sin embargo, ya se ha comentado que la ventaja de ILVES-S respecto a SHAKE es la posibilidad de ser vectorizado y paralelizado.

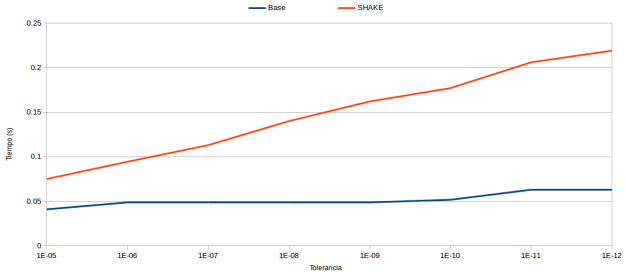

Fig. 21. Comparación de SHAKE e ILVES-S. Tiempo de ejecución respecto a la tolerancia.

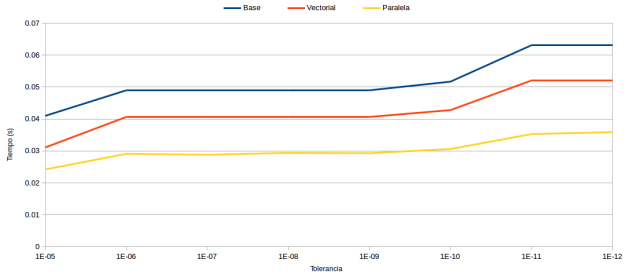

Fig. 22. Comparación de las tres versiones de ILVES-S. Tiempo de ejecución respecto a la tolerancia.

En la Figura 22 se muestran los tiempos de ejecución de las tres versiones de ILVES-S (la versión base, la vectorial y la paralela).

Las diferencias entre las tres versiones aumentan al disminuir la tolerancia debido a que las versiones vectorial y paralela requieren una inicialización más costosa de las estructuras. La importancia relativa del tiempo de esta inicialización disminuye al aumentar el número de iteraciones.

El *speedup* conseguido al vectorizar varía entre un 1.2 y un 1.31 mientras que la paralelización consigue *speedups* entre 1.28 e 1.44 respecto a la versión vectorizada.

La versión final es entre un 66 % y un 79 % más rápida que el algoritmo base y entre un 209 % y un 510 % más rápida que SHAKE.

Pese a los buenos resultados obtenidos somos conscientes de que el potencial que ofrecen la vectorización y la paralelización es mucho mayor que el conseguido. Por tanto, este trabajo servirá de base para desarrollar nuevas optimizaciones en dos líneas de trabajo: i) disminución de la sobrecarga de inicialización de las estructuras de datos adaptadas y ii) redimensionamiento de las submatrices y paralelización de la resolución de la matriz complemento de Schur.

VII. Conclusiones

En este trabajo se ha conseguido vectorizar y paralelizar el algoritmo de imposición de ligaduras ILVES, para una molécula lineal. Para ello, ha sido necesario resolver un sistema de ecuaciones tridiagonal mediante el método del complemento de Schur. Este método es vectorizable y paralelizable, y se puede aplicar fácilmente a un sistema de ecuaciones tridiagonal.

Se han implementado varias versiones de ILVES. Primero se ha implementado una versión a modo de prueba que usaba una representación densa de la matriz. A continuación, se ha implementado una versión base que se ha servido de referencia para compararla con las demás versiones. Después se ha implementado una versión vectorial. Para ello ha necesario definir nuevas estructuras reorganizando los datos para que el compilador sea capaz de vectorizar. Por último se ha implementado la versión paralela que hace uso de varios hilos del procesador para resolver el sistema, y a su vez cada hilo hace uso de las extensiones vectoriales.

Por último, se han realizado pruebas. Se ha implementado una versión secuencial de SHAKE para poder compararla con ILVES. Las iteraciones de ILVES son más costosas que las de SHAKE, pero a cambio se necesitan muchas menos iteraciones para alcanzar la solución. La versión paralela, la más rápida de todas, es hasta 6.1 veces más rápida que SHAKE.

La próxima fase del proyecto consistirá en generalizar esta solución para una cadena de átomos con ramificaciones. El proyecto finalmente acabará por integrarse en GROMACS, de manera que se necesitará menos tiempo para ejecutar los algoritmos de dinámica molecular en esta aplicación.

Agradecimientos

Este trabajo ha sido financiado por los proyectos TIN2016-76635-C2-1-R (AEI/FEDER, UE) y gaZ: grupo de investigación T58_17R (Gobierno de Aragón y Fondo Social Europeo).

Referencias

[1] S.A. Adcock and J.A McCammon, *Molecular Dynamics: Survey of Methods for Simulating the Activity of Protein*, 2006.

[2] D. Frenkel and B. Smit, *Understanding molecular simulations: From algorithms to applications*, 1996.

[3] Frank Jensen, *Introduction to Computational Chemistry*, 1999.

[4] J. P. Ryckaert, G. Ciccotti, and H. J. C. Berendsen, "Numerical integration of the cartesian equations of motion of a system with constraints: Molecular dynamics of n-alkanes," *J. Comp. Phys. 23, 327–341*, 1977.

[5] H. C. Andersen, "Rattle: A "velocity" version of the shake algorithm for molecular dynamics calculations," *J. Comp. Phys. 52, 24–34*, 1983.

[6] Berk Hess, Henk Bekker, Herman J. C. Berendsen, and Johannes G. E. M. Faaije, "Lincs: A linear constraint solver for molecular simulations," *Journal of Computational Chemistry*, 1997.

[7] A. G. Bailey, C. P. Lowe, and A. P. Sutton, "Efficient constraint dynamics using milc shake," *Journal of Computational Physics 227*, 2008.

[8] P. García-Risueño, P. Echenique, and J.L. Alonso, "Exact and efficient calculation of lagrange multipliers in biological polymers with constrained bond lengths and bond angles: Proteins and nucleic acids as example cases," *Journal of Computational Chemistry 32: 3039-3046*, 2011.

[9] P. García-Risueño, "Constraint implementation based on analytical calculations: a possible way to improve widely used solvers," Tech. Rep.

[10] María Astón Serrano Gracia, "Implementación e integración en gromacs de un algoritmo eficiente y preciso para imponer ligaduras en simulaciones de dinámica molecular," Proyecto fin de carrera, Universidad de Zaragoza, 2013.

[11] Carl Christian Kjelgaard Mikkelsen, Jesús Alastruey-Benedé, Pablo Ibáñez-Marín, and Pablo García Risueño, "Accelerating sparse arithmetic in the context of newton's method for small molecules with bond constraints," in *Parallel Processing and Applied Mathematics - 11th International Conference (PPAM 2015)*, September 6-9, 2015, pp. 160–171.