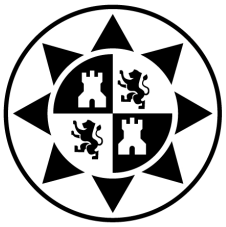
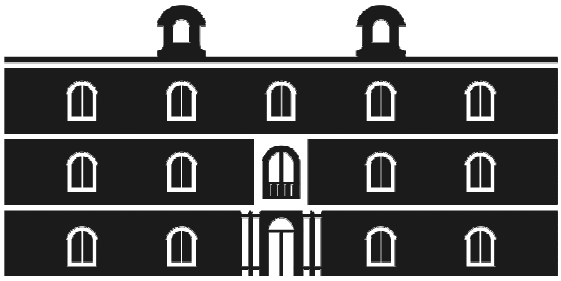


Universidad
Politécnica
de Cartagena



industriales
etsii UPCT

Implementación de un algoritmo de superposición 3D de fragmentos químicos asociados a moléculas en forma de anillo

Titulación: Ingeniero Industrial

Alumno/a: Roberto Castellano Sánchez
Director/a/s: Mathieu Kessler

Cartagena, 5 de marzo de 2010

Agradecimientos

A Mathieu, por su disposición y amabilidad desde el primer momento

A mi padre, por habernos enseñado de primera mano la cultura del esfuerzo.

A Diego y Mariano, por su apoyarme sin dudar en todo lo que he hecho.

A Anica, por ser estar conmigo cuando ya no quedaba nadie.

A Nacho, Ramón, Edu, Manolo, Guille, Elena, Sole, Pedro y todos los que sufrieron mis despistes.

A Pelas, por arrancarme tantas sonrisas.

A Solano, por regalarme tantas y tantas noches.

A Andrés, por todos los momentos que nunca olvidaré.

A Javi, por una amistad que no se puede medir.

A mi madre

Índice

1. Introducción

1.1. Problemática

El análisis de datos moleculares se utiliza en química y biología para conseguir información energética, psicoquímica, biológica o farmacocinética que depende de las características físicas de dichas moléculas.

El análisis conformacional de un conjunto de fragmentos en forma de anillos busca establecer similitudes entre las conformaciones geométricas de los anillos, con el objetivo de relacionarlas con propiedades relevantes del compuesto.

Una primera aproximación al análisis conformacional se puede hacer a través de la mecánica molecular que permite llevar a cabo cálculos energéticos que proporcionan información sobre las conformaciones de mínima energía y ayuda por lo tanto a predecir qué conformaciones serían las más frecuentes para un determinado tipo de fragmentos.

Sin embargo, estos cálculos no son sencillos y se ven dificultados por la presencia de átomos de diferentes naturaleza, o de enlaces más complejos. Es por lo tanto de interés aplicar métodos estadísticos sobre datos cristalográficos empíricos, que permitan, o bien confirmar las conformaciones esperadas predichas por los cálculos de mecánica molecular, o bien obtener información sobre las conformaciones más frecuentes para un tipo de fragmentos donde los cálculos son demasiado complicados.

Dos preguntas generales a las que se intenta contestar en el contexto del análisis conformacional, partiendo de datos cristalográficos empíricos son las siguientes:

- Para comparar dos fragmentos: ¿hasta qué punto los dos fragmentos considerados tienen una conformación geométrica similar?

- Dada un conjunto de varios fragmentos: ¿podemos identificar grupos dentro del conjunto que estén formados por fragmentos de conformación geométrica similar?

1.2. Métodos estadísticos para el análisis conformacional

1.2.1. Datos cristalográficos usados para los métodos estadísticos.

Mencionaremos dos tipos de datos cristalográficos que se han usado en la literatura para llevar a cabo un análisis conformacional con métodos estadísticos: por una parte los llamados ángulos de torsión y por otra parte las coordenadas de los átomos en un sistema cartesiano.

a) Ángulos de torsión

Este análisis se basa en los ángulos que forman planos definidos por tres átomos de una misma molécula.

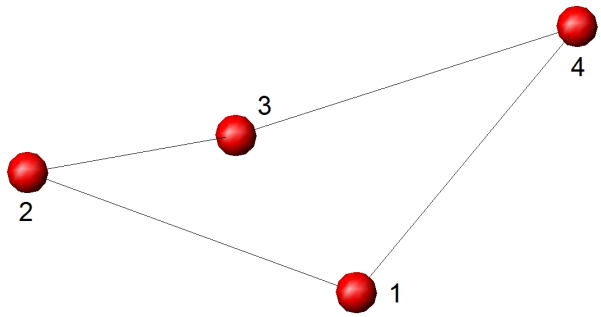


Figura 1: Anillo de 4 átomos

Consideremos un anillo de N átomos consecutivos $A_1, A_2, \ldots, A_N$. Los datos cristalográficos que se utilizan para caracterizar la estructura se resumen en la secuencia de ángulos de torsion $\tau_{1;\,2;\,3;\,4}$, ..., $\tau_{k-2;\,k-1;\,k;\,k+1}$, ... $\tau_{N;\,1;\,2;\,3}$, donde $\tau_{k-2;\,k-1;\,k;\,k+1}$ denota el ángulo de torsión entre los átomos A_{k-2}, A_{k-1}, A_k y A_{k+1}. Este ángulo se corresponde con el formado entre el plano que contiene A_{k-2}, A_{k-1} y A_k, y el plano que contiene A_{k-1}, A_k y A_{k+1}.

Por ejemplo, para el anillo anterior, el $\tau_{1;\,2;\,3;\,4}$ será el ángulo formado por un plano que contenga a A_1, A_2 y A_3 y un plano que contenga A_2, A_3 y A_4.

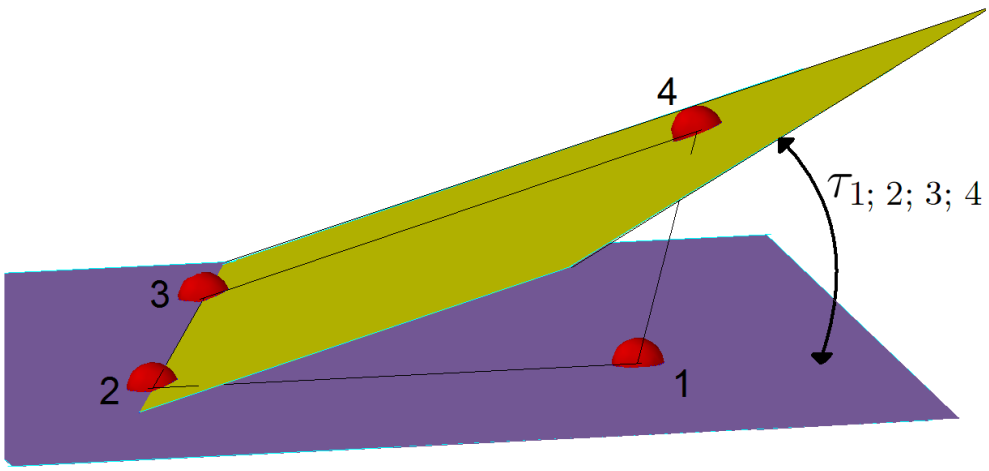


Figura 2: Ejemplo de ángulo de torsión

Ejemplos de métodos estadísticos que usan los ángulos de torsión para llevar a cabo el análisis conformacional de conjuntos de datos se pueden encontrar en [2], [4], [5], y [11].

b) Coordenadas de los átomos La colección más grande de datos cristalográficos sobre estructuras moleculares está en la Cambridge Structural Database (CSD), [3]. Para este trabajo particularmente, se han estudiado fragmentos con forma de anillo.

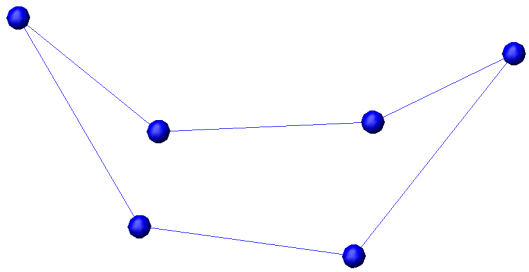


Figura 3: Fragmento en forma de anillo

Se obtienen de la CSD las coordenadas fraccionarias y los parámetros de celda asociados a cada fragmento. De estos datos, se pueden deducir las coordenadas de los distintos átomos en un sistema cartesiano ortonormal (ver sección 3.3.1 para más detalles).

En este trabajo, estos son los datos cristalográficos de partida.

1.2.2. Superposición 3D de dos fragmentos para deducir una medida de distancia entre sus dos conformaciones geométricas

Los métodos que se basan en una comparación gráfica de las posiciones en el espacio de dos moléculas requieren un juicio visual, y no son apropiados para caracterizar un número grande de fragmentos. Por ello es mejor utilizar un método automático, que asocie un número al concepto de "similitud en el espacio".

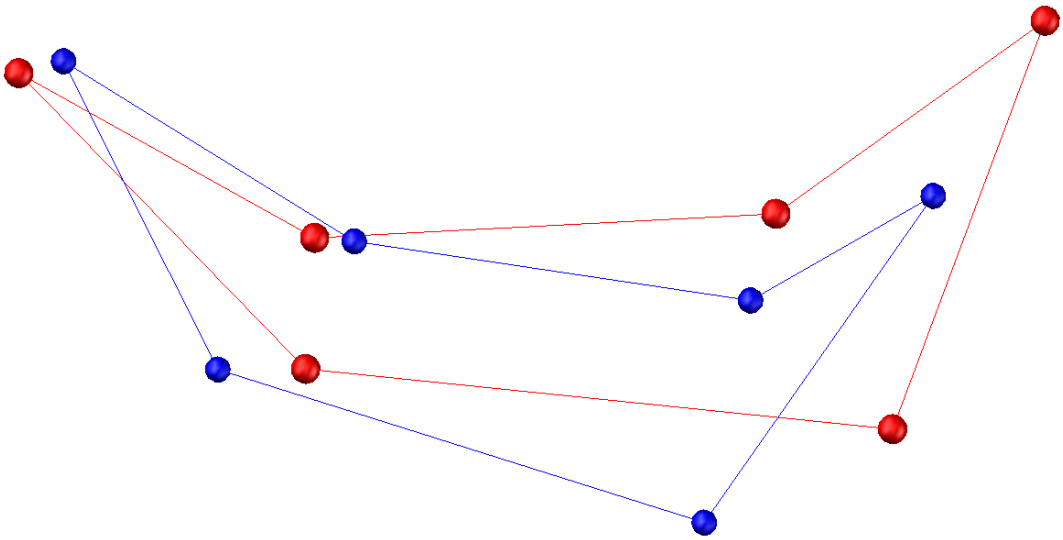


Figura 4: Superposición de dos fragmentos en el espacio

En este trabajo, nos centraremos en un algoritmo de superposición 3D de fragmentos en forma de anillos, inicialmente descrito en [12]. Dados dos fragmentos, el algoritmo busca un sistema cartesiano común en el que éstos queden los más próximos posible.

Para ello, hay que empezar por definir una medida de distancia entre dos fragmentos para los cuales tenemos las coordenadas de sus átomos en un sistema cartesiano concreto.

Dos moléculas de N átomos se sitúan en un origen común. Los átomos de cada una se numeran de 1 a N, y a cada pareja de átomos con el mismo número (átomos coincidentes) les separa una distancia determinada. La media de estas distancias es lo que llamamos “distancia media interatómica”.

Para el ejemplo, sería:

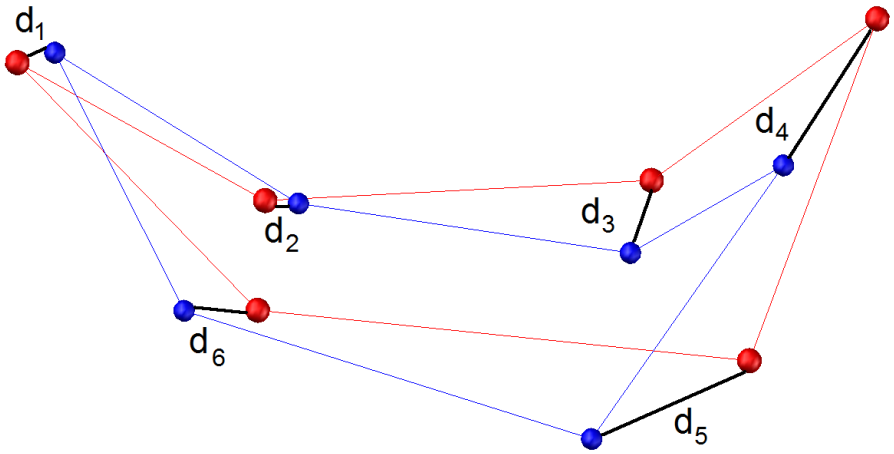


Figura 5: Distancias entre átomos coincidentes (con el mismo número)

$$\bar{d} = \frac{d_1 + d_2 + d_3 + d_4 + d_5 + d_6}{6}$$

En segundo lugar, una vez que hemos escogido cómo medir la distancia entre dos fragmentos descritos en un sistema cartesiano común, buscaremos, dados dos fragmentos, trasladar y rota el uno respecto del otro, para encontrar su posición de mínima distancia, o lo que es lo mismo, la de máxima similitud geométrica.

1.3. Objetivos

La meta final del presente proyecto es implementar el algoritmo descrito en [12] capaz de calcular las distancias medias interatómicas mínimas entre parejas de moléculas de un conjunto de datos dado.

Para ello se escoge un lenguaje de programación de nivel medio: C, que brinda una alta eficiencia, aprovecha muy bien las capacidades de de una computadora y es capaz de realizar muchas operaciones en poco tiempo. .

Se compararán los tiempos de ejecución con los del mismo algoritmo implementado en R [14].

Los datos cristalográficos, tomados de la CSD, se introducirán al programa en forma de archivo de texto, y a su vez se creará otro archivo de texto con los resultados obtenidos.

Obtendremos la matriz de distancias de un conjunto de moléculas. En ella se encuentra la distancia por parejas entre todas las moléculas. Por ejemplo, para un conjunto de cinco fragmentos, la matriz tendría la siguiente forma:

$$\begin{pmatrix} 0 & d_{1,2} & d_{1,3} & d_{1,4} & d_{1,5} \\ & 0 & d_{2,3} & d_{2,4} & d_{2,5} \\ \vdots & & 0 & d_{3,4} & d_{3,5} \\ & & & 0 & d_{4,5} \\ 0 & & \dots & & 0 \end{pmatrix}$$

Donde $d_{i,j}$ representa la distancia media interatómica entre los fragmentos i y j.

1.4. Aplicaciones

Esta matriz de distancias puede utilizarse como comienzo de métodos estadísticos como escalado multidimensional o conglomerado aglomerativo, que proporcionan una información muy valiosa para el análisis conformacional del conjunto de datos.

1.5. Plan de trabajo

A continuación se detalla la estrategia seguida en el proyecto.

1. **Repaso y ampliación de los conocimientos sobre C**: Debido a que la asignatura de Fundamentos de Informática se cursa en primero, hubo que revisar lo aprendido y complementarlo. Para esto se utilizó como texto base [15]. Se aprendió a: leer y escribir datos con archivos de texto, manejar librerías, trabajar con diferentes punteros y funciones, etc.

2. **Cálculo de las coordenadas intrínsecas**: Lo primero fue crear la matriz I_F de un fragmento. Para ello se aplicaron las herramientas que se detallan en 3.3.1.

3. **Comprobación de resultados con Matlab ®**: Los resultados obtenidos para I_F se comprobaron con el programa de cálculo matemático Matlab. También se crearon códigos en este programa para comprobar todas y cada una de las operaciones posteriores. Al ser un programa de alto nivel basado en cálculo matricial, era muy sencillo obtener los resultados correctos de las diferentes multiplicaciones de matrices, y corroborar que los resultados en principio inseguros del código en C eran correctos.

4. **Distancia entre dos moléculas**: A partir de las coordenadas intrínsecas de dos fragmentos, se probó el algoritmo para solamente ellos, consiguiendo así su distancia media interatómica: d.

5. **Representación gráfica de** d: Se creó un código en Matlab que permitía visualizar todas las curvas de d en función del ángulo de giro γ. Éstas fueron la base para entender cómo variaban los mínimos de la distancia y gráficamente comprobar que el punto más bajo de las curvas, el mínimo real, correspondía por el obtenido con nuestro algoritmo.

6. **Ampliación a todo un conjunto de datos**: Una vez comprobado el algoritmo con distintas parejas de fragmentos, se pasa a crear una función que calcule la distancia d, de una a otra para todo un conjunto de moléculas.

7. **Creación de una interfaz**: Por último, se decidió para que el programa pudiera utilizarlo cualquier persona, estuviera o no familiarizada con la informática, que los distintos parámetros que necesita el programa para funcionar (número de átomos de los fragmentos, fichero de origen de datos...) se introdujeran por pantalla directamente después de ejecutar el programa y crear un fichero leeme.txt para explicar cómo utilizarlo.

2. Presentación de la superposición

2.1. Idea básica

Para obtener una medida de la distancia entre dos anillos, procederemos de la siguiente forma:

1. Definir para cada anillo su plano medio como lo establece [7].

2. Hacer coincidir los dos planos.

3. Normalizar los dos fragmentos, es decir, encoger o expandir cada anillo para que su distancia media interatómica sea uno.

4. Buscar la rotación óptima a lo largo del eje normal al plano medio común, para que los dos fragmentos estén lo más cerca posible de acuerdo con el criterio de los mínimos cuadrados. En este paso se debe de tener en cuenta que, para cada anillo, el átomo por el que se empieza a numerar es arbitrario, y que las imágenes especulares deben tomarse como equivalentes.

5. Finalmente, de acuerdo con la configuración obtenida en el paso 4, calcular la distancia entre fragmentos.

2.1.1. Coordenadas “Intrínsecas”

Tomaremos como referencia un anillo de N átomos: $A_1, A_2 \dots A_N$, que llamaremos fragmento F. Asumimos que F es una matriz $N \times 3$ que contiene por filas las coordenadas (x, y, z) de sus N átomos en un sistema cartesiano dado. También asumimos que el origen de este sistema es el centro geométrico de F.

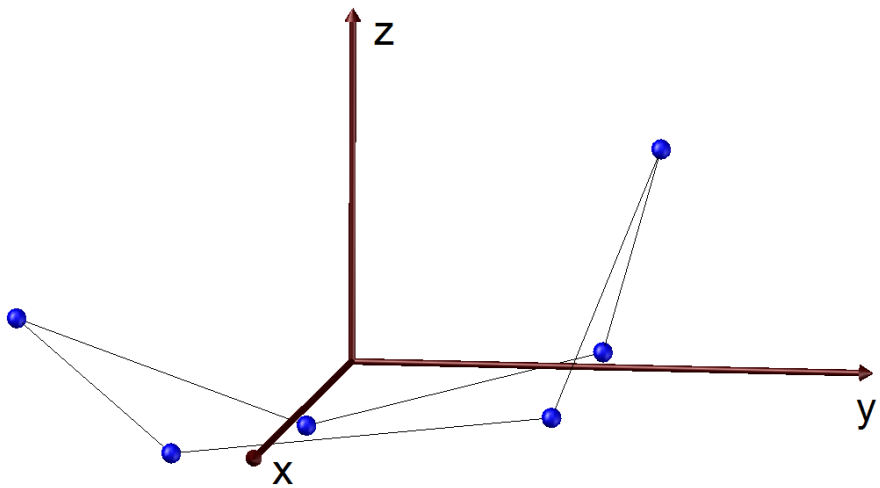
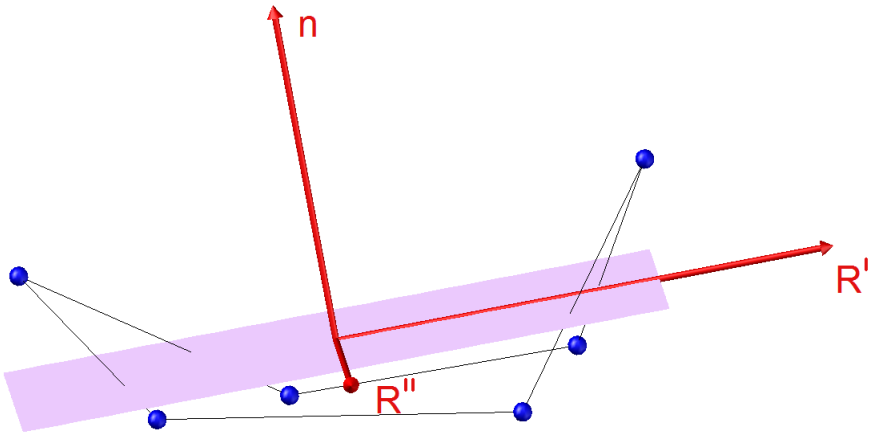


Figura 6: Fragmento de 6 átomos con el origen de coordenadas en su centro geométrico

La definición de sistema “intrínseco” está inspirada en las coordenadas puckering de [7]. Para establecer la definición de plano medio, [7] usa las relaciones (8),(9) y (10) de la pág 1355, que establecen:

$$\begin{aligned}\mathbf{R}' &= \sum_{j=1}^{N} \mathbf{R_j} \sin[\mathbf{2}\pi(\mathbf{j}-\mathbf{1})/\mathbf{N}] \\ \mathbf{R}'' &= \sum_{j=1}^{N} \mathbf{R_j} \cos[\mathbf{2}\pi(\mathbf{j}-\mathbf{1})/\mathbf{N}] \\ \mathbf{n} &= \mathbf{R}' \times \mathbf{R}''/|\mathbf{R}' \times \mathbf{R}''|.\end{aligned}$$



Donde j es el número del átomo, $\mathbf{R_j}$ ($j = 1, \ldots, N$) corresponde a los vecotres de posición de los N átomos del fragmento en un sistema cartesiano dado, con origen en el centro geométrico del anillo, y $\mathbf{n}$ es ortonormal al plano medio.

Diremos que una terna de vectores ortonormales $(\mathbf{i}, \mathbf{j}, \mathbf{k})$ con origen en el centro de F define un sistema cartesiano "intrínseco" para el fragmento F si $\mathbf{k}$ coincide con el vector unitario $\mathbf{n}$, y que puede ser el opuesto, definido en [7], y si el plano generado por **(i,j)** coincide con el denominado plano medio, definido por los vectores $\mathbf{R}'$ y $\mathbf{R}''$.

Hay por lo tanto infinitos sistemas intrínsecos para cada molécula, pero en todos ellos, el plano (x, y) así como el eje z son siempre el mismo para cada anillo. Además, el plano medio no varía si en vez de tener el fragmento como $A_1 A_2 \ldots A_N$:

- Se cambia la numeración de dos átomos consecutivos: tenemos $A_2 A_3 \ldots A_N A_1$
- Se cambia la dirección en la que se numeran los átomos: tenemos $A_1 A_N A_{N-1} \ldots A_3 A_2$.
- Se toma la imagen especular del fragmento.

Las coordenadas atómicas se han establecido pues de una forma arbitraria. Es decir, dos fragmentos con una disposición en el espacio exactamente igual, pueden presentar unas coordenadas If distintas, debido a las distintas arbitrariedades que se han cometido a la hora de definir su configuración.

Matemáticamente este hecho se asocia a la multiplicación de la matriz de coordenadas por una serie de matrices auxiliares que pasamos a describir.

2.2. Matrices auxiliares

En este apartado describiremos las matrices básicas que se usan para expresar las transformaciones de un fragmento dado (rotación, cambio de numeración de los átomos, etc...)

2.2.1. Matriz de coordenadas "intrínsecas" normalizadas I_F

Dos moléculas pueden tener exactamente la misma forma, pero que una sea una copia a escala de la otra, por eso es conveniente pasar las moléculas a la misma escala. Esto se realiza, dividiendo las coordenadas de cada uno de los átomos por su distancia media interatómica. De esta forma conseguiremos encoger o expandir un fragmento hasta que su distancia media sea igual a 1. Tendremos así unas coordenadas "intrínsecas" normalizadas:

$$I_F = M_F/\bar{d}.$$

donde M_F es la matriz de coordenadas de los átomos de F en un sistema cartesiano "intrínseco" y $\bar{d}$ es la media de las distancias entre dos átomos consecutivos:

$$\bar{d} = \frac{d_{1,2} + d_{2,3} + \ldots + d_{N-1,N} + d_{N,1}}{N}$$

$$d_{i,j} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}$$

2.2.2. La matriz de traslación: T

Consideremos la matriz $N \times N$

$$T = \begin{pmatrix} 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & 1 \\ 1 & 0 & 0 & 0 & \cdots & 0 \end{pmatrix}$$

La matriz $T \cdot I_F$ contiene las coordenadas del mismo fragmento, pero como si hubiéramos empezado a nombrar por el átomo 2, es decir, considerando el anillo $A_2A_3 \ldots A_NA_1$.

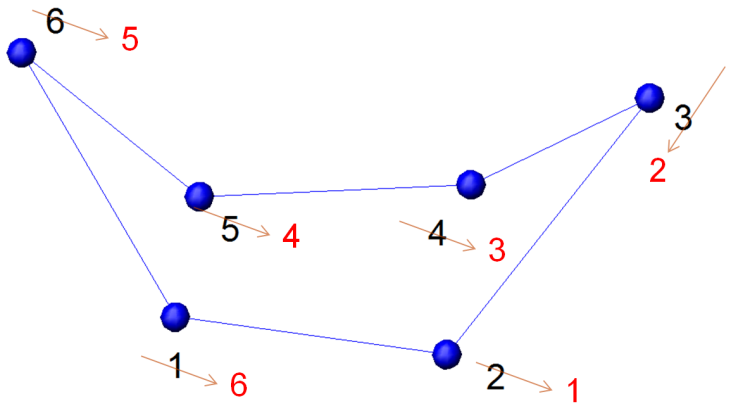


Figura 7: Aplicación de la matriz T a un fragmento dado

Tomaremos como ejemplo de aplicación de esta matriz y de las siguientes, para ver cómo actúan exactamente, el anillo de 6 átomos:

$$I_F = \begin{pmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \\ x_4 & y_4 & z_4 \\ x_5 & y_5 & z_5 \\ x_6 & y_6 & z_6 \end{pmatrix}$$

De esta forma tenemos:

$$T \cdot I_F = \begin{pmatrix} x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \\ x_4 & y_4 & z_4 \\ x_5 & y_5 & z_5 \\ x_6 & y_6 & z_6 \\ x_1 & y_1 & z_1 \end{pmatrix} \qquad T^2 \cdot I_F = \begin{pmatrix} x_3 & y_3 & z_3 \\ x_4 & y_4 & z_4 \\ x_5 & y_5 & z_5 \\ x_6 & y_6 & z_6 \\ x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \end{pmatrix} \qquad T^N \cdot I_F = I_F$$

2.2.3. La matriz de cambio de dirección: D

Consideremos la matriz $N \times N$

$$D = \begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 & 1 \\ 0 & \cdots & 0 & 0 & 1 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 1 & 0 & 0 & \cdots & 0 \end{pmatrix}$$

La matriz $D \cdot I_F$ contiene las coordenadas del fragmento pero cambiando la dirección en la que se nombran los átomos, es decir considerando el anillo $A_1 A_N A_{N-1} \ldots A_2$.

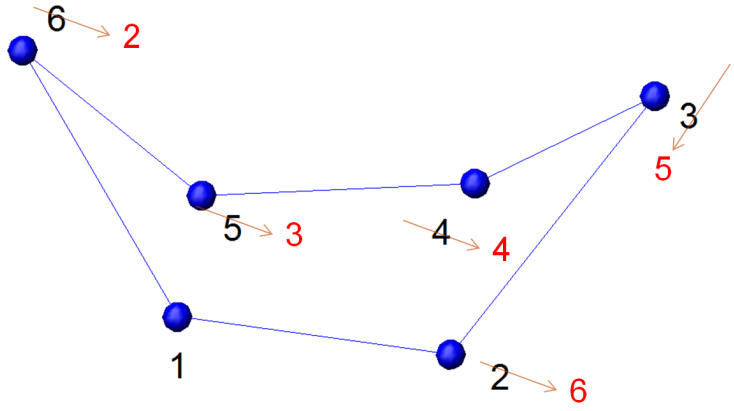


Figura 8: Aplicación de la matriz D a un fragmento dado

$$D \cdot I_F = \begin{pmatrix} x_1 & y_1 & z_1 \\ x_5 & y_5 & z_5 \\ x_6 & y_6 & z_6 \\ x_4 & y_4 & z_4 \\ x_3 & y_3 & z_3 \\ x_2 & y_2 & z_2 \end{pmatrix}$$

2.2.4. La matriz espejo M

Consideremos la matriz 3×3

$$M = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix}$$

La matriz $I_F \cdot M$ contiene las coordenadas de la imagen especular del fragmento F en un sistema cartesiano “intrínseco”.

$$I_F \cdot M = \begin{pmatrix} x_1 & y_1 & -z_1 \\ x_2 & y_2 & -z_2 \\ x_3 & y_3 & -z_3 \\ x_4 & y_4 & -z_4 \\ x_5 & y_5 & -z_5 \\ x_6 & y_6 & -z_6 \end{pmatrix}$$

2.2.5. La matriz J

Consideremos la matriz 3×3

$$J = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & -1 \end{pmatrix}$$

La matriz $I_F \cdot J$ contiene las coordenadas del mismo fragmento, pero en el sistema formado por $(\vec{y}, \vec{x}, -\vec{z})$, que es también un sistema cartesiano intrínseco pero con el sentido opuesto de $\vec{z}$.

$$I_F \cdot J = \begin{pmatrix} y_1 & x_1 & -z_1 \\ y_2 & x_2 & -z_2 \\ y_3 & x_3 & -z_3 \\ y_4 & x_4 & -z_4 \\ y_5 & x_5 & -z_5 \\ y_6 & x_6 & -z_6 \end{pmatrix}$$

2.2.6. La matriz de rotación $R_z(\gamma)$

Consideremos para $\gamma \in \mathbf{R}$, la matriz 3×3

$$R_z(\gamma) = \begin{pmatrix} \cos(\gamma) & \sin(\gamma) & 0 \\ -\sin(\gamma) & \cos(\gamma) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

La matriz $I_F R_z(\gamma)^T$ contiene las coordenadas del fragmento, girado a lo largo del eje z usando un ángulo γ.

2.2.7. Acción combinada de matrices

Tendremos primero en cuenta las posibles simetrías topológicas de los datos. Así, dos moléculas pueden ser iguales físicamente, pero parecer diferentes analíticamente porque sus átomos hayan sido nombrados en direcciones diferentes o porque sean imágenes especulares una de la otra, ver por ejemplo la referencia [11].

Siguiendo [11], consideramos un anillo de N átomos. La secuencia de símbolos asociada se denota con $\mathsf{S}_1\mathsf{S}_2 \cdots \mathsf{S}_N$. Si, por ejemplo, se han tomado del CSD las coordenadas de ciclohexanos, la secuencia de átomos asociada es `CCCCCC`, que permanece invariante aunque se cambie el átomo inicial a la hora de nombrarlos. Sin embargo, si se toman los datos cristalográficos de un complejo en anillo, con dos metales y dos enlaces de fósforo, la secuencia será `MOPOMOPO`, que no es invariante con el átomo inicial al nombrar.

Se introduce así el subconjunto $\mathcal{S}$ de $\{1, \ldots, N\}$, que contiene los posibles puntos iniciales, que no cambian globalmente la secuencia de símbolos $\mathsf{S}_1\mathsf{S}_2 \cdots \mathsf{S}_N$.

En el caso de los ciclohexanos tendremos claramente $\mathcal{S} = \{1, \ldots, 6\}$, todas las posiciones de los carbonos, mientras que para los anillos `MOPOMOPO`, tenemos $\mathcal{S} = \{1, 5\}$, sólo las posiciones de los metales.

Consideremos dos fragmentos F_1 y F_2, y calculemos sus matrices de coordenadas intrínsecas normalizadas: I_{F_1} e I_{F_2} respectivamente. Éstos serán equivalentes en cuanto a su conformación geométrica si, cambiando la dirección al nombrar, tomando la imagen especular o cambiando el sistema intrínseco considerado y calculando las matrices resultantes, éstas coinciden. Concretamente, para comparar F_1 y F_2, se premultiplicará I_{F_2} por T y D y se postmultiplicará por M, J y $R_z(\gamma)$ para compararlo con I_{F_1}.

2.3. Definición de la distancia

Podemos definir ahora una medida de la distancia entre dos fragmentos, siendo I_{F_1} e I_{F_2} dos matrices cualesquiera de coordenadas intrínsecas normalizadas para F_1 y F_2 respectivamente:

$$d(F_1, F_2) = \min_{\substack{s\in\mathcal{S}\\ v,a,b\in\{0,1\}}} \min_{\gamma\in[0,2\pi[} \|I_{F_1} - (T^{(s-1)}D^v I_{F_2} M^a J^b)(R_z(\gamma))^T\|$$

donde $\mathcal{S}$ se definió en 2.2.7 y, para una matriz A $n \times m$, $\|A\|$ es una norma cualquiera para matrices. La cantidad d(F_1,F_2) será nuestra propuesta de medida de la distancia entre F_1 y F_2.

NOTA: La definición de norma que tomaremos será:

$$\|A\| = \frac{1}{n}\sum_{i=1}^{N}\sqrt{a_{i,1}^2 + a_{i,2}^2 + a_{i,3}^2}$$

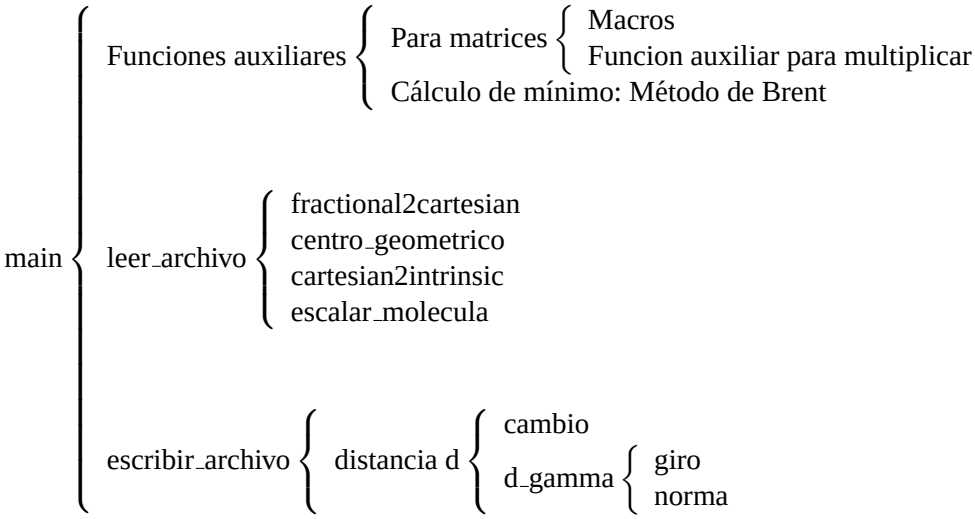
Se ha tomado esta definición y no otra porque así se le da un sentido físico: si A_1 and A_2 contienen las coordenadas de dos fragmentos F_1 y F_2, la norma $\|A_1 - A_2\|$ es el valor promedio de la distancia entre dos pares de átomos coincidentes.

Como consecuencia, teniendo en cuenta que cada fragmento normalizado tiene una distancia interatómica igual a uno, si obtenemos, por ejemplo d(F_1,F_2)=0.05, quiere decir que, después de una rotación apropiada, la distancia media entre dos pares de átomos coincidentes es sólo del 5 % de su longitud interatómica media, con lo cual sus formas geométricas están muy próximas.

3. Programación

El código ha sido escrito en el lenguaje de programación C. Se ha utilizado el Entorno de Desarrollo Integrado (IDE) Dev-C++ 4.9.9.2. Dicho entorno es de distribución gratuita (http://www.bloodshed.net/devcpp.html), funciona bajo Microsoft Windows y utiliza el compilador GCC.

El programa tiene la siguiente estructura:



Escribiendo en pseudocódigo, sería:

```
introducción de parámetros


leer archivo
    for 1:j:n
        fractian2cartesian (j)
        centro_geometrico (j)
        cartesian2intrinsic (j)
        escalar_molecula (j)    ==> If(j): Coordenadas
                                           intrínsecas

escribir archivo
    for 1:i:n
        for (i+1):j:n
            distancia If(i,j)  ==> fichero salida: matriz
                                                   distancias
```

Es decir: se introducen los parámetros por pantalla, el programa lee una a una las moléculas y las guarda en la matriz If. Después lee todas las parejas distintas de esta If, calcula su distancia y la escribe en un fichero de texto.

3.1. Función principal: main

Como todos los programas en C, al ejecutarse comienza por la función `main()`, que irá llamando al resto de funciones.

En nuestro caso, la función principal, primero pide al usuario que introduzca los parámetros de las moléculas que se van a evaluar, después lee el archivo de origen de datos, y por último calcula su matriz de distancias y la escribe en un archivo de texto.

Los parámetros se introducen directamente por pantalla, no se pasa ningún argumento a la función. Se debe especificar:

- Número de átomos de las moléculas
- Máximo número de moléculas a procesar

Es necesario conocer estos datos, porque el programa debe reservar un espacio en la memoria del ordenador, donde se guardarán las matrices de cálculo que depende directamente del número de fragmentos a estudiar.

Además, también se debe conocer:

- Valores posibles de posición inicial *s*

s corresponde a las posiciones en las que se puede comenzar a leer la molécula.

```
171: /* Lo primero que debe hacer el usuario es introducir los parámetros que va a utilizar
172:    el programa */
173:
174: printf("\n\nIntroduzca por favor los parametros a utilizar:\n\n\n");
175:
176: printf("Numero de atomos de las moleculas: ");
177: scanf("%d",&N);
178:
179: printf("\n\nMaximo numero de moleculas a procesar: ",max);
180: scanf("%d",&max);
181:
182: printf("\n\nNumero de valores posibles de posicion inicial s:  ");
183: scanf("%d",&size_s);
184:
185: S=(int *) malloc(sizeof(int)*size_s);  /* Asignar al puntero S una direccion de memoria
186:                                            donde guardara las posibles posiciones s */
187:
188: printf("\n1er valor posible de s: ");
189: scanf("%d",S);
190: for(i=2;i<=size_s;i++){
191:        printf("%do valor posible de s: ",i);
192:        scanf("%d",S+i-1);}
193:
194: /* El usuario ya ha definido todos los parámetros del programa */
```

La función `cambio()` utiliza las matrices D, M y J (ver 2.2). Recordemoslas:

$$D = \begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 & 1 \\ 0 & \cdots & 0 & 0 & 1 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 1 & 0 & 0 & \cdots & 0 \end{pmatrix} \quad M = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix} \quad J = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & -1 \end{pmatrix}$$

Estas matrices se definen una sola vez en la funcion `main`, en vez de hacerlo en la función donde se utilizan, pasando a ser variables globales. De esta forma se consigue acortar el tiempo de ejecución del programa, ya que `cambio` se ejecuta muchísimas veces dentro del algoritmo.

```
199: // Definición de las variables globales que se utilizan en cambio() ////////////////////
200:
201: D=(float *)malloc(sizeof(float)*N*N);  // Asignar al puntero D una direccion de memoria
202:
203: for(i=0; i<N; ++i)            //  Definición de D
204:        for(j=0; j<N; ++j)     //
205:                *(D+i*N+j)=0;  //
206: int D00=1;                    //
207: *D=D00;                       //
208: for(i=1; i<N; ++i)            //
209:        *(D+(N-i)*N+i)=1;      //
210:
211: for(i=0; i<3; ++i)                            //  Definición de M
212:        for(j=0; j<3; ++j)                     //
213:                if((i==j)&(i!=3)) M[i][j]=1;   //
214:                else M[i][j]=0;                //
215: M[2][2]=-1;                                  //
216:
217: for(i=0; i<3; ++i)           //  Definición de J
218:        for(j=0; j<3; ++j)    //
219:                J[i][j]=0;    //
220: J[0][1]=J[1][0]=1;           //
221: J[2][2]=-1;                  //
```

3.2. Funciones auxiliares

Para trabajar con las matrices presentes en el problema, se han utilizado unas macros que facilitan las operaciones de vectores de tres componentes y una función auxiliar para multiplicar dos matrices de dimensiones cualesquiera.

3.2.1. Macros para vectores de tres componentes

Las macros automatizan las siguientes operaciones:

- `crossProduct(a,b,c)`:

$$\vec{a} = \vec{b} \times \vec{c}$$

```
20: // Producto vectorial: a = b x c
21: #define crossProduct(a,b,c) \
22:       (a)[0] = (b)[1] * (c)[2] - (c)[1] * (b)[2]; \
23:       (a)[1] = (b)[2] * (c)[0] - (c)[2] * (b)[0]; \
24:       (a)[2] = (b)[0] * (c)[1] - (c)[0] * (b)[1];
```

- `dotProduct(a,b,c)`:

$$\vec{a} = \vec{b} \cdot \vec{c}$$

```
26: // Producto escalar: a = b · c
27: #define dotProduct(a,b,c) \
28:       (a)= (b)[0] * (c)[0] + (b)[1] * (c)[1] + (b)[2] * (c)[2];
```

- `normalize(v\_n,v)`:

$$\vec{v_n} = \frac{\vec{v}}{\| \vec{v} \|}$$

```
30: // Normalización de un vector: v_n = v / |v|
31: #define normalize(v_n,v) \
32:       (v_n)[0] = (v)[0] / pow((v)[0]*(v)[0]+(v)[1]*(v)[1]+(v)[2]*(v)[2],0.5); \
33:       (v_n)[1] = (v)[1] / pow((v)[0]*(v)[0]+(v)[1]*(v)[1]+(v)[2]*(v)[2],0.5); \
34:       (v_n)[2] = (v)[2] / pow((v)[0]*(v)[0]+(v)[1]*(v)[1]+(v)[2]*(v)[2],0.5);
```

- `scalar(a,constant,b)`:

$$\vec{a} = constant \times \vec{b}$$

```
36: // Multiplicación de un vector por un escalar: a = constant * b
37: #define scalar(a,constant,b) \
38:       (a)[0] = constant*(b)[0]; \
39:       (a)[1] = constant*(b)[1]; \
40:       (a)[2] = constant*(b)[2];
```

- `subtraction(a,b,c)`:

$$\vec{a} = \vec{b} - \vec{c}$$

```
42: // Resta de dos vectores: a = b - c
43: #define subtraction(a,b,c) \
44:       (a)[0] = (b)[0] - (c)[0]; \
45:       (a)[1] = (b)[1] - (c)[1]; \
46:       (a)[2] = (b)[2] - (c)[2];
```

Los datos de entrada son siempre matrices 1x3. Estas operaciones se utilizarán para calcular el plano medio de las móléculas.

3.2.2. Función auxiliar para multiplicar matrices

La función se define como:
`void multYXZ(float *A1, float *A2, float *A3, int Y, int X, int Z)`
Y calcula la operación:

$$A_1 \times A_2 = A_3$$

Donde A_1 es una matriz YxX, A_2 es una matriz XxZ y obviamente A_3 es YxZ.

En este caso, los datos de entrada y salida son punteros, que apuntan al primer elemento de cada matriz.

```
789: /**** Multiplica dos matrices:  a1[Y][X] * a2[X][Z] = a3[Y][Z] =  ****/
790: void multYXZ(float *a1, float *a2, float *a3, int Y, int X, int Z) //a1*a2=a3
791: {
792: int i,j,k;
793: float temp;
794:        for(i=0; i<Y; i++) //Filas de la izquierda
795:               for(j=0; j<Z; j++) //Columnas de la derecha
796:                      {temp=0;
797:                      for(k=0; k<X; k++) //Columnas de la izq=Filas de la derecha
798:                             temp+=(*(a1+i*X+k))*(*(a2+k*Z+j));
799:                      *(a3+i*Z+j)=temp;
800:                      }
```

3.2.3. Librería de cálculo del mínimo de una función de una variable: Método de Brent

Esta librería, que nos da el mínimo de $d(\gamma)$ utilizando la interpolación parabólica, está tomada de [1].

Se define como:

```
float brent(float ax, float bx, float cx, float (*f)(float)
,float tol, float *xmin)
```

donde:

ax, bx, cx Son los puntos iniciales donde buscar el mínimo. Deben cumplir ax $<$ bx $<$ cx. En nuestro caso estos puntos deben de estar entre 0 y 2π.

(*f)(float) Es el puntero a la función

tol Es la tolerancia, la precisión que se desea obtener en el mínimo

$*$**min** Es un puntero donde se guardará el valor de la variable (γ para nosotros) donde se alcanza el mínimo

El valor devuelto por la función, es el de $f(min)$, para nosotros: $d(\gamma_{min})$.

```
716: /*  Definicion de Brent: Búsqueda del mínimo de f a partir de la distancia aúrea */
717:
718: float brent(float ax, float bx, float cx, float (*f)(float), float tol, float *xmin)
719:
720: /* Dada una función f, y tres puntos de inicio: ax<bx<cx, esta rutina encuentra el mínimo
721: con una precisión aproximada de tol, usando el método de Brent (interpolación parabólica)
722: La abscisa del mínimo es xmin y el valor mínimo es el float devuelto por la funcion */
723:
724:
725: {
726: int iter;
727: float a,b,d,etemp,fu,fv,fw,fx,p,q,r,tol1,tol2,u,v,w,x,xm;
728: float e=0.0;
729: a=(ax < cx ? ax : cx);
730: b=(ax > cx ? ax : cx);
731: x=w=v=bx;
732: fw=fv=fx=(*f)(x);
733: for (iter=1;iter<=ITMAX;iter++)
734:     {
735:     xm=0.5*(a+b);
736:     tol2=2.0*(tol1=tol*fabs(x)+ZEPS);
737:
738:     if (fabs(x-xm) <= (tol2-0.5*(b-a))) {
739:        *xmin=x;
740:        return fx;}
741:
742:     if (fabs(e) > tol1) {
743:        r=(x-w)*(fx-fv);
744:        q=(x-v)*(fx-fw);
745:        p=(x-v)*q-(x-w)*r;
746:        q=2.0*(q-r);
747:        if (q > 0.0) p = -p;
748:        q=fabs(q);
749:        etemp=e;
750:        e=d;
751:        if (fabs(p) >= fabs(0.5*q*etemp) || p <= q*(a-x) || p >= q*(b-x))
752:        d=CGOLD*(e=(x >= xm ? a-x : b-x));
753:        else {
754:             d=p/q;
755:             u=x+d;
756:             if (u-a < tol2 || b-u < tol2)
757:                d=SIGN(tol1,xm-x);
758:             }
759: } else {
760:   d=CGOLD*(e=(x >= xm ? a-x : b-x));
761: }
```

```
762: u=(fabs(d) >= tol1 ? x+d : x+SIGN(tol1,d));
763: fu=(*f)(u);
764: if (fu <= fx) {
765:    if (u >= x) a=x; else b=x;
766:    SHFT(v,w,x,u)
767:    SHFT(fv,fw,fx,fu)
768: } else {
769:        if (u < x) a=u; else b=u;
770:        if (fu <= fw || w == x) {
771:           v=w;
772:           w=u;
773:           fv=fw;
774:           fw=fu;
775: } else if (fu <= fv || v == x || v == w) {
776:        v=u;
777:        fv=fu;
778: }
779: }
780: }
781: printf("Too many iterations in brent");
782: }
783: /*  Fin de Brent */
```

3.3. Rutina leer_archivo

La primera acción que debe hacer el programa es leer de un archivo de texto las coordenadas fraccionarias de todas las moléculas a estudiar, y pasarlas a coordenadas intrínsecas. El esquema de actuación será el siguiente:

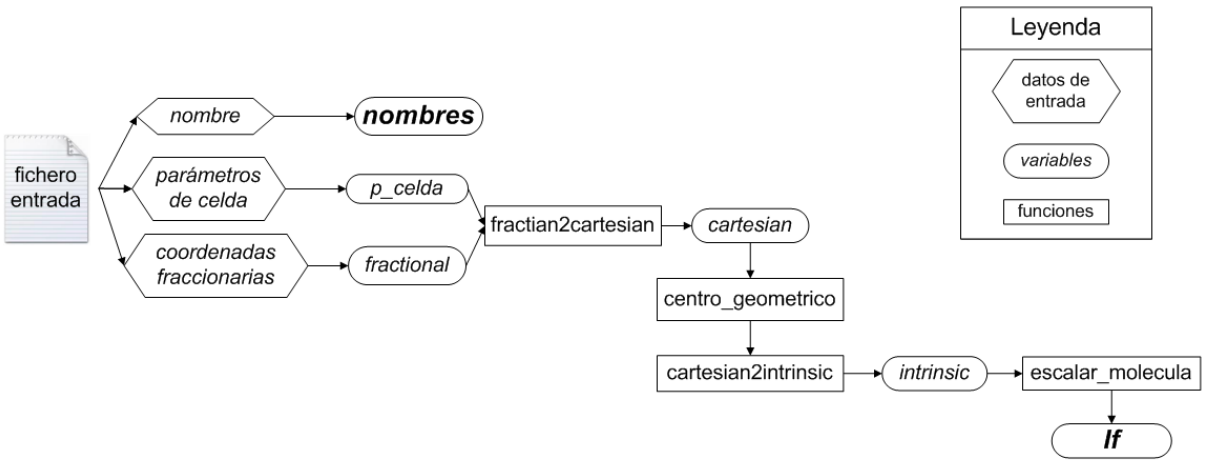


Figura 9: Esquema de la función leer_archivo

Este esquema se repite para cada molécula, es decir, se lee la primera molécula, se le aplica el bucle anterior y se guardan en memoria su nombre y sus coordenadas intrínsecas. Para ello, el proceso es el siguiente:

1. Leer y guardar el nombre de la molécula
2. Leer los parámetros de celda y las coordenadas fraccionarias. Pasar estos datos a la función `fractional2cartesian` y obtener la matriz de coordenadas cartesianas `cartesian`. Para saber más sobre coordenadas fraccionarias, ver 3.3.1.
3. Aplicarle a esta matriz la función `centro_geométrico` para hacer el cambio de origen al centro de la molécula.

4. Ejecutar la funcion `cartesian2intrinsic` para calcular la matriz de coordenadas intrínsecas `intrinsic`.

5. Pasar la molécula a escala (hacer que su distancia interatómica sea igual a 1) a través de la función `escalar_molecula` y obtener así la matriz `If`.

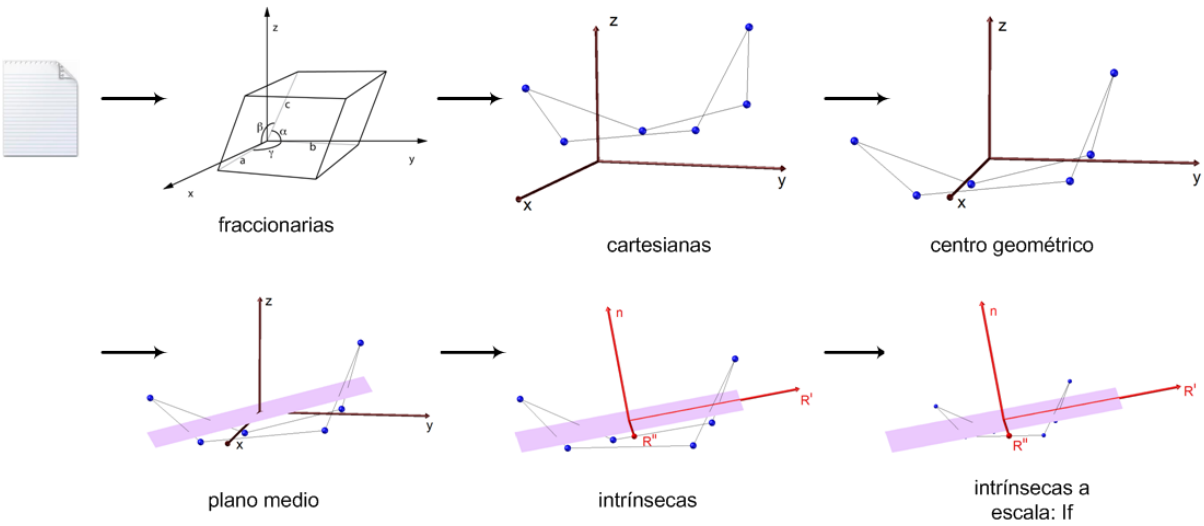


Figura 10: Etapas de la función leer_archivo

Acto seguido, se lee la siguiente molécula, se le aplica la misma secuencia de procedimientos y se guarda, su nombre en la siguiente fila de la matriz `nombres` y sus coordenadas intrínsecas obtenidas en la siguiente dirección de memoria de If. Su estructura es:

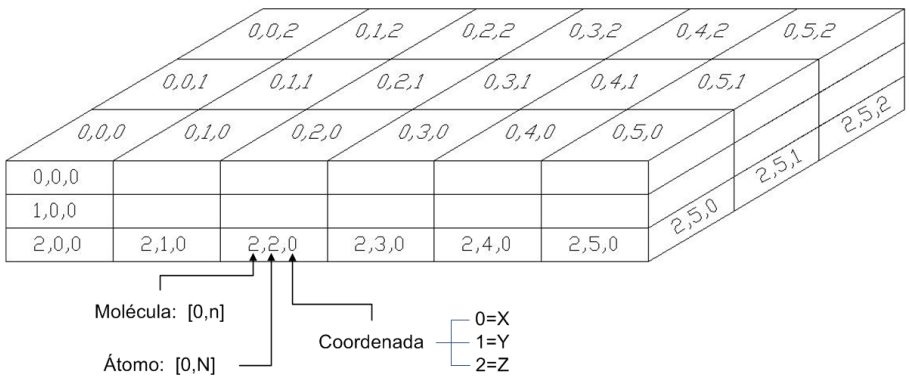


Figura 11: Matriz de coordenadas intrínsecas normalizada

En C los índices de las matrices comienzan por cero, de modo que `If[5][4][2]` corresponde a la coordenada z, del quinto átomo de la sexta molécula.

Pasemos ahora a analizar paso a paso la función `leer_archivo()`:

Primero se asigna una dirección en memoria a las matrices que se van a utilizar:

```
283: nombres=(char *) malloc(sizeof(char)*max*long_max_nombre); // Nombres de las moléculas
284: fractional=(float *) malloc(sizeof(float)*N*3);  // Coordenadas fraccionarias
285: cartesian=(float *) malloc(sizeof(float)*N*3);  // Coordenadas cartesianas
286: intrinsic=(float *) malloc(sizeof(float)*N*3);  // Coordenadas "intrínsecas"
287: If=(float *) malloc(sizeof(float)*N*max*3);     /* Coordenadas "intrínsecas" referidas al
288:                                                    centro geométrico */
```

Se define una variable contador que recorrerá las moléculas que se leen y un vector donde se guardarán los parámetros de celda: $[a; b; c; \alpha; \beta; \gamma]$

```
290: k=0;  // Molécula que se lee
291: float celda[6];  // Parámetros de celda
```

Después se le pide al usuario que introduzca el nombre del fichero de texto donde están guardadas las coordenadas fraccionarias, y se comprueba que el fichero existe:

```
294: FILE *entrada;
295: char fichero_entrada[241];
296:
297: printf("\n\nFichero de datos de entrada: ");
298: scanf("%s",fichero_entrada);
299:
300: /* Comprobamos que el fichero introducido por el usuario, se encuentra en el mismo
301:    directorio que el programa */
302:
303: while ((entrada=fopen(fichero_entrada,"r"))==NULL){
304:    printf("\nEl archivo no existe. Especifique otro: ");
305:    scanf("%s",fichero_entrada);
306:    }
```

Los datos de entrada no pueden estar escritos de una forma cualquiera, para que sean leídos correctamente, deben de tener el siguiente orden:

Nombre_Molecula_1;a;b;c;α;β;γ;x_1;y_1;z_1;x_2;y_2;z_2; (...);x_N;y_N;z_N [Retorno de carro]
Nombre_Molecula_2; (...)

Una vez comprobado que el fichero de entrada de datos existe, se pasa a leerlo. Los parámetros de celda se guardan en la variable `celda[6]`, y las coordenadas fraccionarias en la matriz `fractional`. Estos son los datos de partida para el bucle.

```
310: while(!feof(entrada))
311: {
312: if(entrada==NULL)
313:       printf("Error al abrir el fichero");
314:
315: char caracter;       //Variable temporal de lectura de nombre
316:
317: int u=0;
318: caracter='0';
319:
320:
321: while(caracter !=';')                                  //
322: {                                                      //
323: caracter=getc(entrada);                               //
324: *(nombres+k*long_max_nombre+u)=caracter;               //
325: u++;                                                  //
326: }                                                      //
327:                                                       // Extrae datos del fichero
328: for(i=0;i<5;++i)                                      // origen, copia el nombre
329:        fscanf(entrada, "%f %*c",&celda[i]);           // y cambia a coordenadas
330: fscanf(entrada, "%f", &celda[5]);                     // cartesianas
331:                                                       //
332: for(i=0;i<N;++i)                                      //
333:        for(j=0;j<3;++j)                               //
334:               fscanf(entrada, "%*c %f ", fractional+i*3+j);
```

A partir de aquí se le aplican las sucesivas funciones hasta obtener `If`

```
336: /* A partir de los parámetros de celda y las coordenadas fraccionarias,
337: hallamos las cartesianas */
338: fractian2cartesian(celda,fractional);
339:
340: // Cambiamos el origen de las coordenadas cartesianas al centro geométrico
341: centro_geometrico(cartesian);
342:
343: // Hallamos las coordenadas "intrínsecas"
344: cartesian2intrinsic(cartesian,k);
345:
346: // Dividimos las coordenadas intrinsecas entre la distancia media interatómica */
347: escalar_molecula(intrinsic,k);
```

3.3.1. función void fractional2cartesian (float p_celda[], float *fractional)

En cristalografía, se usan mucho las coordenadas fraccionarias. En este sistema de coordenadas, los bordes de la celda se utilizan como vectores directores para describir la posición de los átomos [8]. La celda unitaria es un paralelepípedo definido por la longitud de sus lados a, b c y los ángulos entre ellos α, β y γ, como se muestra en la figura:

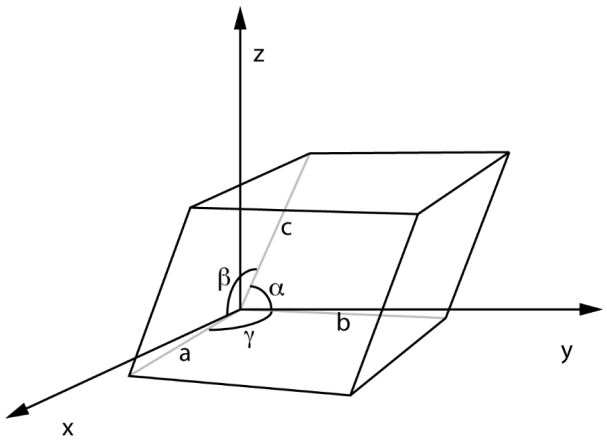


Figura 12: Celda de coordenadas fraccionarias

Lo que hace esta función es, como su propio nombre indica, pasar de coordenadas fraccionarias a cartesianas.

Para ello utiliza la relación:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}_{cart} = \begin{bmatrix} a & b\cos(\gamma) & c\cos(\beta) \\ 0 & b\sin(\gamma) & c\frac{\cos(\alpha)-\cos(\beta)\cos(\gamma)}{\sin(\gamma)} \\ 0 & 0 & c\frac{v}{\sin(gamma)} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}_{fract}$$

donde

$$v = \sqrt{1-\cos(\alpha)^2-\cos(\beta)^2+2\cos(\alpha)\cos(\beta)\cos(\gamma)}$$

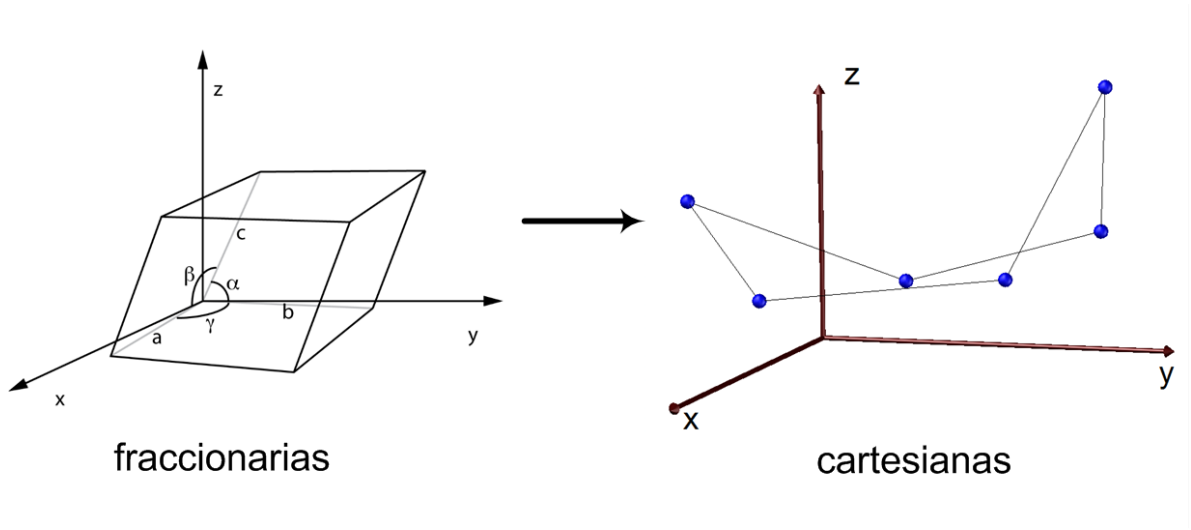


Figura 13: Cambio de coordenadas fraccionarias a coordenadas cartesianas

```
363:                                                        // Parámetros  de la
364: float cos_alfa=cos(p_celda[3]/180*pi);                 // matriz de cambio de
365: float cos_beta=cos(p_celda[4]/180*pi);                 //  coordenadas  M
366: float cos_gamma=cos(p_celda[5]/180*pi);                //
367:                                                        //
368:                                                        //
369: float sin_gamma=sin(p_celda[5]/180*pi);                //
370:                                                        //
371: float v= sqrt( 1 - pow(cos_alfa,2) - pow(cos_beta,2) -  //
372:       pow(cos_gamma,2) + 2*cos_alfa*cos_beta*cos_gamma); //
373:
374:
375: float M01=p_celda[1]*cos_gamma;                                    //
376: float M02=p_celda[2]*cos_beta;                                     //
377: float M11=p_celda[1]*sin_gamma;                                    // Matriz M
378: float M12=(p_celda[2]*(cos_alfa-cos_beta*cos_gamma))/sin_gamma;    //
379: float M22=p_celda[2]*v/sin_gamma;                                  //
380:
381:
382:
383: //Cambio de coordenadas: M x [x;y;z]
384: for (i=0;i<N;++i)
385:        {
386:        *(cartesian+i*3+0)= p_celda[0]*(*(fractional+i*3+0))+M01*(*(fractional+i*3+1)) +
387:                            M02*(*(fractional+i*3+2));
388:        *(cartesian+i*3+1)= M11*(*(fractional+i*3+1)) + M12*(*(fractional+i*3+2));
389:        *(cartesian+i*3+2)= M22*(*(fractional+i*3+2));
390:        }
```

3.3.2. función void centro_geometrico(float *cartesian)

Con esta función se realiza un cambio de coordenadas, desde un origen cualquiera (O) que se había tomado, hasta el centro geométrico de la molécula. Si todos los átomos tuvieran la misma masa, este punto sería también el centro de masas.

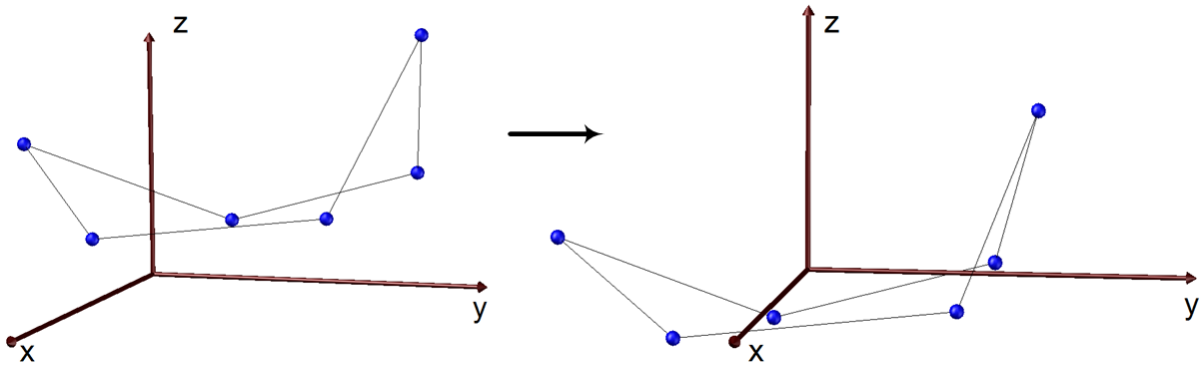


Figura 14: Cambio de origen al centro geométrico del anillo

Las coordenadas del centro geométrico (c.g.) son:

$$x_c = \frac{\sum_{i=1}^{N} x_i}{N} \qquad y_c = \frac{\sum_{i=1}^{N} y_i}{N} \qquad z_c = \frac{\sum_{i=1}^{N} z_i}{N}$$

Y para realizar el cambio de coordenadas:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}_{c.g} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}_{O} - \begin{bmatrix} x_c \\ y_c \\ z_c \end{bmatrix}$$

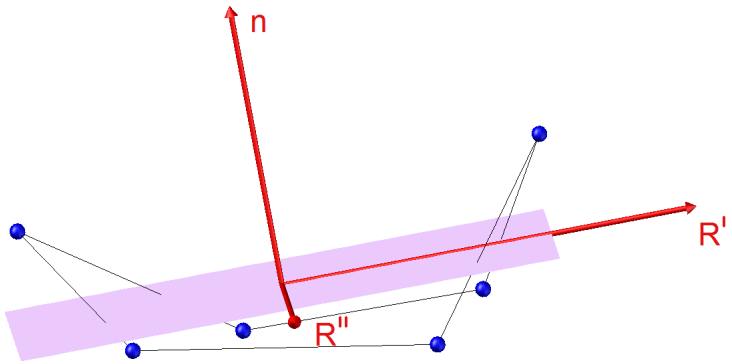
```
396: float x=0,y=0,z=0;
397: float xc, yc, zc;   //Coordenadas del centro de gravedad
398:
399: for(i=0;i<N;++i)     //Cálculo del c.d.g.
400:        {
401:        x=x+(*(cartesian+i*3+0));
402:        y=y+(*(cartesian+i*3+1));
403:        z=z+(*(cartesian+i*3+2));
404:        }
405: xc=x/N;
406: yc=y/N;
407: zc=z/N;
408:
409: for(i=0;i<N;++i)     //Cambio de origen del coordenadas al c.d.g.
410:        {
411:        (*(cartesian+i*3+0))=(*(cartesian+i*3+0))-xc;
412:        (*(cartesian+i*3+1))=(*(cartesian+i*3+1))-yc;
413:        (*(cartesian+i*3+2))=(*(cartesian+i*3+2))-zc;
414:        }
```

3.3.3. función void cartesian2intrinsic(float *cartesian,int k)

Esta función lleva a cabo el paso de coordenadas cartesianas a ”intrínsecas”.

Para primero calcula los vectores directores del plano medio y no perpendicular a estos dos, que ya se definieron en 2.1.1. Recordando:

$$\begin{aligned}\mathbf{R'} &= \sum_{j=1}^{N} \mathbf{R_j} \sin[\mathbf{2}\pi(\mathbf{j}-\mathbf{1})/\mathbf{N}] \\ \mathbf{R''} &= \sum_{j=1}^{N} \mathbf{R_j} \cos[\mathbf{2}\pi(\mathbf{j}-\mathbf{1})/\mathbf{N}] \\ \mathbf{n} &= \mathbf{R'} \times \mathbf{R''}/|\mathbf{R'} \times \mathbf{R''}|\end{aligned}$$



De los infinitos referenciales posibles, nosotros tomaremos uno con tres vectores ortonormales entre sí. De modo que cumplirán:

$$\begin{aligned}\mathbf{R_1} &= \frac{\mathbf{R'}}{|\mathbf{R'}|} \\ \mathbf{R_{2_0}} &= \mathbf{R''} - < \mathbf{R_1}, \mathbf{R'} > \mathbf{R_1} \\ \mathbf{R_2} &= \frac{\mathbf{R_{2_0}}}{|\mathbf{R_{2_0}}|} \\ \mathbf{n} &= \mathbf{R_1} \times \mathbf{R_2}\end{aligned}$$

Tenemos así tres vectores directores: $\mathbf{R_1}$, $\mathbf{R_2}$ y $\mathbf{n}$ que puestos en el centro geométrico (c.g.), constituyen un nuevo referencial, S'.

Para realizar el cambio de coordenadas, respecto del referencial inicial S:[O,$\hat{\imath}$,$\hat{\jmath}$,$\hat{k}$], al nuevo S':[c.g.,**R**',**R**",**n**], utilizamos la relación propuesta en [9]:

$$\mathbf{r'} = \mathbf{G} \cdot \mathbf{r}$$

r': Coordenadas de un vector respecto a S'
r: Coordenadas de un vector respecto a S

$$G^T = \begin{pmatrix} \hat{\imath} \cdot \mathbf{R'} & \hat{\imath} \cdot \mathbf{R''} & \hat{\imath} \cdot \mathbf{n} \\ \hat{\jmath} \cdot \mathbf{R'} & \hat{\jmath} \cdot \mathbf{R''} & \hat{\jmath} \cdot \mathbf{n} \\ \hat{\mathbf{k}} \cdot \mathbf{R'} & \hat{\mathbf{k}} \cdot \mathbf{R''} & \hat{\mathbf{k}} \cdot \mathbf{n} \end{pmatrix} = \begin{pmatrix} R'_x & R''_x & n_x \\ R'_y & R''_y & n_y \\ R'_z & R''_z & n_z \end{pmatrix}$$

De forma que nos queda la matriz de cambio de coordenadas como:

$$G = \begin{pmatrix} R'_x & R'_y & R'_z \\ R''_x & R''_y & R''_z \\ n_x & n_y & n_z \end{pmatrix}$$

En el programa, primero se calculan los vectores directores del nuevo referencial: $\mathbf{R_1}$, $\mathbf{R_2}$ y $\mathbf{n}$

```
420: float R1_0[3]={0,0,0},R2_0[3]={0,0,0},R2_1[3],R0[3],k1;  // Variables temporales
421: float n[3],R1[3],R2[3];     // Ejes del sistema de coordenadas
422:
423: for(i=0;i<N;++i)
424:        for(j=0;j<3;++j)
425:        {
426:               R1_0[j]=R1_0[j]+(*(cartesian+i*3+j))*sin(2*pi*i/N);
427:               R2_0[j]=R2_0[j]+(*(cartesian+i*3+j))*cos(2*pi*i/N);
428:        }
429:
430:
431: normalize(R1,R1_0);
432: dotProduct(k1,R1,R2_0);
433: scalar(R0,k1,R1);
434: subtraction(R2_1,R2_0,R0);
435: normalize(R2,R2_1);
436:
437: crossProduct(n,R1,R2)
```

Después se construye la matriz G:

```
440: float G[3][3]; //Matriz de cambio de coordenadas
441: for(i=0;i<3;++i)
442:        {
443:        G[0][i]=R1[i];
444:        G[1][i]=R2[i];
445:        G[2][i]=n[i];
446:        }
```

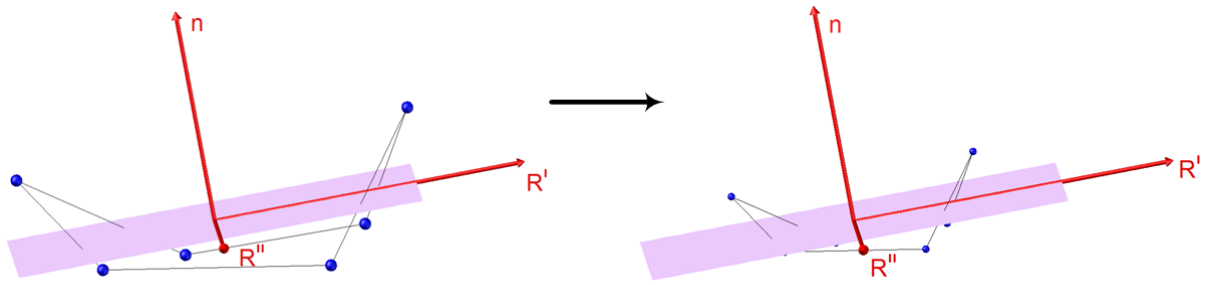
Y por último se realiza la multiplicación:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}_{intrinsecas} = G \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix}_{c.g.}$$

```
448: for(i=0;i<N;++i)
449:        for(j=0;j<3;++j)
450:               *(intrinsic+i*3+j)=G[j][0]*(*(cartesian+i*3+0))+G[j][1]*(*(cartesian+i*3+1))
451:                                  +G[j][2]*(*(cartesian+i*3+2));
```

3.3.4. función void escalar_molecula(float *Mf,int k)

Para que las moléculas sean comparables unas con otras, deben estar todas ”en la misma escala”. Matemáticamente esto se traduce en que todas tengan para sí mismas, una distancia interatómica igual a 1.



Para más información acerca de las coordenadas intrínsecas normalizadas, ver apartado 2.2.1.

A la función `escalar_molecula` se le pasa el puntero al primer elemento de las coordenadas cartesianas, `Mf`, y el número de la molécula `k`, y ésta calcula

$$I_F = M_F/\bar{d}.$$

```
458: float temp;  //Variable temporal
459: float dist=0;     //Distancia temporal entre dos atomos consecutivos
460: float dist_media; //Distancia media entre los N atomos
461:
462: for(i=0;i<(N-1);i++)
463:        {
464:        temp=0;
465:        for(j=0;j<3;j++)
466:               temp=temp+pow((*(in+i*3+j))-(*(in+(i+1)*3+j)),2);
467:        dist=dist+sqrt(temp);
468:        }
469:
470: temp=0;
471: for(j=0;j<3;j++)
472:        temp=temp+pow((*(in+0*3+j))-(*(in+(N-1)*3+j)),2);
473:
474:
475: dist=dist+sqrt(temp);
476: dist_media=dist/N;
477:
478: for(i=0;i<N;i++)
479:        for(j=0;j<3;j++)
480:               *(If+k*3*N+i*3+j)=(*(in+i*3+j))/dist_media;
```

Con esta función terminamos de crear la matriz tridimensional I_F, que es la base de todos los cálculos del algoritmo. Contiene las coordenadas de las moléculas que se pasarán a la función `escribir_archivo()`

3.4. Rutina escribir_archivo

Esta función se encarga de crear dos ficheros de texto donde se guardará la matriz de distancias de las moléculas a estudiar. El primer fichero sólo contendrá las distancias separadas por retorno de carro, y ordenadas como sigue:

$$
\begin{matrix}
d_{0,1} \\
d_{0,2} \\
\vdots \\
d_{0,n} \\
d_{1,2} \\
\vdots \\
d_{1,n} \\
d_{2,3} \\
\vdots \\
d_{n-1,n}
\end{matrix}
$$

En el segundo fichero, además de las distancias, se guarda:

- Número y nombre de las moléculas entre las que se calcula su distancia.
- Valores de los exponentes de la matrices para los que se consigue el mínimo: [s,v,a,b].
- Valor del ángulo de giro γ que proporciona mínima distancia.

He aquí un ejemplo del comienzo de este fichero:

```
0 1:ACAVIJ1 ACAVIJ2;    d: 0.02178, s=4, v=0, a=0, b=0, gamma=3.1327

0 2:ACAVIJ1 ACAVIJ3;    d: 0.02659, s=4, v=0, a=0, b=0, gamma=3.1304

0 3:ACAVIJ1 AHEKON1;    d: 0.01454, s=1, v=0, a=1, b=0, gamma=0.0072
```

El usuario debe introducir primeramente el nombre del archivo donde se guardará la matriz. El programa añade directamente la extensión .txt, y genera el archivo con la información de los mínimos añadiendo *_detalle.txt* al nombre elegido, por eso es conveniente (aunque no estrictamente necesario) que el nombre se escriba sin ninguna extensión.

```
493: FILE *salida; // Archivo en el que se escribirá únicamente la matriz de distancias
494:
495: char fichero_salida[228];
496: printf("Fichero de escritura de la matriz de distancias (sin extension): ");
497: scanf("%s",fichero_salida);
498: strcat(fichero_salida,".txt"); // Añade la extension por defecto .txt
```

Después se comprueba que no existe un fichero de salida con el mismo nombre. En caso de ser así, se ha de escoger otro.

```
502: while ((salida=fopen(fichero_salida,"r"))!=NULL){
503:    printf("\nEl archivo ya existe. Especifique otro: ");
504:    scanf("%s",fichero_salida);
505:    strcat(fichero_salida,".txt");
506:    }
507:
508: salida=fopen(fichero_salida,"w");
```

El programa genera automáticamente otro archivo, que se llama como el definido por el usuario, pero añadiendo _detalle.txt al final. Este archivo grabará, además de la matriz, los valores de $[s, a, v, b, gamma]$ de cada mínimo:

```
515: FILE *salida_detalle; // Archivo en el que se escribirán los nombres de las moleculas,
516:                       // su distancia mínima y [s,a,v,b,gamma]
517:
518: char fichero_salida_detalle[241];
519: strcpy(fichero_salida_detalle,fichero_salida);
520: strcpy(strstr(fichero_salida_detalle,".txt"),"_detalle.txt");
521:
522: salida_detalle=fopen(fichero_salida_detalle,"w");
```

A partir de aquí, la función lo único que hace es un bucle donde se "barren" las n moléculas, y se va calculando su distancia d por parejas a través de la función `distancia_d`.

```
524: int m1,m2; // Moleculas que se van a comparar
525: float dist; // d
526:
527: for(m1=0;m1<leidas;m1++){
528:        for(m2=m1+1;m2<leidas;m2++) // Barre todas las moleculas
529:               {
530:               dist=distancia_d(If+m1*N*3,If+m2*N*3); // Calcula la distancia entre m1 y m2
531:
532:               fprintf(salida,"%.5f\n",dist); // Escribe la distancia entre m1 y m2
533:
534:               /* Se escribe en el fichero de detalle la información complementaria */
535:               fprintf(salida_detalle,"%d %d:",m1,m2); // Subíndice dentro de la matriz
537:               t=0; // El nombre de la primera molécula
538:               while(*(nombres+m1*long_max_nombre+t) !=';'){
539:                      fprintf(salida_detalle,"%c",*(nombres+m1*long_max_nombre+t));
540:                      t++;}
541:
542:               fprintf(salida_detalle," ");  // Un espacio intermedio
543:
544:               t=0; // El nombre de la segunda molécula
545:               while(*(nombres+m2*long_max_nombre+t) !=';'){
546:                      fprintf(salida_detalle,"%c",*(nombres+m2*long_max_nombre+t));
547:                      t++;}
548:
549:               fprintf(salida_detalle,";\t"); // Un tabulador intermedio
550:
551:               /* Y por último, los valores de [s,a,v,b,gamma] para la distancia mínima */
552:               fprintf(salida_detalle,"d: %.5f, s=%d, v=%d, a=%d, b=%d, gamma=%.4f\n\n",
553:               dist,umin+1,vmin,amin,bmin,gamma_min);
554:
555:               }
556: printf(".");}  /* Hace que se vayan escribiendo puntos en pantalla mientras
557:                           corre el programa */
```

Se ha escogido una precisión para la distancia de 10^{-5}. Normalmente los datos cristalográficos de la bibliografía también tienen una precisión para las coordenadas fraccionarias de 10^{-5}. Disminuir la tolerancia del programa para llegar hasta nivel de precisión superior supone un coste en tiempo considerable para ficheros grandes.

3.4.1. función float distancia_d(float *If1, float *If2)

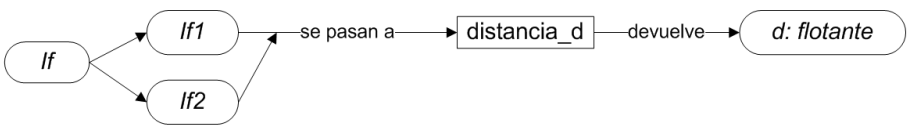


Figura 15: Esquema de la función distancia_d()

Para calcular la distancia entre dos moléculas, se pasan a esta función sus coordenadas intrínsecas a través dos punteros,`If1` e `If2`, que señalan a la coordenada x del primer átomo de la cada una molécula. La función calcula la distancia `d` entre las dos, y devuelve un flotante.

Recordemos que la definición de d es:

$$d(F_1, F_2) = \min_{\substack{s \in \mathcal{S} \\ v,a,b \in \{0,1\}}} \min_{\gamma \in [0, 2\pi[} \| I_{F_1} - (T^{(s-1)} D^v I_{F_2} M^a J^b)(R_z(\gamma))^T \|$$

Más en detalle, el esquema que sigue la función `distancia_d` es el siguiente:

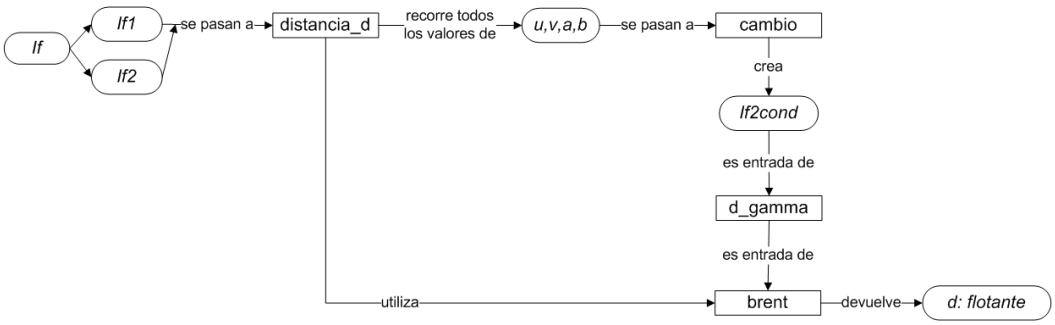


Figura 16: Esquema de la función distanciad()

Funciones auxiliares de `distancia_d`:

- cambio(int u, int v, int a, int b, float *If2, float *If2_cambiado)
- float d_gamma(float gamma)

Primero se ejecuta la función `cambio` tomando todos los valores posibles de [s,v,a,b]. En cada ejecución, se crea una matriz `If2_cond`, que se corresponde con $T^{(s-1)} D^v I_{F_2} M^a J^b$.

Se ha llamado “condición” a unos determinados valores de los exponentes [s,v,a,b].
Por lo tanto, `If2_cond` no es más que `If2`, multiplicada por las matrices de cambio con los exponentes de una determinada condición.

Para cada una de estas condiciones, el mínimo `d` se encuentra en un determinada valor del ángulo de giro γ. De esta forma, para dos moléculas dadas, y fijos [s,v,a,b], la distancia d sólo depende de una variable, γ.
A una función que calcule este mínimo que sólo cambia con γ se la ha llamado `d\_gamma`.

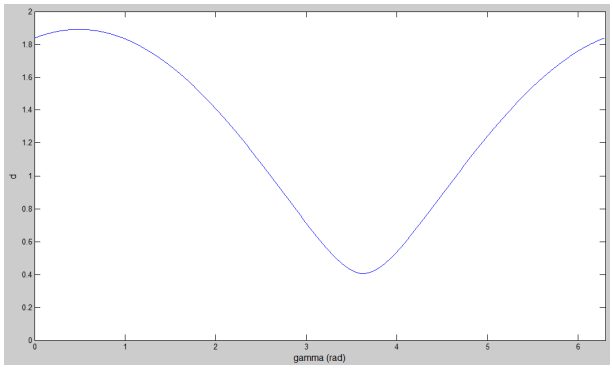


Figura 17: Distancia d entre las moléculas ACAVIJ1 y BAJNAB3, dados s=2,v=1,a=1,b=1

A la hora de hallar este mínimo, se ha recurrido a la librería `brent` [1]. Se trata de una rutina, que calcula el mínimo de una función de una sola variable, y devuelve un flotante (el propio mínimo) y el valor de la variable en que se consigue (en nuestro caso γ).

El bucle de `distancia_d` parte de un valor de `dmin` muy alto, 10^6, calcula el mínimo de `d_gamma` para una condición: `dcond` y si éste es menor que el valor anterior de `dmin`, lo establece como nuevo mínimo. El valor de la distancia se guarda en la variable global `dmin` y los valores de [s,v,a,b,γ] en los que se produce en las variables globales `umin,vmin,amin,bmin,gamma_min`.

Para cada valor de s, como v, a y b pueden valer 0 ó 1, encontramos 2^3=8 curvas $d(\gamma)$. La rutina `brent`, devuelve el mínimo de cada una de estas curvas.

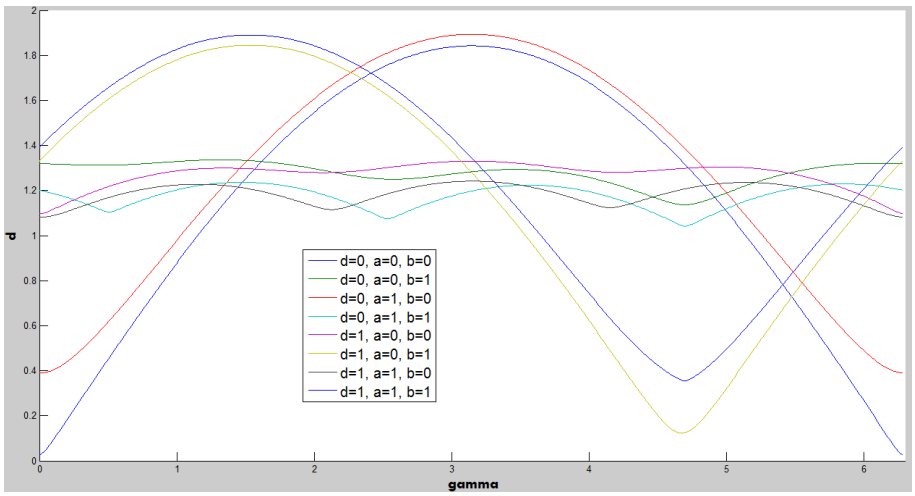


Figura 18: Curvas $d(\gamma)$

Habrá por lo tanto 8·S candidatos a mínimo (`dcond`), siendo S el número de valores que puede tomar s (que en el programa se ha llamado `size_s`).

Al principio definimos las variables que se van a usar en esta función:

```
563: float distancia_d(float *If1, float *If2)
564: {
565: int u,v,a,b;      // Variables contador. Nota: u = (s-1)
566:
567: p_If1=If1;        /* Asigno al puntero que se utilizara en d_gamma() el valor del If1
568:                      que paso a la funcion distancia_d() */
569:
570: dmin=10000; /* Punto inicial de iteración. La distancia va decreciendo,
571:                de modo que se empieza por un valor muy alto */
572:
573: float dcond;      // Minima distancia d(gamma) para esas condiciones: u,v,a,b
574:
575: float *p_min_temp;    // Puntero donde escribirá golden el mínimo
576: float min_temp;       // Mínimo de la función devuelto por golden
577: p_min_temp=&min_temp; // Asignación de p_min_temp a una direccion de memoria
578:
579:
580: float (*p_d_gamma)(float gamma);  // Puntero a la funcion d_gamma
581: p_d_gamma=d_gamma; // Asigna al puntero la dirección de la funcion
```

Después comienza el barrido de todas las posibles condiciones:

```
584: for(u=0; u<size_s; ++u)
585:         for(v=0; v<=1; ++v)
586:                 for(a=0; a<=1; ++a)
587:                         for(b=0; b<=1; ++b)
588:                                 {
589:                                 cambio((*(S+u)-1),v,a,b,If2,If2cond);
590:                                 // Escribe en la matriz If2_cond: T^(s-1)*D^v*If2*M^a*J^b
591:                                 dcond=brent(-0.0001,2.4,6.2833,p_d_gamma,0.000001,p_min_temp);
592:
593:                                 if(dcond<dmin)
594:                                         {
595:                                         dmin=dcond;
596:                                         umin=(*(S+u)-1);
597:                                         vmin=v;
598:                                         amin=a;
599:                                         bmin=b;
600:                                         gamma_min=min_temp;
601:                                         }
602:                                 }
603: return(dmin);
```

3.4.2. función void cambio(int u, int v, int a, int b, float *If2)

A esta función se le pasan los valores de [u=s-1,v,a,b] así como la dirección en memoria de la molécula I_{F_2} y calcula:

$$I_{F_{2_{cond}}} = T^{(s-1)} D^v I_{F_2} M^a J^b$$

Para ahorrar tiempo de cálculo, no se multiplican todas las matrices, ya que si un exponente es cero, se estaría multiplicando por una matriz identidad, lo cual no cambiaría nada y haría retardar al programa. En su lugar se evalúan todos los exponentes, y sólo se realizan las multiplicaciones estrictamente necesarias.

Se ha descompuesto el problema en varias partes.

Primero se define:

$$I_{F_{pre}} = D^v I_{F_2} M^a J^b$$

De forma que:

$$I_{F_{2_{cond}}} = T^{(s-1)} I_{f_{pre}} = T^u I_{f_{pre}}$$

Para calcular esta $I_{F_{pre}}$, se parte de la siguiente tabla:

v	a	b	$I_{f_{pre}}$
0	0	0	I_{F_2}
0	0	1	$I_{F_2}J$
0	1	0	$I_{F_2}M$
0	1	1	$I_{F_2}MJ$
1	0	0	DI_{F_2}
1	0	1	$DI_{F_2}J$
1	1	0	$DI_{F_2}M$
1	1	1	$DI_{F_2}MJ$

Como la función que se ha utilizado para multiplicar las matrices (`multYXZ(a1,a2, a3,Y, X,Z)`, con a1[Y][X]*a2[X][Z]=a3[Y][Z]) sólo puede multiplicar dos matrices cada vez, en los casos en que aparecen más de dos matrices, se utilizan unas variables temporales para guardar las matrices intermedias: `mult_temp1` y `mult_temp2`.

Para el caso particular v=0, a=0, b=0, tenemos $I_{F_{pre}} = I_{F_2}$. Cuando esto sucede el puntero de la matriz interna de cálculo `Ifpre`, queda apuntando a una dirección de memoria que contiene las coordenadas de I_{F_2}, por eso para evitar que en la siguiente ejecución de cambio se modifiquen los datos de entrada, hay que dirigir siempre `Ifpre` a su dirección de memoria inicial: `Ifpre_inicial`.

```
524: int m1,m2; // Moleculas que se van a comparar
525: float dist; // d
526:
618: // Cálculo de Ifpre = D^v * If2 * M^a * J^b
619: if (v==0)
620:         if (a==0)
621:                 if (b==0)
622:                         Ifpre=If2;
623:
624:                 else // b=1
625:                         multYXZ(If2,&J[0][0],Ifpre,N,3,3);
626:         else // a=1
627:                 if (b==0)
628:                         multYXZ(If2,&M[0][0],Ifpre,N,3,3);
629:
630:                 else{ // b=1
631:                         multYXZ(If2,&M[0][0],mult_temp1,N,3,3);
632:                         multYXZ(mult_temp1,&J[0][0],Ifpre,N,3,3);}
633: else // v=1
634:      if (a==0)
635:                 if (b==0)
636:                         multYXZ(D,If2,Ifpre,N,N,3);
637:
638:                 else{// b=1
639:                         multYXZ(D,If2,mult_temp1,N,N,3);
640:                         multYXZ(mult_temp1,&J[0][0],Ifpre,N,3,3);}
641:         else // a=1
642:                 if (b==0){
643:                         multYXZ(D,If2,mult_temp1,N,N,3);
644:                         multYXZ(mult_temp1,&M[0][0],Ifpre,N,3,3);}
645:
646:                 else{ // b=1
647:                         multYXZ(D,If2,mult_temp1,N,N,3);
648:                         multYXZ(mult_temp1,&M[0][0],mult_temp2,N,3,3);
649:                         multYXZ(mult_temp2,&J[0][0],Ifpre,N,3,3);}
```

Una vez calculada $I_{F_{pre}}$, se crea la matriz T^u y se calcula $I_{F_{2_{cond}}} = T^u I_{f_{pre}}$. Notar que D^v, M^a y J^b son matrices fijas, es decir, sólo tienen una forma para un exponente no nulo. Sin embargo, T^u puede tener distintas formas para un $u \neq 0$, por eso a diferencia de las otras no se ha dejado como variable global, sino que es una variable local que se crea en cada ejecución de `cambio`.

```
652: if (u==0)
653:         If2cond=Ifpre;
654: else{
655:         float Tcond[N][N];                  //  Definición de T^u
656:         for(i=0; i<N; ++i)                  //
657:                 for(j=0; j<N; ++j)          //
658:                         Tcond[i][j]=0;      //
659:                 for(j=0; j<N-u; ++j)        //
660:                         Tcond[j][j+u]=1;    //
661:         for(i=0; i<u; ++i)                  //
662:                 Tcond[N-u+i][i]=1;          //
663:
664:         multYXZ(&Tcond[0][0],Ifpre,If2_cambiado,N,N,3);
665:       }
```

3.4.3. función float d_gamma(float gamma)

Esta función devuelve la distancia que hay entre dos moléculas apuntadas por los punteros `p_If1` e `If2cond`, dependiendo del ángulo γ que gire el segundo respecto del primero, es decir:

$$\|p_{I_{F_1}} - I_{F_{2_{cond}}}(R_z(\gamma))^T\|$$

Para ello se utiliza la función `giro(float *m, float gamma, float *m_girada)` para calcular $I_{F_{2_{cond}}}(R_z(\gamma))^T$, que se define como `If2cond_gamma` y contiene las coordenadas del fragmento girado un ángulo γ a lo largo del eje z.

Después la función `norma(float *a,float *b)`, que devuelve la norma de la resta de las dos matrices: $\|a - b\|$.

Por último se devuelve como resultado de la función `d_gamma` el valor de $\|$`p_If1`$-$`If2cond_gamma`$\|$

```
675: giro(If2cond,gamma,If2cond_gamma);
676: float distancia=norma(p_If1,If2cond_gamma);
677: return(distancia);
```

3.4.4. función void giro(float *m, float gamma, float *m_girada)

Considerando, para $\gamma \in \mathbf{R}$, la matriz

$$R_z(\gamma) = \begin{pmatrix} \cos(\gamma) & \sin(\gamma) & 0 \\ -\sin(\gamma) & \cos(\gamma) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Esta función calcula la matriz de giro, y gira la matriz `m`, guardando el resultado en la matriz `m_girada`.

```
684: float matriz_giro[3][3]={
685:         cos(gamma),-sin(gamma),0,
686:         sin(gamma),cos(gamma), 0,
687:         0,         0,          1};
688:
689: multYXZ(m,&matriz_giro[0][0],m_girada,N,3,3);
```

3.4.5. función float norma(float *a,float *b)

A esta función se le pasan dos punteros, `a` y `b`, que apuntan a dos matrices N $\times$ 3 y delvuelve:

$$\|a-b\| = \frac{1}{N}\sum_{i=1}^{N}\sqrt{(a_{i,1}-b_{i,1})^2+(a_{i,2}-b_{i,2})^2+(a_{i,3}-b_{i,3})^2}$$

```
697: float suma2;     // Suma de (a[i][j]-b[i][j])^2 para un i fijo, de las tres componentes
698: float norma_i=0;   // Norma del átomo [i]
699:
700: for(i=0; i<N; ++i)
701:        {
702:        suma2=0;
703:        for(j=0; j<3; ++j)
704:               suma2=suma2+pow((*(a+i*3+j)-*(b+i*3+j)),2);
705:        norma_i=norma_i+pow(suma2,0.5);
706:        }
707: float norma_resta=norma_i/N;
708:
709: return(norma_resta);
```

4. Ejemplos de aplicación

4.1. Conjunto 8C1

Aplicaremos ahora nuestro programa para calcular la matriz de distancias de un conjunto de 31 anillos de ciclo octano sp^3. Se ha escogido este sistema porque los anillos de este tipo se han estudiado ampliamente de forma teórica en [10] y de forma experimental en [6].

Los anillos estudiados son, por orden alfabético:

Nombre	Referencia
AMCOCA0	0
BAGPII0	1
BCOCTB0	2
CLCOCT0	22
COCOAC0	23
COCOXA100	3
COVLUU0	4
CYOCDL0	24
CURBIA0	5
CUVZEY0	6
DEZPUT0	7
EOCNON100	26
ECOTDA0	25
GATRAU0	8
GIVBAO0	9
HOXTHD0	10
HUMULB100	27
KESVIN0	11
OCSHYD0	12
PCDODO0	13
SATKIH1	14
SATKIH2	15
SATKIH3	16
SATKIH4	17
SEJFIW1	28
SEJFIW0	29
SPTZBN0	30
SPOCTC100	18
VALGOE1	19
VALGOE2	20
VASWOB0	21

Los resultados obtenidos de las distancias entre parejas de anillos se muestran, de forma resumida, en la página siguiente.

Ejemplo de aplicación: Conjunto 8 C1

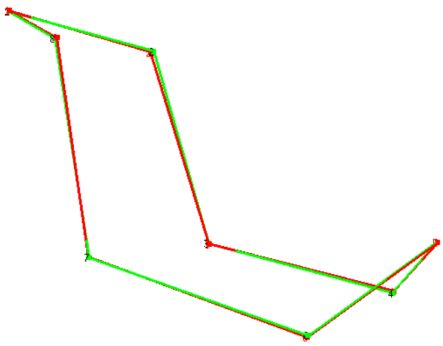
0 1:AMCOCA0 BAGPII0; 0.0309
0 2:AMCOCA0 BCOCTB0; 0.3512
0 3:AMCOCA0 COCOXA100; 0.0341
0 4:AMCOCA0 COVLUU0; 0.1978
0 5:AMCOCA0 CURBIA0; 0.3545
0 6:AMCOCA0 CUVZEY0; 0.0226
0 7:AMCOCA0 DEZPUT0; 0.2558
0 8:AMCOCA0 GATRAU0; 0.2280
0 9:AMCOCA0 GIVBAO0; 0.3584
0 10:AMCOCA0 HOXTHD0; 0.0672
0 11:AMCOCA0 KESVIN0; 0.0535
0 12:AMCOCA0 OCSHYD0; 0.0370
0 13:AMCOCA0 PCDODO0; 0.3012
0 14:AMCOCA0 SATKIH1; 0.2247
0 15:AMCOCA0 SATKIH2; 0.2282
0 16:AMCOCA0 SATKIH3; 0.2627
0 17:AMCOCA0 SATKIH4; 0.2206
0 18:AMCOCA0 SPOCTC100; 0.0380
0 19:AMCOCA0 VALGOE1; 0.3345
0 20:AMCOCA0 VALGOE2; 0.3299
0 21:AMCOCA0 VASWOB0; 0.3173
0 22:AMCOCA0 CLCOCT0; 0.0343
0 23:AMCOCA0 COCOAC0; 0.0281
0 24:AMCOCA0 CYOCDL0; 0.3648
0 25:AMCOCA0 ECOTDA0; 0.3023
0 26:AMCOCA0 EOCNON100; 0.2778
0 27:AMCOCA0 HUMULB100; 0.2675
0 28:AMCOCA0 SEJFIW1; 0.2908
0 29:AMCOCA0 SEJFIW0; 0.2895
0 30:AMCOCA0 SPTZBN0; 0.2046
1 2:BAGPII0 BCOCTB0; 0.3579
1 3:BAGPII0 COCOXA100; 0.2418
1 4:BAGPII0 COVLUU0; 0.0440
1 5:BAGPII0 CURBIA0; 0.3628
1 6:BAGPII0 CUVZEY0; 0.0255
1 7:BAGPII0 DEZPUT0; 0.2552
1 8:BAGPII0 GATRAU0; 0.2300
1 9:BAGPII0 GIVBAO0; 0.3599
1 10:BAGPII0 HOXTHD0; 0.0596
1 11:BAGPII0 KESVIN0; 0.0681
1 12:BAGPII0 OCSHYD0; 0.0319
1 13:BAGPII0 PCDODO0; 0.3001
1 14:BAGPII0 SATKIH1; 0.1325
1 15:BAGPII0 SATKIH2; 0.1437
1 16:BAGPII0 SATKIH3; 0.2709
1 17:BAGPII0 SATKIH4; 0.2355
1 18:BAGPII0 SPOCTC100; 0.0267
1 19:BAGPII0 VALGOE1; 0.3403
1 20:BAGPII0 VALGOE2; 0.3356
1 21:BAGPII0 VASWOB0; 0.3224
1 22:BAGPII0 CLCOCT0; 0.0217
1 23:BAGPII0 COCOAC0; 0.0215
1 24:BAGPII0 CYOCDL0; 0.3676
1 25:BAGPII0 ECOTDA0; 0.2950
1 26:BAGPII0 EOCNON100; 0.2780
1 27:BAGPII0 HUMULB100; 0.2658
1 28:BAGPII0 SEJFIW1; 0.3035
1 29:BAGPII0 SEJFIW0; 0.2966
1 30:BAGPII0 SPTZBN0; 0.2070
2 3:BCOCTB0 COCOXA100; 0.3490
2 4:BCOCTB0 COVLUU0; 0.3814
2 5:BCOCTB0 CURBIA0; 0.0580
2 6:BCOCTB0 CUVZEY0; 0.3541
2 7:BCOCTB0 DEZPUT0; 0.2435
2 8:BCOCTB0 GATRAU0; 0.3596
2 9:BCOCTB0 GIVBAO0; 0.2342
2 10:BCOCTB0 HOXTHD0; 0.3358
2 11:BCOCTB0 KESVIN0; 0.3171
2 12:BCOCTB0 OCSHYD0; 0.3595
2 13:BCOCTB0 PCDODO0; 0.1969
2 14:BCOCTB0 SATKIH1; 0.3466
2 15:BCOCTB0 SATKIH2; 0.3247
2 16:BCOCTB0 SATKIH3; 0.2869
2 17:BCOCTB0 SATKIH4; 0.3261
2 18:BCOCTB0 SPOCTC100; 0.3594
2 19:BCOCTB0 VALGOE1; 0.0379
2 20:BCOCTB0 VALGOE2; 0.0421
2 21:BCOCTB0 VASWOB0; 0.0528
2 22:BCOCTB0 CLCOCT0; 0.3586
2 23:BCOCTB0 COCOAC0; 0.3479
2 24:BCOCTB0 CYOCDL0; 0.2867
2 25:BCOCTB0 ECOTDA0; 0.1997
2 26:BCOCTB0 EOCNON100; 0.3872
2 27:BCOCTB0 HUMULB100; 0.1363
2 28:BCOCTB0 SEJFIW1; 0.2616
2 29:BCOCTB0 SEJFIW0; 0.2663
2 30:BCOCTB0 SPTZBN0; 0.3000
3 4:COCOXA100 COVLUU0; 0.0550
3 5:COCOXA100 CURBIA0; 0.3544
3 6:COCOXA100 CUVZEY0; 0.2349
3 7:COCOXA100 DEZPUT0; 0.2469
3 8:COCOXA100 GATRAU0; 0.0213
3 9:COCOXA100 GIVBAO0; 0.3538
3 10:COCOXA100 HOXTHD0; 0.2170
3 11:COCOXA100 KESVIN0; 0.2660
3 12:COCOXA100 OCSHYD0; 0.2192
3 13:COCOXA100 PCDODO0; 0.2949
3 14:COCOXA100 SATKIH1; 0.1206
3 15:COCOXA100 SATKIH2; 0.1338
3 16:COCOXA100 SATKIH3; 0.2625
3 17:COCOXA100 SATKIH4; 0.1423
3 18:COCOXA100 SPOCTC100; 0.2243
3 19:COCOXA100 VALGOE1; 0.3550
3 20:COCOXA100 VALGOE2; 0.3502
3 21:COCOXA100 VASWOB0; 0.3379
3 22:COCOXA100 CLCOCT0; 0.2429
3 23:COCOXA100 COCOAC0; 0.2294
3 24:COCOXA100 CYOCDL0; 0.3583
3 25:COCOXA100 ECOTDA0; 0.3056
3 26:COCOXA100 EOCNON100; 0.2639
3 27:COCOXA100 HUMULB100; 0.2596
3 28:COCOXA100 SEJFIW1; 0.2831
3 29:COCOXA100 SEJFIW0; 0.2883
3 30:COCOXA100 SPTZBN0; 0.0841
4 5:COVLUU0 CURBIA0; 0.3856
4 6:COVLUU0 CUVZEY0; 0.2639
4 7:COVLUU0 DEZPUT0; 0.2582
4 8:COVLUU0 GATRAU0; 0.0592
4 9:COVLUU0 GIVBAO0; 0.3459
4 10:COVLUU0 HOXTHD0; 0.0992
4 11:COVLUU0 KESVIN0; 0.2886
4 12:COVLUU0 OCSHYD0; 0.0484
4 13:COVLUU0 PCDODO0; 0.2898
4 14:COVLUU0 SATKIH1; 0.1351
4 15:COVLUU0 SATKIH2; 0.1462
4 16:COVLUU0 SATKIH3; 0.2757
4 17:COVLUU0 SATKIH4; 0.1525
4 18:COVLUU0 SPOCTC100; 0.2519
4 19:COVLUU0 VALGOE1; 0.3112
4 20:COVLUU0 VALGOE2; 0.3063
4 21:COVLUU0 VASWOB0; 0.3628
4 22:COVLUU0 CLCOCT0; 0.2706
4 23:COVLUU0 COCOAC0; 0.0363
4 24:COVLUU0 CYOCDL0; 0.3381
4 25:COVLUU0 ECOTDA0; 0.3239
4 26:COVLUU0 EOCNON100; 0.2851
4 27:COVLUU0 HUMULB100; 0.2435
4 28:COVLUU0 SEJFIW1; 0.2693
4 29:COVLUU0 SEJFIW0; 0.2743
4 30:COVLUU0 SPTZBN0; 0.1265
5 6:CURBIA0 CUVZEY0; 0.3597
5 7:CURBIA0 DEZPUT0; 0.2464
5 8:CURBIA0 GATRAU0; 0.3676
5 9:CURBIA0 GIVBAO0; 0.2365
5 10:CURBIA0 HOXTHD0; 0.3484
5 11:CURBIA0 KESVIN0; 0.3176
5 12:CURBIA0 OCSHYD0; 0.3602
5 13:CURBIA0 PCDODO0; 0.2058
5 14:CURBIA0 SATKIH1; 0.3610
5 15:CURBIA0 SATKIH2; 0.3384
5 16:CURBIA0 SATKIH3; 0.3030
5 17:CURBIA0 SATKIH4; 0.3274
5 18:CURBIA0 SPOCTC100; 0.3677
5 19:CURBIA0 VALGOE1; 0.0552
5 20:CURBIA0 VALGOE2; 0.0580
5 21:CURBIA0 VASWOB0; 0.0679
5 22:CURBIA0 CLCOCT0; 0.3653
5 23:CURBIA0 COCOAC0; 0.3490
5 24:CURBIA0 CYOCDL0; 0.2830
5 25:CURBIA0 ECOTDA0; 0.2199
5 26:CURBIA0 EOCNON100; 0.4022
5 27:CURBIA0 HUMULB100; 0.1505
5 28:CURBIA0 SEJFIW1; 0.2637
5 29:CURBIA0 SEJFIW0; 0.2678
5 30:CURBIA0 SPTZBN0; 0.3143
6 7:CUVZEY0 DEZPUT0; 0.2561
6 8:CUVZEY0 GATRAU0; 0.0231
6 9:CUVZEY0 GIVBAO0; 0.3522
6 10:CUVZEY0 HOXTHD0; 0.0552
6 11:CUVZEY0 KESVIN0; 0.0616
6 12:CUVZEY0 OCSHYD0; 0.0432
6 13:CUVZEY0 PCDODO0; 0.2908
6 14:CUVZEY0 SATKIH1; 0.2302
6 15:CUVZEY0 SATKIH2; 0.2339
6 16:CUVZEY0 SATKIH3; 0.2642
6 17:CUVZEY0 SATKIH4; 0.1501
6 18:CUVZEY0 SPOCTC100; 0.0283
6 19:CUVZEY0 VALGOE1; 0.3373
6 20:CUVZEY0 VALGOE2; 0.3326
6 21:CUVZEY0 VASWOB0; 0.3194
6 22:CUVZEY0 CLCOCT0; 0.0336
6 23:CUVZEY0 COCOAC0; 0.0296
6 24:CUVZEY0 CYOCDL0; 0.3603
6 25:CUVZEY0 ECOTDA0; 0.2924
6 26:CUVZEY0 EOCNON100; 0.2796
6 27:CUVZEY0 HUMULB100; 0.2563
6 28:CUVZEY0 SEJFIW1; 0.2872
6 29:CUVZEY0 SEJFIW0; 0.2977
6 30:CUVZEY0 SPTZBN0; 0.1963
7 8:DEZPUT0 GATRAU0; 0.2465
7 9:DEZPUT0 GIVBAO0; 0.2990
7 10:DEZPUT0 HOXTHD0; 0.2455
7 11:DEZPUT0 KESVIN0; 0.2527
7 12:DEZPUT0 OCSHYD0; 0.2446
7 13:DEZPUT0 PCDODO0; 0.2755
7 14:DEZPUT0 SATKIH1; 0.1901
7 15:DEZPUT0 SATKIH2; 0.1614
7 16:DEZPUT0 SATKIH3; 0.1118
7 17:DEZPUT0 SATKIH4; 0.1864
7 18:DEZPUT0 SPOCTC100; 0.2613
7 19:DEZPUT0 VALGOE1; 0.2310
7 20:DEZPUT0 VALGOE2; 0.2312
7 21:DEZPUT0 VASWOB0; 0.2279
7 22:DEZPUT0 CLCOCT0; 0.2743
7 23:DEZPUT0 COCOAC0; 0.2467
7 24:DEZPUT0 CYOCDL0; 0.2462
7 25:DEZPUT0 ECOTDA0; 0.2967
7 26:DEZPUT0 EOCNON100; 0.2447
7 27:DEZPUT0 HUMULB100; 0.2026
7 28:DEZPUT0 SEJFIW1; 0.2304
7 29:DEZPUT0 SEJFIW0; 0.2300
7 30:DEZPUT0 SPTZBN0; 0.2318
8 9:GATRAU0 GIVBAO0; 0.3473
8 10:GATRAU0 HOXTHD0; 0.2041
8 11:GATRAU0 KESVIN0; 0.2005
8 12:GATRAU0 OCSHYD0; 0.2278
8 13:GATRAU0 PCDODO0; 0.3422
8 14:GATRAU0 SATKIH1; 0.1299
8 15:GATRAU0 SATKIH2; 0.1388
8 16:GATRAU0 SATKIH3; 0.2655
8 17:GATRAU0 SATKIH4; 0.1516
8 18:GATRAU0 SPOCTC100; 0.0161
8 19:GATRAU0 VALGOE1; 0.3446
8 20:GATRAU0 VALGOE2; 0.3397
8 21:GATRAU0 VASWOB0; 0.3256
8 22:GATRAU0 CLCOCT0; 0.0364
8 23:GATRAU0 COCOAC0; 0.2354
8 24:GATRAU0 CYOCDL0; 0.3606
8 25:GATRAU0 ECOTDA0; 0.3657
8 26:GATRAU0 EOCNON100; 0.2763
8 27:GATRAU0 HUMULB100; 0.2657
8 28:GATRAU0 SEJFIW1; 0.2876
8 29:GATRAU0 SEJFIW0; 0.2897
8 30:GATRAU0 SPTZBN0; 0.0781

9 10:GIVBAO0 HOXTHD0; 0.3197
9 11:GIVBAO0 KESVIN0; 0.3255
9 12:GIVBAO0 OCSHYD0; 0.3596
9 13:GIVBAO0 PCDODO0; 0.0740
9 14:GIVBAO0 SATKIH1; 0.2845
9 15:GIVBAO0 SATKIH2; 0.2911
9 16:GIVBAO0 SATKIH3; 0.2704
9 17:GIVBAO0 SATKIH4; 0.2665
9 18:GIVBAO0 SPOCTC100; 0.3462
9 19:GIVBAO0 VALGOE1; 0.2106
9 20:GIVBAO0 VALGOE2; 0.2067
9 21:GIVBAO0 VASWOB0; 0.1998
9 22:GIVBAO0 CLCOCT0; 0.3629
9 23:GIVBAO0 COCOAC0; 0.3563
9 24:GIVBAO0 CYOCDL0; 0.2272
9 25:GIVBAO0 ECOTDA0; 0.0930
9 26:GIVBAO0 EOCNON100; 0.2766
9 27:GIVBAO0 HUMULB100; 0.1920
9 28:GIVBAO0 SEJFIW1; 0.1258
9 29:GIVBAO0 SEJFIW0; 0.1227
9 30:GIVBAO0 SPTZBN0; 0.2858
10 11:HOXTHD0 KESVIN0; 0.1067
10 12:HOXTHD0 OCSHYD0; 0.0761
10 13:HOXTHD0 PCDODO0; 0.2602
10 14:HOXTHD0 SATKIH1; 0.1405
10 15:HOXTHD0 SATKIH2; 0.1474
10 16:HOXTHD0 SATKIH3; 0.2529
10 17:HOXTHD0 SATKIH4; 0.2268
10 18:HOXTHD0 SPOCTC100; 0.0419
10 19:HOXTHD0 VALGOE1; 0.3485
10 20:HOXTHD0 VALGOE2; 0.3460
10 21:HOXTHD0 VASWOB0; 0.3415
10 22:HOXTHD0 CLCOCT0; 0.0725
10 23:HOXTHD0 COCOAC0; 0.0694
10 24:HOXTHD0 CYOCDL0; 0.3480
10 25:HOXTHD0 ECOTDA0; 0.2603
10 26:HOXTHD0 EOCNON100; 0.2688
10 27:HOXTHD0 HUMULB100; 0.2263
10 28:HOXTHD0 SEJFIW1; 0.2855
10 29:HOXTHD0 SEJFIW0; 0.2834
10 30:HOXTHD0 SPTZBN0; 0.1879
11 12:KESVIN0 OCSHYD0; 0.0632
11 13:KESVIN0 PCDODO0; 0.2666
11 14:KESVIN0 SATKIH1; 0.2312
11 15:KESVIN0 SATKIH2; 0.2297
11 16:KESVIN0 SATKIH3; 0.2594
11 17:KESVIN0 SATKIH4; 0.2222
11 18:KESVIN0 SPOCTC100; 0.0796
11 19:KESVIN0 VALGOE1; 0.2965
11 20:KESVIN0 VALGOE2; 0.2913
11 21:KESVIN0 VASWOB0; 0.2786
11 22:KESVIN0 CLCOCT0; 0.0618
11 23:KESVIN0 COCOAC0; 0.0517
11 24:KESVIN0 CYOCDL0; 0.3244
11 25:KESVIN0 ECOTDA0; 0.2651
11 26:KESVIN0 EOCNON100; 0.2850
11 27:KESVIN0 HUMULB100; 0.2930
11 28:KESVIN0 SEJFIW1; 0.2609
11 29:KESVIN0 SEJFIW0; 0.2589
11 30:KESVIN0 SPTZBN0; 0.1309
12 13:OCSHYD0 PCDODO0; 0.2985
12 14:OCSHYD0 SATKIH1; 0.1324
12 15:OCSHYD0 SATKIH2; 0.1410
12 16:OCSHYD0 SATKIH3; 0.2677
12 17:OCSHYD0 SATKIH4; 0.2176
12 18:OCSHYD0 SPOCTC100; 0.0461
12 19:OCSHYD0 VALGOE1; 0.3402
12 20:OCSHYD0 VALGOE2; 0.3354
12 21:OCSHYD0 VASWOB0; 0.3221
12 22:OCSHYD0 CLCOCT0; 0.0341
12 23:OCSHYD0 COCOAC0; 0.0239
12 24:OCSHYD0 CYOCDL0; 0.3669
12 25:OCSHYD0 ECOTDA0; 0.2948
12 26:OCSHYD0 EOCNON100; 0.2861
12 27:OCSHYD0 HUMULB100; 0.2642
12 28:OCSHYD0 SEJFIW1; 0.2960
12 29:OCSHYD0 SEJFIW0; 0.2895
12 30:OCSHYD0 SPTZBN0; 0.2136
13 14:PCDODO0 SATKIH1; 0.2357
13 15:PCDODO0 SATKIH2; 0.2481
13 16:PCDODO0 SATKIH3; 0.2439
13 17:PCDODO0 SATKIH4; 0.2533
13 18:PCDODO0 SPOCTC100; 0.2862
13 19:PCDODO0 VALGOE1; 0.1768
13 20:PCDODO0 VALGOE2; 0.1771
13 21:PCDODO0 VASWOB0; 0.1738
13 22:PCDODO0 CLCOCT0; 0.3163
13 23:PCDODO0 COCOAC0; 0.3004
13 24:PCDODO0 CYOCDL0; 0.1933
13 25:PCDODO0 ECOTDA0; 0.0537
13 26:PCDODO0 EOCNON100; 0.2734
13 27:PCDODO0 HUMULB100; 0.2314
13 28:PCDODO0 SEJFIW1; 0.1073
13 29:PCDODO0 SEJFIW0; 0.0996
13 30:PCDODO0 SPTZBN0; 0.2957
14 15:SATKIH1 SATKIH2; 0.0380
14 16:SATKIH1 SATKIH3; 0.1626
14 17:SATKIH1 SATKIH4; 0.0279
14 18:SATKIH1 SPOCTC100; 0.2350
14 19:SATKIH1 VALGOE1; 0.3012
14 20:SATKIH1 VALGOE2; 0.2990
14 21:SATKIH1 VASWOB0; 0.3154
14 22:SATKIH1 CLCOCT0; 0.2380
14 23:SATKIH1 COCOAC0; 0.1305
14 24:SATKIH1 CYOCDL0; 0.3201
14 25:SATKIH1 ECOTDA0; 0.2900
14 26:SATKIH1 EOCNON100; 0.1741
14 27:SATKIH1 HUMULB100; 0.2711
14 28:SATKIH1 SEJFIW1; 0.2124
14 29:SATKIH1 SEJFIW0; 0.2176
14 30:SATKIH1 SPTZBN0; 0.1463
15 16:SATKIH2 SATKIH3; 0.1484
15 17:SATKIH2 SATKIH4; 0.0437
15 18:SATKIH2 SPOCTC100; 0.2400
15 19:SATKIH2 VALGOE1; 0.3097
15 20:SATKIH2 VALGOE2; 0.3060
15 21:SATKIH2 VASWOB0; 0.2950
15 22:SATKIH2 CLCOCT0; 0.2437
15 23:SATKIH2 COCOAC0; 0.1401
15 24:SATKIH2 CYOCDL0; 0.3273
15 25:SATKIH2 ECOTDA0; 0.2928
15 26:SATKIH2 EOCNON100; 0.1763
15 27:SATKIH2 HUMULB100; 0.2825
15 28:SATKIH2 SEJFIW1; 0.2096
15 29:SATKIH2 SEJFIW0; 0.2152
15 30:SATKIH2 SPTZBN0; 0.1460
16 17:SATKIH3 SATKIH4; 0.1493
16 18:SATKIH3 SPOCTC100; 0.2697
16 19:SATKIH3 VALGOE1; 0.2762
16 20:SATKIH3 VALGOE2; 0.2782
16 21:SATKIH3 VASWOB0; 0.2738
16 22:SATKIH3 CLCOCT0; 0.2783
16 23:SATKIH3 COCOAC0; 0.2689
16 24:SATKIH3 CYOCDL0; 0.3197
16 25:SATKIH3 ECOTDA0; 0.2534
16 26:SATKIH3 EOCNON100; 0.1610
16 27:SATKIH3 HUMULB100; 0.2586
16 28:SATKIH3 SEJFIW1; 0.2265
16 29:SATKIH3 SEJFIW0; 0.2271
16 30:SATKIH3 SPTZBN0; 0.2413
17 18:SATKIH4 SPOCTC100; 0.1608
17 19:SATKIH4 VALGOE1; 0.3286
17 20:SATKIH4 VALGOE2; 0.3245
17 21:SATKIH4 VASWOB0; 0.2875
17 22:SATKIH4 CLCOCT0; 0.1588
17 23:SATKIH4 COCOAC0; 0.2305
17 24:SATKIH4 CYOCDL0; 0.3128
17 25:SATKIH4 ECOTDA0; 0.2762
17 26:SATKIH4 EOCNON100; 0.1613
17 27:SATKIH4 HUMULB100; 0.2684
17 28:SATKIH4 SEJFIW1; 0.2025
17 29:SATKIH4 SEJFIW0; 0.2074
17 30:SATKIH4 SPTZBN0; 0.1601
18 19:SPOCTC100 VALGOE1; 0.3503
18 20:SPOCTC100 VALGOE2; 0.3476
18 21:SPOCTC100 VASWOB0; 0.3384
18 22:SPOCTC100 CLCOCT0; 0.0366
18 23:SPOCTC100 COCOAC0; 0.0368
18 24:SPOCTC100 CYOCDL0; 0.3629
18 25:SPOCTC100 ECOTDA0; 0.2839
18 26:SPOCTC100 EOCNON100; 0.2803
18 27:SPOCTC100 HUMULB100; 0.2519
18 28:SPOCTC100 SEJFIW1; 0.3002
18 29:SPOCTC100 SEJFIW0; 0.2982
18 30:SPOCTC100 SPTZBN0; 0.1953
19 20:VALGOE1 VALGOE2; 0.0082
19 21:VALGOE1 VASWOB0; 0.0219
19 22:VALGOE1 CLCOCT0; 0.3419
19 23:VALGOE1 COCOAC0; 0.3291
19 24:VALGOE1 CYOCDL0; 0.0342
19 25:VALGOE1 ECOTDA0; 0.1922
19 26:VALGOE1 EOCNON100; 0.3722
19 27:VALGOE1 HUMULB100; 0.3069
19 28:VALGOE1 SEJFIW1; 0.2464
19 29:VALGOE1 SEJFIW0; 0.2447
19 30:VALGOE1 SPTZBN0; 0.2862
20 21:VALGOE2 VASWOB0; 0.0180
20 22:VALGOE2 CLCOCT0; 0.3372
20 23:VALGOE2 COCOAC0; 0.3242
20 24:VALGOE2 CYOCDL0; 0.0394
20 25:VALGOE2 ECOTDA0; 0.1948
20 26:VALGOE2 EOCNON100; 0.3721
20 27:VALGOE2 HUMULB100; 0.3056
20 28:VALGOE2 SEJFIW1; 0.2431
20 29:VALGOE2 SEJFIW0; 0.2414
20 30:VALGOE2 SPTZBN0; 0.2812
21 22:VASWOB0 CLCOCT0; 0.3241
21 23:VASWOB0 COCOAC0; 0.3108
21 24:VASWOB0 CYOCDL0; 0.2877
21 25:VASWOB0 ECOTDA0; 0.1912
21 26:VASWOB0 EOCNON100; 0.3686
21 27:VASWOB0 HUMULB100; 0.0894
21 28:VASWOB0 SEJFIW1; 0.2376
21 29:VASWOB0 SEJFIW0; 0.2362
21 30:VASWOB0 SPTZBN0; 0.2683
22 23:CLCOCT0 COCOAC0; 0.0298
22 24:CLCOCT0 CYOCDL0; 0.3850
22 25:CLCOCT0 ECOTDA0; 0.3128
22 26:CLCOCT0 EOCNON100; 0.2791
22 27:CLCOCT0 HUMULB100; 0.2607
22 28:CLCOCT0 SEJFIW1; 0.2942
22 29:CLCOCT0 SEJFIW0; 0.3049
22 30:CLCOCT0 SPTZBN0; 0.2047
23 24:COCOAC0 CYOCDL0; 0.3556
23 25:COCOAC0 ECOTDA0; 0.2997
23 26:COCOAC0 EOCNON100; 0.2832
23 27:COCOAC0 HUMULB100; 0.2677
23 28:COCOAC0 SEJFIW1; 0.2912
23 29:COCOAC0 SEJFIW0; 0.2842
23 30:COCOAC0 SPTZBN0; 0.2125
24 25:CYOCDL0 ECOTDA0; 0.2983
24 26:CYOCDL0 EOCNON100; 0.3876
24 27:CYOCDL0 HUMULB100; 0.1347
24 28:CYOCDL0 SEJFIW1; 0.2538
24 29:CYOCDL0 SEJFIW0; 0.2516
24 30:CYOCDL0 SPTZBN0; 0.3588
25 26:ECOTDA0 EOCNON100; 0.2611
25 27:ECOTDA0 HUMULB100; 0.2237
25 28:ECOTDA0 SEJFIW1; 0.1366
25 29:ECOTDA0 SEJFIW0; 0.1350
25 30:ECOTDA0 SPTZBN0; 0.3230
26 27:EOCNON100 HUMULB100; 0.3436
26 28:EOCNON100 SEJFIW1; 0.3079
26 29:EOCNON100 SEJFIW0; 0.3100
26 30:EOCNON100 SPTZBN0; 0.2659
27 28:HUMULB100 SEJFIW1; 0.2127
27 29:HUMULB100 SEJFIW0; 0.2113
27 30:HUMULB100 SPTZBN0; 0.1890
28 29:SEJFIW1 SEJFIW0; 0.0089
28 30:SEJFIW1 SPTZBN0; 0.2527
29 30:SEJFIW0 SPTZBN0; 0.2554

A partir de aquí se pueden aplicar los métodos estadísticos de análisis conformacional.

Aunque estas herramientas se escapan del objetivo del proyecto, resumiremos las conclusiones obtenidas en [12]. Los datos de la matriz se importaron al programa estadístico R para su agrupamiento, escalado mutidimensional...

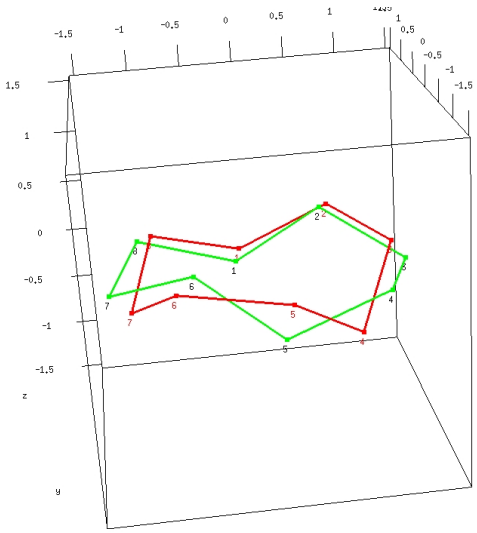
La distancia mínima entre fragmentos es:
`19 20:VALGOE1 VALGOE2;  0.00822.`



Teniendo en cuenta que el anillo normalizado tiene una distancia interatómica igual a 1, esto significa que los dos fragmentos difieren en un 0.822 %, es decir que son visualmente indistinguibles. De hecho se ve en la figura como las líneas roja y amarilla parecen coincidentes.

Por otra parte, la distancia máxima entre fragmentos es:
`26 28:EOCNON100 SEJFIW1;  0.30787.`

Esto quiere decir que la distancia media entre átomos uno a uno de los dos fragmentos es un 31 % de la distancia media interatómica, lo que acarrea una disposición en el espacio significativamente diferente, como se ve en la figura siguiente.



La media de las distancias entre pares de anillos del conjunto es 0.111 y el tercer cuartil es 0.145, lo que nos indica que la mayoría de conformaciones son más o menos iguales.

4.1.1. Agrupamiento agregado

Utilizando el criterio del vecino más lejano, se estima oportuno formar 8 grupos dentro del conjunto.

Nombre	**Método de clasificación, Ref. [13]**, $\sigma = 20$	**Método de Bayes completo Ref. [13]**
	Grupo 1	
AMCOCA	1.00 BC	$\mu(1)$
BAGPII	1.00 BC	$\mu(1)$
BCOCTB	1.00 BC	$\mu(1)$
COCOXA10	1.00 BC	$\mu(1)$
COVLUU	0.99 BC; 0.01 TBC	$\mu(2)$
CURBIA	1.00 BC	$\mu(1)$
CUVZEY	1.00 BC	$\mu(1)$
GATRAU	1.00 BC	$\mu(1)$
GIVBAO	1.00 BC	$\mu(1)$
OCSHYD	1.00 BC	$\mu(1)$
SPOCTC10	1.00 BC	$\mu(1)$
CLCOCT	1.00 BC	$\mu(1)$
COCOAC	1.00 BC	$\mu(1)$
CYOCDL	1.00 BC	$\mu(1)$
	Grupo 2	
HOXTHD	0.98 BC; 0.02 TBC	$\mu(2)$
KESVIN	0.98 BC; 0.02 TBC	$\mu(2)$
PCDODO	0.78 BC; 0.22 TBC	$\mu(3)$
VALGOE (1)	0.99 BC; 0.01 TBC	$\mu(1)$
VALGOE (2)	0.99 BC; 0.01 TBC	$\mu(1)$
VASWOB	0.97 BC; 0.03 TBC	$\mu(2)$
	Grupo 3	
SATKIH (1)	0.47 CC; 0.52 TCC	$\mu(6)$
SATKIH (2)	0.35 CC; 0.65 TCC	$\mu(6)$
SATKIH (3)	0.01 CR; 0.65 CC; 0.34 TCC	$\mu(6)$
	Grupo 4	
DEZPUT	0.67 CR; 0.15 CC; 0.19 TCC	$\mu(4)$
SATKIH (4)	0.55 CR; 0.18 CC; 0.27 TCC	$\mu(4)$
	Grupo 5	
ECOTDA	0.51 BC; 0.49 TBC	$\mu(3)$
SPTZBN	0.49 BC; 0.51 TBC	$\mu(3)$
	Grupo 6	
SEJFIW1	0.38 BC; 0.33 TBC; 0.29 TC	$\mu(7)$
SEJFIW	0.35 BC; 0.35 TBC; 0.30 TC	$\mu(7)$
	Grupo 7	
EOCNON10	0.06 CC; 0.94 TCC	$\mu(5)$
	Grupo 8	
HUMULB10	0.27 BC; 0.73 TBC	$\mu(3)$

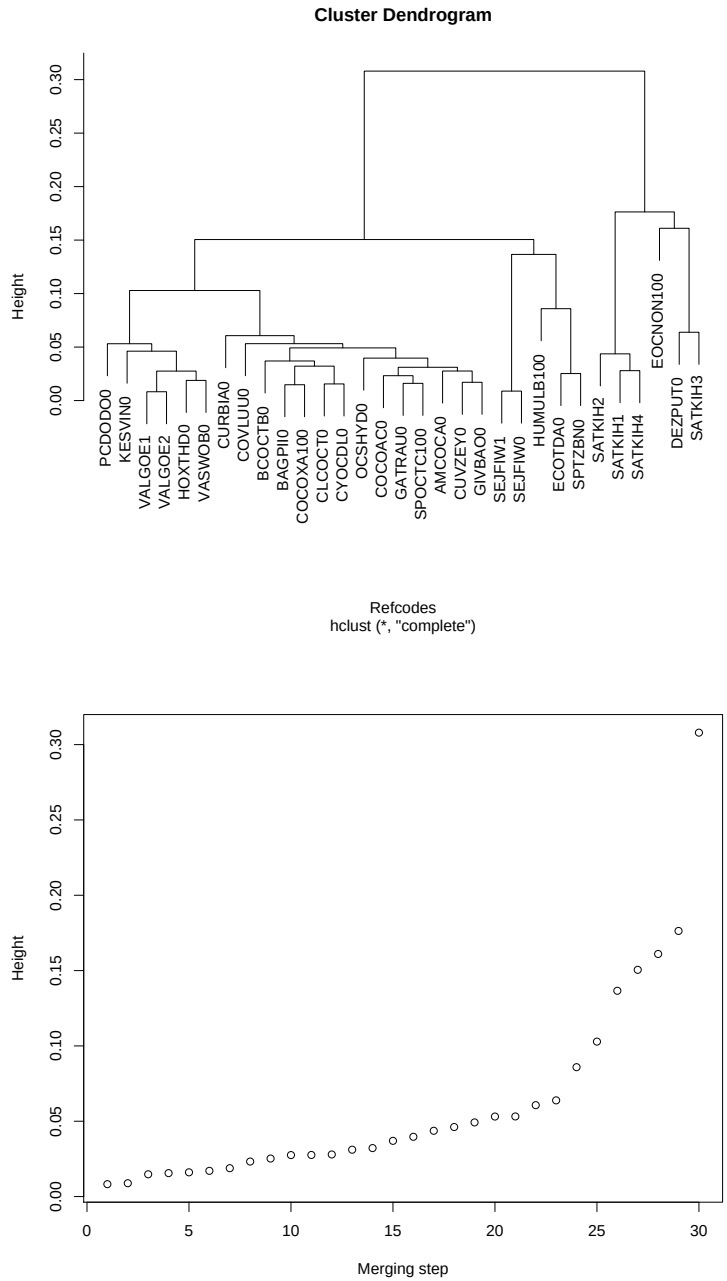


Figura 19: Dendograma para el conjunto 8C1. Método del vecino más lejano

La altura de conglomeración correspondiente a los 8 grupos es 0.064, lo que indica que cada grupo de la partición tiene un diámetro menor de 0.064, es decir, los grupos son bastante compactos.

Grupo	N. fragmentos	Distancias al centroide			Distancia entre extremos
		Min	Max	Media	
1	14	0.010	0.043	0.023	0.061
2	6	0.010	0.035	0.022	0.053
3	3	0.017	0.026	0.021	0.043
4	2	0.032	0.032	0.032	0.064
5	2	0.013	0.013	0.013	0.025
6	2	0.004	0.004	0.004	0.009
7	1	0	0	0	0
8	1	0	0	0	0

Cuadro 1: Distancias a los centroides con una partición de 8 grupos. Criterio del vecino más lejano

4.1.2. Escalado multidimensional

El escalado multidimensional consiste en una técnica de reducción dimensional que se puede utilizar cuando uno sólo dispone de una matriz de distancias. Su objetivo es dar una representación visual de la proximidad entre un conjunto de objetos, asignando a cada punto un lugar en un espacio de dimensión menor, de forma que de manera global, las distancias entre parejas se mantengan.

Por ejemplo, a partir de la matriz obtenida, y del agrupamiento determinado en el apartado 2, la representación gráfica del grupo 1 sería:

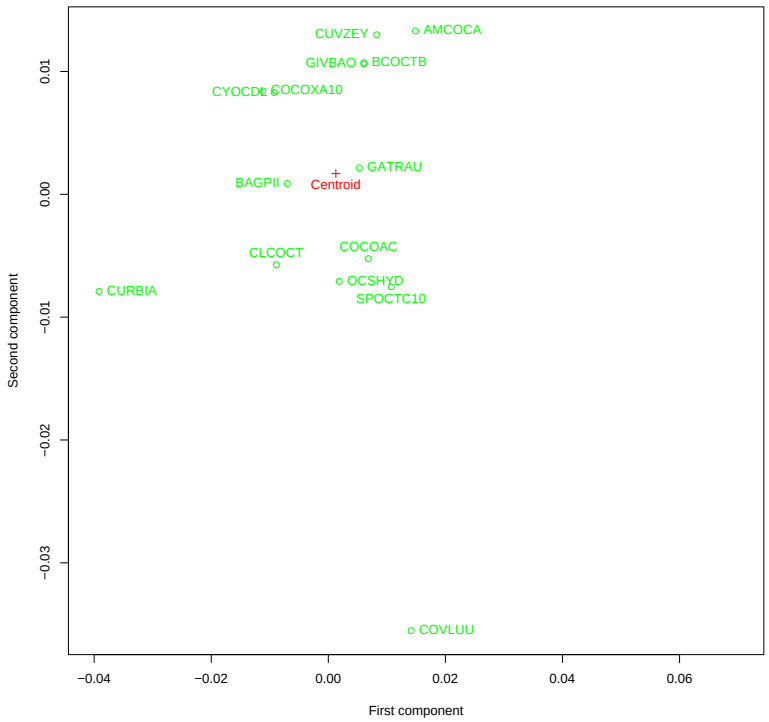


Figura 20: Representación bidimensional del Grupo 1, usando escalado multidimensional.

Se ha de tener en cuenta que es difícil de evaluar en qué grado la representación es coherente con la matriz original, de modo que la gráfica, aunque útil, ha de aceptarse son cautela.

Vemos como el fragmento COVLUU es el más alejado del centroide, lo que es coherente con lo obtenido en la Tabla 1.

4.2. Conjunto 8C1: Comparación de tiempos

Una de las ventajas que ofrece la creación de un código en C, como se comentó en 1.3, es que aprovecha mejor la computadora que un programa de alto nivel. De hecho, esta es una de las bases que justifican la creación de este proyecto.

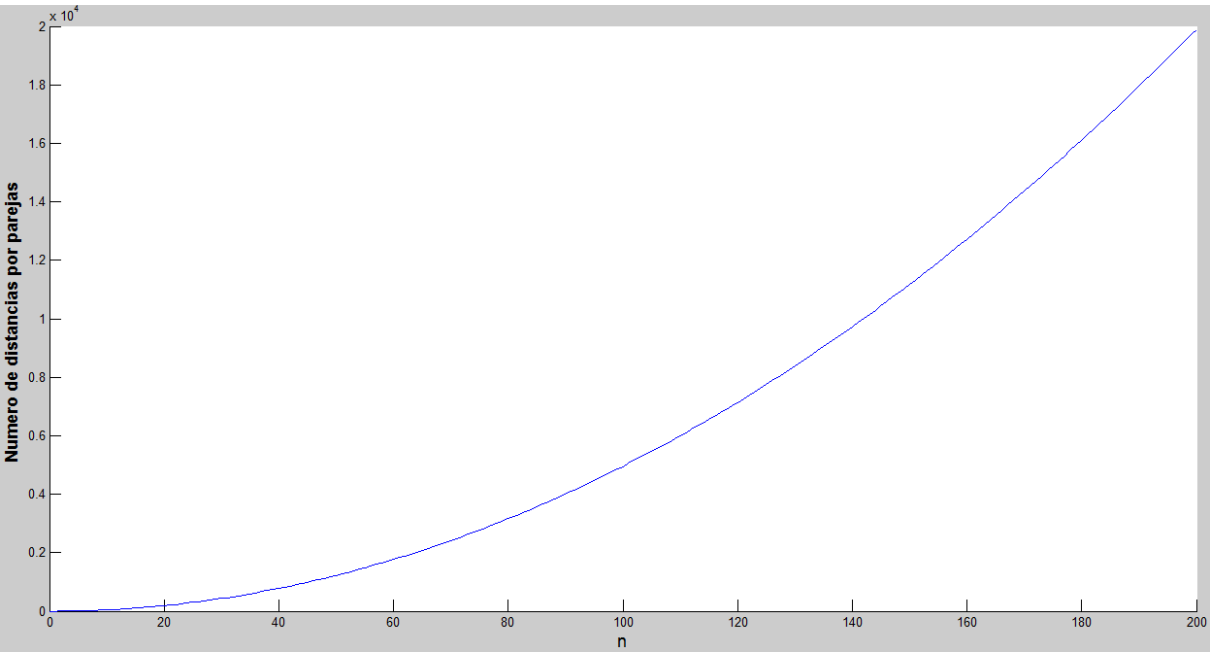
Para comprobar cuanto tiempo se ahorra con el programa en C, lo compararemos con otro código escrito en R, y anterior al presente proyecto que también calcula las distancias entre átomos.

El tiempo que tarda cada programa depende de:

- Número de moléculas que contenga el fichero a estudiar. Obviamente, cuantas más parejas tengamos que estudiar, más tardará el programa.
- Distintas valores de s. Cuantas más formas diferentes haya de numerar cada fragmento, más combinaciones distintas habrá que evaluar.

El número de distancias por parejas para un fichero de n moléculas es

$$distancias = \sum_{i=1}^{n-1}(n-i) = \frac{1}{2}(n^2 - n)$$



El número distancias crece por tanto parabólicamente con el número de moléculas, no de forma lineal.

Sólo se ha evaluado el tiempo de cálculo: para el programa en C no se ha tenido en cuenta el tiempo que se tarda en introducir las variables al programa, ni tampoco se ha cuantificado el tiempo invertido en abrir y cargar los ficheros en R.

En cuanto al número de operaciones que tiene que realizar el ordenador será proporcional a

$$K = \frac{1}{2}(n^2 - n) \cdot s$$

.

Tomamos dos conjuntos de datos diferentes, y los resultados en tiempo de computación son los siguientes:

Nombre	Nº moléculas: n	valores posibles de s	K	tiempo en R	tiempo en C
MOPOMOPO-88	88	2	7656	52 min 3 seg	8 seg
Ciclohexano	219	6	143226	15 h 1 min 1 seg	2 min 2 seg

Vemos por tanto que el ahorro es más considerable. Además cuando el fichero comienza a ser muy grande, el número de operaciones se dispara, con lo que el algoritmo de ya existente de [12] creado para R puede tardar días en calcular la matriz, mientras que el código en C seguirá empleando algunos minutos.

Como conclusión final vemos que la creación del algoritmo de este trabajo ha conseguido reducir notoriamente el tiempo necesario para calcular la matriz de distancias, con lo que el objetivo principal del mismo se ha conseguido.

5. Anexo I: Comprobación de resultados con Matlab

Para comprobar que los cálculos realizados por nuestro algoritmo son correctos, se ha elaborado un código en un lenguaje de alto nivel como es Matlab®.

Para estas comprobaciones se definieron unas funciones auxiliares de uso exclusivo del programador, que sólo sirven para cotejar resultados, y que por tanto no aparecen en la versión final del programa.

5.1. Coordenas “intrinsecas´´: Matriz If

Primero se comprueban las coordenadas ”intrínsecas”del fichero de entrada, es decir, la matriz `If`. Se creó una función que imprimía por pantalla una molécula específica de esta matriz:

```
void imp_molecula(int n)
{
int t=0;
printf("\n");
printf("Molecula %d: ",n);

while(*(nombres+n*long_max_nombre+t) !=';')
{
       printf("%c",*(nombres+n*long_max_nombre+t));
       t++;
}

printf("\n\n");
printf("En coordenadas intrinsecas a escala : ");
printf("\n\n");

for(i=0; i<N; i++)
       {
       for(j=0; j<3; j++){
              printf("\t");
              printf("%.8g ",*(If+N*3*n+i*3+j));}
       printf("\n");
       }

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

Tomaremos dos ejemplos:

Para $N = 6$: Molécula *ACAVIJ1*

El código en Matlab, al que se le introducen las coordenadas fraccionarias manuelamente, es el siguiente:

```
clear all
clc
format compact

alfa_deg=90.000
```

```
beta_deg=90.000
gamma_deg=90.000

a=16.534
b=16.534
c=19.886

alfa= alfa_deg/180*pi;
beta= beta_deg/180*pi;
gamma= gamma_deg/180*pi;

fract1=[0.28845;0.62250;-0.00318];
fract2=[0.26215;0.70595;0.02636];
fract3=[0.29917;0.77881;-0.01393];
fract4=[0.26697;0.77615;-0.08780];
fract5=[0.22365;0.69527;-0.10237];
fract6=[0.27983;0.62608;-0.08097];

fract=[fract1';fract2';fract3';fract4';fract5';fract6']
%% Matriz de cambio de coordenadas

v=sqrt(1-cos(alfa)^2-cos(beta)^2-cos(gamma)^2+...
2*cos(alfa)*cos(beta)*cos(gamma));

M=[a, b*cos(gamma),c*cos(beta);0, b*sin(gamma), ...
c*(cos(alfa)-cos(beta)*cos(gamma))/sin(gamma);0, 0, c*v/sin(gamma)];


cart1=M*fract1;
cart2=M*fract2;
cart3=M*fract3;
cart4=M*fract4;
cart5=M*fract5;
cart6=M*fract6;

cartesian=[cart1';cart2';cart3';cart4';cart5';cart6']
%%

xc=(cart1(1)+cart2(1)+cart3(1)+cart4(1)+cart5(1)+cart6(1))/6;
yc=(cart1(2)+cart2(2)+cart3(2)+cart4(2)+cart5(2)+cart6(2))/6;
zc=(cart1(3)+cart2(3)+cart3(3)+cart4(3)+cart5(3)+cart6(3))/6;

for(i=1:6)     %%Cambio de origen del coordenadas al c.d.g.
       cartesian_cdg(i,1)=cartesian(i,1)-xc;
       cartesian_cdg(i,2)=cartesian(i,2)-yc;
       cartesian_cdg(i,3)=cartesian(i,3)-zc;
end
cartesian_cdg

R1_0=zeros(1,3);
R2_0=zeros(1,3);
for(j=1:6)
```

```
    R1_0(1)=R1_0(1)+cartesian_cdg(j,1)*sin(2*pi*(j-1)/6);
    R1_0(2)=R1_0(2)+cartesian_cdg(j,2)*sin(2*pi*(j-1)/6);
    R1_0(3)=R1_0(3)+cartesian_cdg(j,3)*sin(2*pi*(j-1)/6);
end

for(j=1:6)
    R2_0(1)=R2_0(1)+cartesian_cdg(j,1)*cos(2*pi*(j-1)/6);
    R2_0(2)=R2_0(2)+cartesian_cdg(j,2)*cos(2*pi*(j-1)/6);
    R2_0(3)=R2_0(3)+cartesian_cdg(j,3)*cos(2*pi*(j-1)/6);
end
R1_0;
R2_0;


R1=R1_0/sqrt(dot(R1_0,R1_0));

R2_1=R2_0-dot(R1,R2_0)*R1;
R2=R2_1/sqrt(dot(R2_1,R2_1));

n=cross(R1,R2);

G=[R1; R2; n]

c1=[cartesian_cdg(1,1); cartesian_cdg(1,2); cartesian_cdg(1,3)];
c2=[cartesian_cdg(2,1); cartesian_cdg(2,2); cartesian_cdg(2,3)];
c3=[cartesian_cdg(3,1); cartesian_cdg(3,2); cartesian_cdg(3,3)];
c4=[cartesian_cdg(4,1); cartesian_cdg(4,2); cartesian_cdg(4,3)];
c5=[cartesian_cdg(5,1); cartesian_cdg(5,2); cartesian_cdg(5,3)];
c6=[cartesian_cdg(6,1); cartesian_cdg(6,2); cartesian_cdg(6,3)];

c1_i=G*c1; %%Coordenadas intrinsecas sin escalar
c2_i=G*c2;
c3_i=G*c3;
c4_i=G*c4;
c5_i=G*c5;
c6_i=G*c6;

intrinsecas=[c1_i';c2_i';c3_i';c4_i';c5_i';c6_i']

d12=sqrt((c1_i(1)-c2_i(1))^2+(c1_i(2)-c2_i(2))^2+(c1_i(3)-c2_i(3))^2);
d23=sqrt((c2_i(1)-c3_i(1))^2+(c2_i(2)-c3_i(2))^2+(c2_i(3)-c3_i(3))^2);
d34=sqrt((c3_i(1)-c4_i(1))^2+(c3_i(2)-c4_i(2))^2+(c3_i(3)-c4_i(3))^2);
d45=sqrt((c4_i(1)-c5_i(1))^2+(c4_i(2)-c5_i(2))^2+(c4_i(3)-c5_i(3))^2);
d56=sqrt((c5_i(1)-c6_i(1))^2+(c5_i(2)-c6_i(2))^2+(c5_i(3)-c6_i(3))^2);
d61=sqrt((c6_i(1)-c1_i(1))^2+(c6_i(2)-c1_i(2))^2+(c6_i(3)-c1_i(3))^2);

dist_media=(d12+d23+d34+d45+d56+d61)/6

disp('ACAVIJ1')
If=intrinsecas/dist_media
```

Y el resultado que obtenemos es:

```
alfa_deg =
    90
beta_deg =
    90
gamma_deg =
    90
a =
   16.5340
b =
   16.5340
c =
   19.8860
fract =
    0.2884    0.6225   -0.0032
    0.2621    0.7059    0.0264
    0.2992    0.7788   -0.0139
    0.2670    0.7762   -0.0878
    0.2236    0.6953   -0.1024
    0.2798    0.6261   -0.0810
M =
   16.5340    0.0000    0.0000
         0   16.5340    0.0000
         0         0   19.8860
cartesian =
    4.7692   10.2924   -0.0632
    4.3344   11.6722    0.5242
    4.9465   12.8768   -0.2770
    4.4141   12.8329   -1.7460
    3.6978   11.4956   -2.0357
    4.6267   10.3516   -1.6102
cartesian_cdg =
    0.3044   -1.2945    0.8048
   -0.1304    0.0853    1.3922
    0.4817    1.2899    0.5910
   -0.0507    1.2459   -0.8780
   -0.7670   -0.0913   -1.1677
    0.1619   -1.2353   -0.7422
Gt =
    0.1978    0.5589    0.8053
    0.1225   -0.8292    0.5454
    0.9726   -0.0093   -0.2325
intrinsecas =
   -0.0152    1.5496    0.1210
    1.1430    0.6726   -0.4513
    1.2921   -0.6883    0.3191
   -0.0207   -1.5182    0.1433
   -1.1431   -0.6551   -0.4736
   -1.2560    0.6394    0.3415
dist_media =
    1.5547
ACAVIJ1
If =
```

```
-0.0098    0.9967    0.0778
 0.7352    0.4326   -0.2903
 0.8311   -0.4427    0.2053
-0.0133   -0.9765    0.0922
-0.7353   -0.4213   -0.3046
-0.8079    0.4113    0.2196
```

Si ejecutamos nuestro código, como la molécula *ACAVIJ1* es la primera, pondremos en el main: `imp_molecula(0)`. Lo que obtenemos por pantalla es:

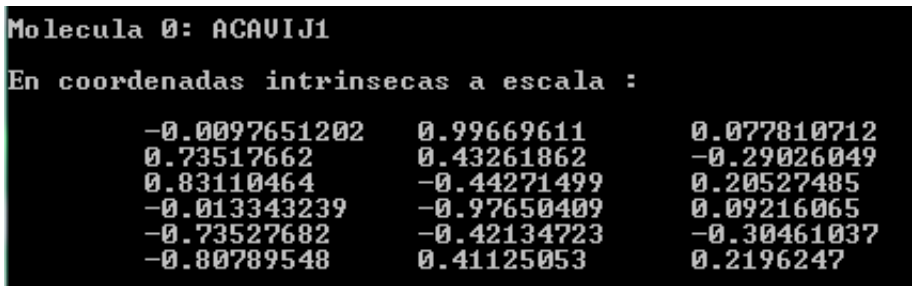


Figura 21: Ejemplo de la función imprimir_molecula para ACAVIJ1

Vemos que los resultados coinciden, por lo que podemos afirmar afirmar que el código funciona correctamente.

Para $N = 8$: Molécula *AMCOCA0*

El código en Matlab es:

```
a=26.026
b=7.087
c=6.149

alfa_deg=90.000
beta_deg=90.000
gamma_deg=90.000

fract1=[0.15660;-0.60310;0.24620];
fract2=[0.11150;-0.70830;0.13850];
fract3=[0.08770;-0.88320;0.25480];
fract4=[0.05940;-0.85020;0.46460];
fract5=[0.02400;-0.66950;0.47640];
fract6=[0.04570;-0.50280;0.58100];
fract7=[0.09260;-0.38830;0.47810];
fract8=[0.14440;-0.50740;0.46950];

fract=[fract1';fract2';fract3';fract4';fract5';...
fract6';fract7';fract8']

alfa= alfa_deg/180*pi;
beta= beta_deg/180*pi;
gamma= gamma_deg/180*pi;
```

```
%% Matriz de cambio de coordenadas

v=sqrt(1-cos(alfa)^2-cos(beta)^2-cos(gamma)^2+...
2*cos(alfa)*cos(beta)*cos(gamma));

M=[a, b*cos(gamma),c*cos(beta);0, b*sin(gamma),...
c*(cos(alfa)-cos(beta)*cos(gamma))/sin(gamma);0, 0, c*v/sin(gamma)];



cart1=M*fract1;
cart2=M*fract2;
cart3=M*fract3;
cart4=M*fract4;
cart5=M*fract5;
cart6=M*fract6;
cart7=M*fract7;
cart8=M*fract8;


cartesian=[cart1';cart2';cart3';cart4';cart5';...
cart6';cart7';cart8']

%%

xc=(cart1(1)+cart2(1)+cart3(1)+cart4(1)+cart5(1)+cart6(1)+
cart7(1)+cart8(1))/8;
yc=(cart1(2)+cart2(2)+cart3(2)+cart4(2)+cart5(2)+cart6(2)+
cart7(2)+cart8(2))/8;
zc=(cart1(3)+cart2(3)+cart3(3)+cart4(3)+cart5(3)+cart6(3)+
cart7(3)+cart8(3))/8;

for(i=1:8)     %%Cambio de origen del coordenadas al c.d.g.
       cartesian_cdg(i,1)=cartesian(i,1)-xc;
       cartesian_cdg(i,2)=cartesian(i,2)-yc;
       cartesian_cdg(i,3)=cartesian(i,3)-zc;
end
cartesian_cdg

R1_0=zeros(1,3);
R2_0=zeros(1,3);
for(j=1:8)
    R1_0(1)=R1_0(1)+cartesian_cdg(j,1)*sin(2*pi*(j-1)/8);
    R1_0(2)=R1_0(2)+cartesian_cdg(j,2)*sin(2*pi*(j-1)/8);
    R1_0(3)=R1_0(3)+cartesian_cdg(j,3)*sin(2*pi*(j-1)/8);
end

for(j=1:8)
    R2_0(1)=R2_0(1)+cartesian_cdg(j,1)*cos(2*pi*(j-1)/8);
    R2_0(2)=R2_0(2)+cartesian_cdg(j,2)*cos(2*pi*(j-1)/8);
```

```
    R2_0(3)=R2_0(3)+cartesian_cdg(j,3)*cos(2*pi*(j-1)/8);
end
R1_0;
R2_0;


R1=R1_0/sqrt(dot(R1_0,R1_0));

R2_1=R2_0-dot(R1,R2_0)*R1;
R2=R2_1/sqrt(dot(R2_1,R2_1));

n=cross(R1,R2);

G=[R1; R2; n]

c1=[cartesian_cdg(1,1); cartesian_cdg(1,2); cartesian_cdg(1,3)];
c2=[cartesian_cdg(2,1); cartesian_cdg(2,2); cartesian_cdg(2,3)];
c3=[cartesian_cdg(3,1); cartesian_cdg(3,2); cartesian_cdg(3,3)];
c4=[cartesian_cdg(4,1); cartesian_cdg(4,2); cartesian_cdg(4,3)];
c5=[cartesian_cdg(5,1); cartesian_cdg(5,2); cartesian_cdg(5,3)];
c6=[cartesian_cdg(6,1); cartesian_cdg(6,2); cartesian_cdg(6,3)];
c7=[cartesian_cdg(7,1); cartesian_cdg(7,2); cartesian_cdg(7,3)];
c8=[cartesian_cdg(8,1); cartesian_cdg(8,2); cartesian_cdg(8,3)];

c1_i=G*c1; %%Coordenadas intrinsecas sin escalar
c2_i=G*c2;
c3_i=G*c3;
c4_i=G*c4;
c5_i=G*c5;
c6_i=G*c6;
c7_i=G*c7;
c8_i=G*c8;

intrinsecas=[c1_i';c2_i';c3_i';c4_i';c5_i';c6_i';c7_i';c8_i']


d12=sqrt((c1_i(1)-c2_i(1))^2+(c1_i(2)-c2_i(2))^2+...
(c1_i(3)-c2_i(3))^2);
d23=sqrt((c2_i(1)-c3_i(1))^2+(c2_i(2)-c3_i(2))^2+...
(c2_i(3)-c3_i(3))^2);
d34=sqrt((c3_i(1)-c4_i(1))^2+(c3_i(2)-c4_i(2))^2+...
(c3_i(3)-c4_i(3))^2);
d45=sqrt((c4_i(1)-c5_i(1))^2+(c4_i(2)-c5_i(2))^2+...
(c4_i(3)-c5_i(3))^2);
d56=sqrt((c5_i(1)-c6_i(1))^2+(c5_i(2)-c6_i(2))^2+...
(c5_i(3)-c6_i(3))^2);
d67=sqrt((c6_i(1)-c7_i(1))^2+(c6_i(2)-c7_i(2))^2+...
(c6_i(3)-c7_i(3))^2);
d78=sqrt((c7_i(1)-c8_i(1))^2+(c7_i(2)-c8_i(2))^2+...
(c7_i(3)-c8_i(3))^2);
d81=sqrt((c8_i(1)-c1_i(1))^2+(c8_i(2)-c1_i(2))^2+...
(c8_i(3)-c1_i(3))^2);
```

```
dist_media=(d12+d23+d34+d45+d56+d67+d78+d81)/8

disp('AMCOCA0')
If=intrinsecas/dist_media
```

Y el resultado:

```
a =
   26.0260
b =
    7.0870
c =
    6.1490
alfa_deg =
    90
beta_deg =
    90
gamma_deg =
    90
fract =
    0.1566   -0.6031    0.2462
    0.1115   -0.7083    0.1385
    0.0877   -0.8832    0.2548
    0.0594   -0.8502    0.4646
    0.0240   -0.6695    0.4764
    0.0457   -0.5028    0.5810
    0.0926   -0.3883    0.4781
    0.1444   -0.5074    0.4695
M =
   26.0260    0.0000    0.0000
         0    7.0870    0.0000
         0         0    6.1490
cartesian =
    4.0757   -4.2742    1.5139
    2.9019   -5.0197    0.8516
    2.2825   -6.2592    1.5668
    1.5459   -6.0254    2.8568
    0.6246   -4.7447    2.9294
    1.1894   -3.5633    3.5726
    2.4100   -2.7519    2.9398
    3.7582   -3.5959    2.8870
cartesian_cdg =
    1.7272    0.2551   -0.8758
    0.5534   -0.4904   -1.5381
   -0.0660   -1.7299   -0.8230
   -0.8026   -1.4961    0.4671
   -1.7239   -0.2154    0.5397
   -1.1591    0.9660    1.1828
    0.0615    1.7774    0.5501
    1.4096    0.9334    0.4972
Gt =
```

```
   -0.0678   -0.8814   -0.4676
    0.8720    0.1754   -0.4571
    0.4849   -0.4387    0.7566
intrinsecas =
    0.0676    1.9511    0.0628
    1.1139    1.0995   -0.6803
    1.9140    0.0151    0.1042
    1.1545   -1.1758    0.6206
    0.0544   -1.7876   -0.3330
   -1.3259   -1.3819   -0.0908
   -1.8279    0.1140   -0.3337
   -1.1506    1.1656    0.6502
dist_media =
    1.5492
AMCOCA0
If =
    0.0437    1.2594    0.0406
    0.7190    0.7097   -0.4391
    1.2355    0.0097    0.0672
    0.7453   -0.7590    0.4006
    0.0351   -1.1539   -0.2150
   -0.8559   -0.8920   -0.0586
   -1.1799    0.0736   -0.2154
   -0.7427    0.7524    0.4197
```

De nuevo, como la molécula *AMCOCA0* es la primera, pondremos en el main: `imp_molecula(0)`, y obtenemos por pantalla:

```
Molecula 0: AMCOCA0

En coordenadas intrinsecas a escala :

        0.043657191     1.2594328       0.040553045
        0.71903133      0.70972139      -0.43914399
        1.2354684       0.0097194463    0.067247532
        0.74526525      -0.75896853     0.40057141
        0.035092134     -1.1539184      -0.21497199
        -0.85585618     -0.89200765     -0.058609523
        -1.1799282      0.073597446     -0.2153846
        -0.74272877     0.75242269      0.41973901
```

Figura 22: Ejemplo de la función imprimir_molecula para AMCOCA0

Los resultados coinciden así que el programa funciona correctamente para cualquier valor de N.

Como nota final, añadir que además del resultado final, `If`, durante el desarrollo del programa también se probaron los resultados del centro geométrico, los vectores directores, etc..

5.2. Comprobación de la distancia d

Pasamos ahora a comprobar la distancia d entre una pareja de moléculas. Recordemos que d es:

$$d(F_1, F_2) = \min_{\substack{s\in\mathcal{S}\\ v,a,b\in\{0,1\}}} \min_{\gamma\in[0,2\pi[} \|I_{F_1} - (T^{(s-1)}D^v I_{F_2} M^a J^b)(R_z(\gamma))^T\|$$

Como ya se explicó anteriormente, para cada *condición*, tenemos una valor de d que sólo depende de γ. El código de Matlab que se ha realizado crea gráficas como la siguiente.

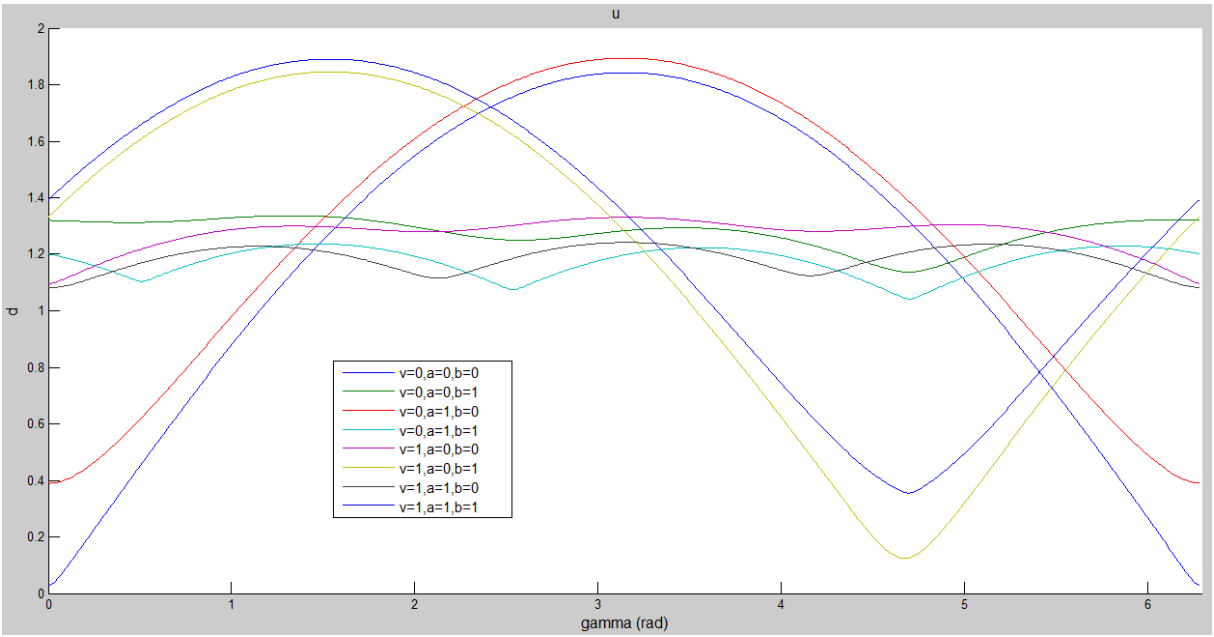


Figura 23: Valor de d(γ) para un u en todas las *condiciones*

De nuevo tomaremos dos ejemplos, uno para $N = 6$ y otro para $N = 8$.

Para $N = 6$: Moléculas *ACAVIJ1* y *DIVLOJ1*

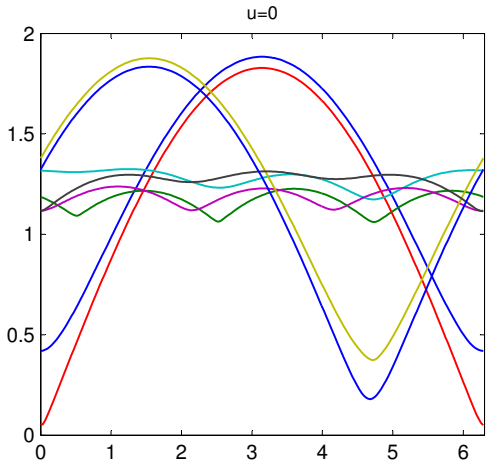
El código en Matlab es:

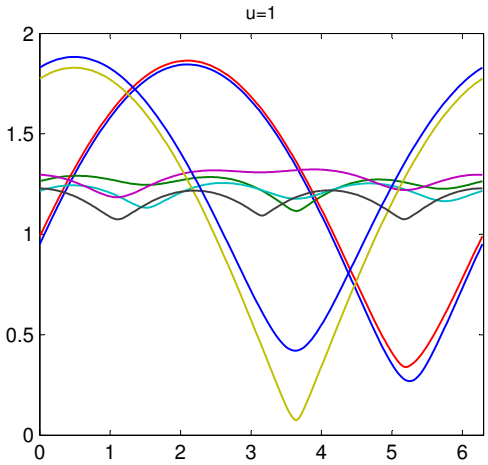
```
If1=[
    -0.0097651202   0.99669611     0.077810712 ;
    0.73517662      0.43261862     -0.29026049 ;
    0.83110464      -0.44271499    0.20527485 ;
    -0.013343239    -0.97650409    0.09216065 ;
    -0.73527682     -0.42134723    -0.30461037 ;
    -0.80789548     0.41125053     0.2196247 ]; %ACAVIJ1


If2 =[
    0.01202096      0.950656       -0.14615548 ;
    0.7117582       0.42609265     0.32992834 ;
    0.84490687      -0.42751932    -0.18371502 ;
    -0.018625582    -0.94779915    -0.14627022 ;
    -0.71648687     -0.42608437    0.33004311 ;
    -0.8335762      0.42465755     -0.18382977 ]; %DIVLOJ1

N=6;
```

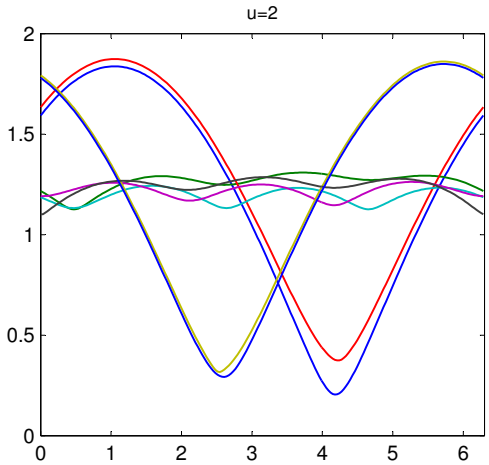
```
%% Cálculo de la minima distancia: d

T=[0 1 0 0 0 0;0 0 1 0 0 0;0 0 0 1 0 0;0 0 0 0 1 0;...
0 0 0 0 0 1;1 0 0 0 0 0];
D=[1 0 0 0 0 0;0 0 0 0 0 1;0 0 0 0 1 0;0 0 0 1 0 0;...
0 0 1 0 0 0;0 1 0 0 0 0];
M=[1 0 0;0 1 0;0 0 -1];
J=[0 1 0;1 0 0;0 0 -1];


n=1; %Inicio de contador para grabar datos en la matriz de distancias
hold on
puntos=400; %Divisiones del intervalo [0,2pi]

for u=0:1:5
n=1;

% figure1 = figure('PaperSize',[20.98 29.68]);
for v=0:1:1
for a=0:1:1
for b=0:1:1


for k=0:puntos
            gamma=k*2*pi/puntos;
            giro(k+1)=gamma;
            R=[cos(gamma), -sin(gamma),0;...
                sin(gamma), cos(gamma), 0;0, 0, 1];

            If2_nuevo=(T^u)*(D^v)*If2*(M^a)*(J^b)*R;
            If=If1-If2_nuevo;

            norma_i=0;
            for i=1:N
                suma=0;
                for j=1:3
                    suma=If(i,j)^2+suma;
                end
                norma_i=norma_i+sqrt(suma);
            end
            norma=norma_i/N;

            distancia(n,k+1)=norma;
        end %k (fin de giro)
        n=n+1;

end

end

end
```

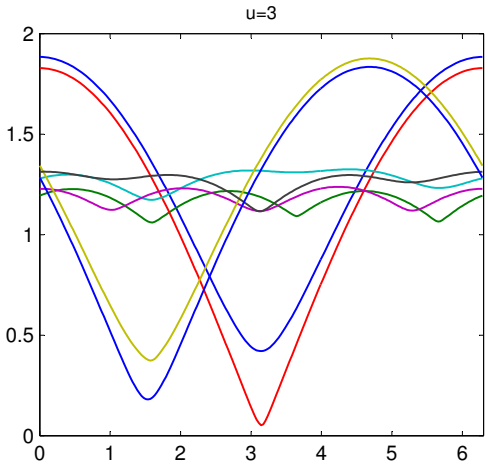
```
        subplot(3,2,u+1);
        plot(giro,distancia,'LineWidth',1)
        axis([0 6.3 0 2.0])
%         axis([0 6.3 0 0.035])
        title(['u=',num2str(u)],'FontSize',10)
        [minimo,condicion]=min(min(distancia,[],2));
        disp(['u=',num2str(u), ': dmin= ',num2str(minimo),', ...
        condicion=',num2str(condicion-1)])
end
```

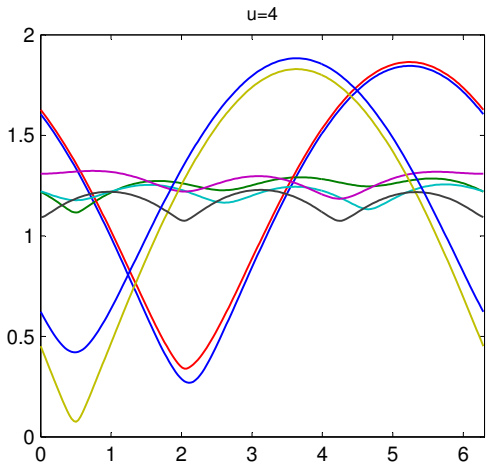
Al ejecutarlo, vemos gráficamente el aspecto que tienen las curvas de la distancia según gamma, como se muestra en la página siguiente.

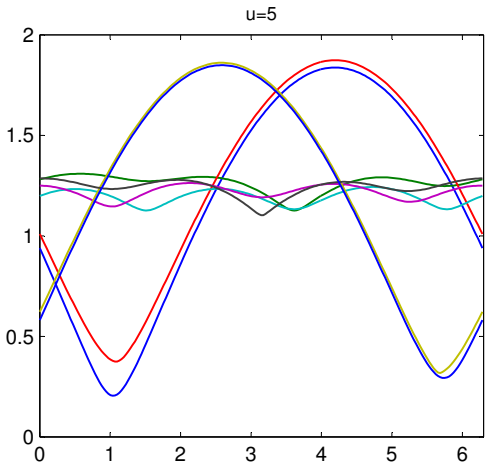

u=0
2
1.5
1
0.5
0
0 1 2 3 4 5 6


u=1
2
1.5
1
0.5
0
0 1 2 3 4 5 6

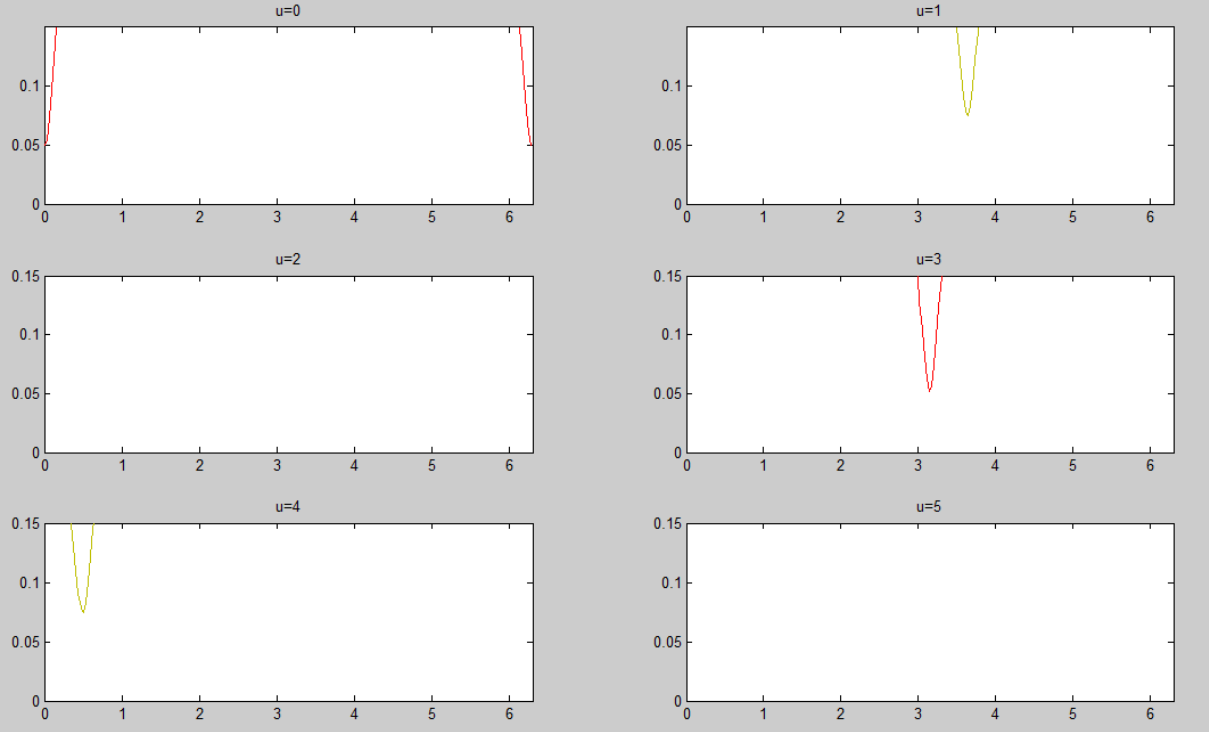

u=2
2
1.5
1
0.5
0
0 1 2 3 4 5 6


u=3
2
1.5
1
0.5
0
0 1 2 3 4 5 6


u=4
2
1.5
1
0.5
0
0 1 2 3 4 5 6


u=5
2
1.5
1
0.5
0
0 1 2 3 4 5 6

Si hacemos un zoom a la página anterior en el eje y de la distancia, para fijarnos sólo en los valores más bajos, candidatos a mínimos tenemos:



Vemos que hay tres posibles mínimos, dos para $u = 0$ y uno para $u = 3$.

Además obtenemos como salida de Matlab los mínimos para cada u y en qué condición se halla:

```
u=0: dmin= 0.050172, condicion=2
u=1: dmin= 0.075046, condicion=5
u=2: dmin= 0.20405, condicion=0
u=3: dmin= 0.0517, condicion=2
u=4: dmin= 0.074657, condicion=5
u=5: dmin= 0.20431, condicion=0
```

Para simplificar, el valor de *condición* que obtenemos se ha definido como el numero binario de base: $2 \times v + 2 \times b + 2 \times a$. Es decir, condición=7 equivale en binario a 1 1 1, o lo que es lo mismo, $v = 1, a = 1, b = 1$. Del mismo modo, condión=0 quiere decir $v = 0, a = 0, b = 0$.

Observamos que para este ejemplo, la menor de todas las distancias es 0.050172, y está en u=0 y la condición 2, es decir: $s = 1, v = 0, a = 1, b = 0$.

Si ejecutamos nuestro programa en C, en el fichero de detalle observamos:

```
0 28:ACAVIJ1 DIVLOJ1;   d: 0.049685, s=1, v=0, a=1, b=0,
gamma=0.007573
```

La diferencia entre los valores de d se debe a que el código programado en Matlab, para que haga las curvas rápido no se ha precisado mucho, sólo se han cogido 400 puntos por curva es decir una tolerancia de $\frac{2\times\pi}{400} \simeq 0{,}016$, muy superior al 0.0001 de nuestro algoritmo.

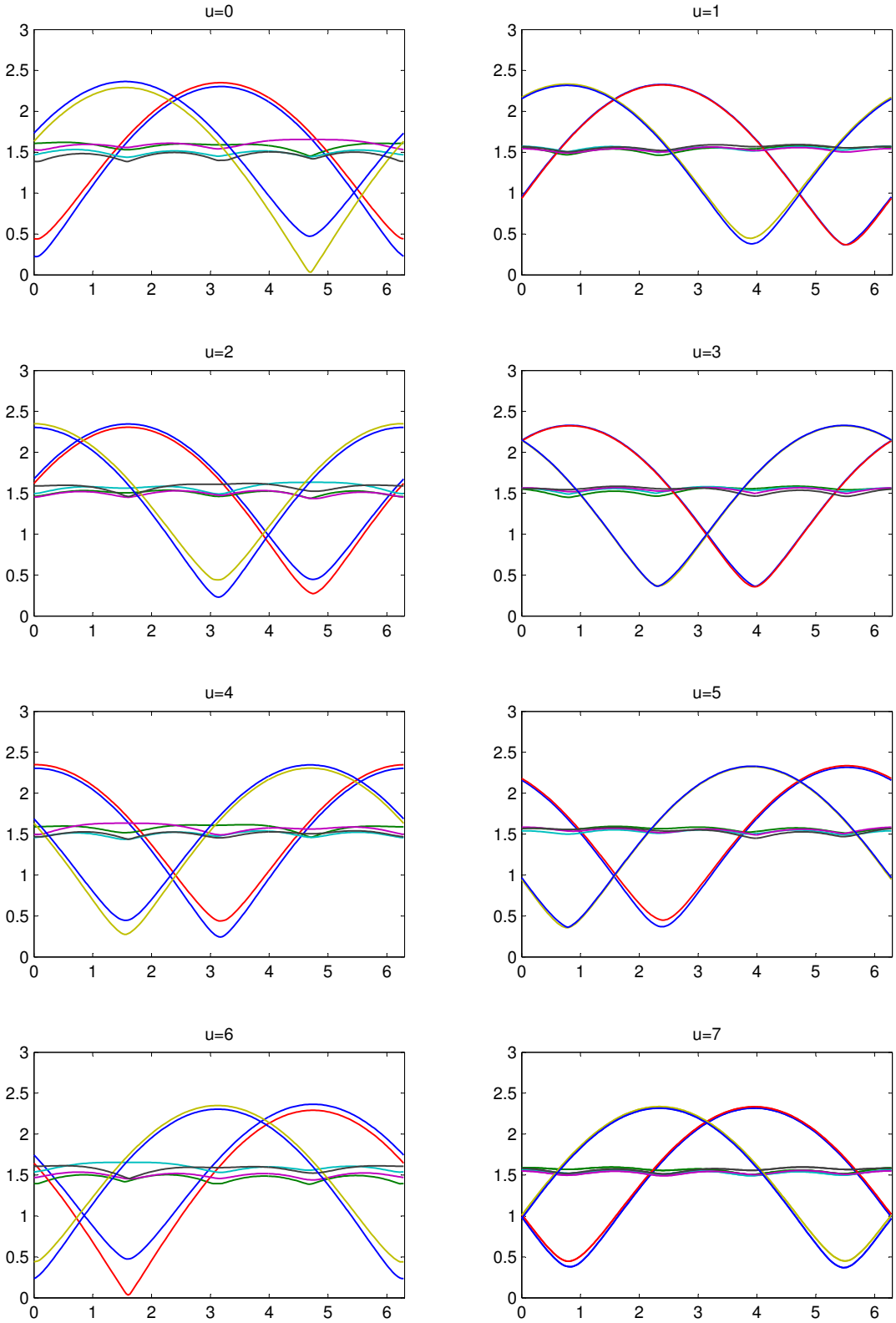
Por lo tanto, el algoritmo ha funcionado correctamente.

Para $N = 8$: Moléculas *AMCOCA0* y *BAGPII0*

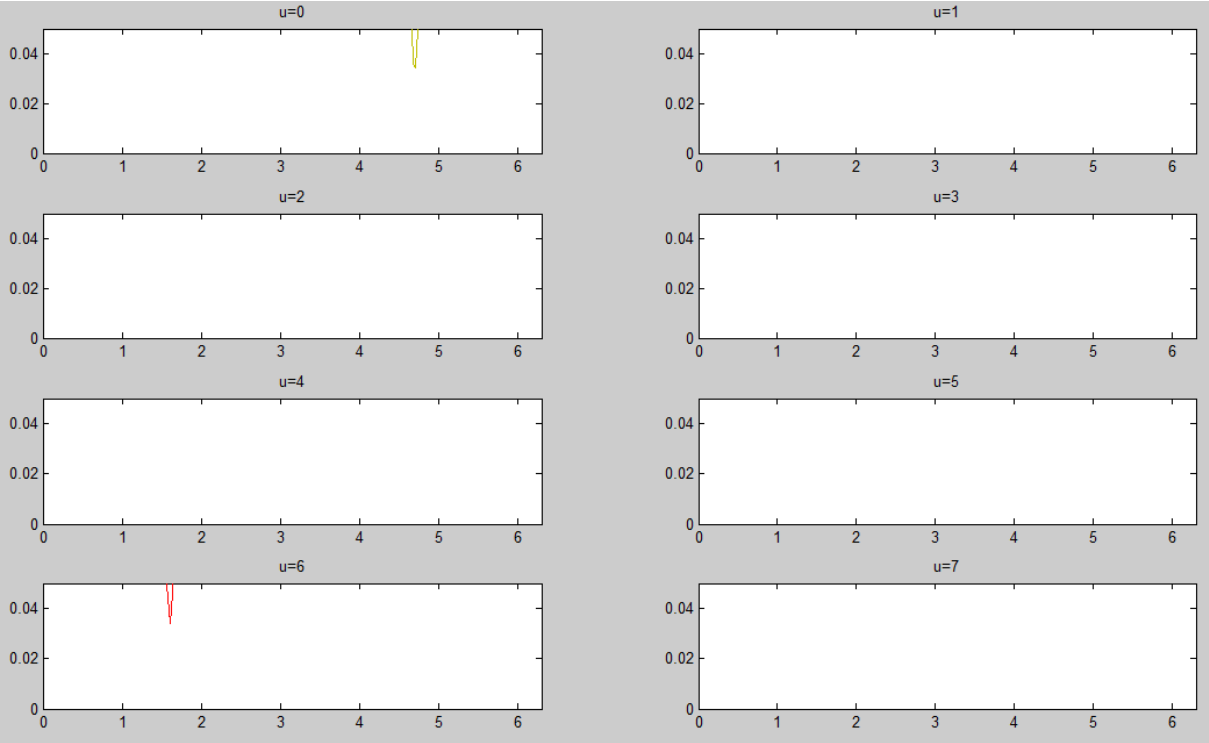
El código en Matlab es muy parecido, sólo que ahora los datos de los fragmentos a estudiar son:

```
If1=[
    0.043657191     1.2594328       0.040553045 ;
    0.71903133      0.70972139      -0.43914399 ;
    1.2354684       0.0097194463    0.067247532 ;
    0.74526525      -0.75896853     0.40057141 ;
    0.035092134     -1.1539184      -0.21497199 ;
    -0.85585618     -0.89200765     -0.058609523 ;
    -1.1799282      0.073597446     -0.2153846 ;
    -0.74272877     0.75242269      0.41973901] %AMCOCA0

If2 =[
    -0.071945213    1.240748        -0.060562465 ;
    0.70900369      0.72516739      -0.41834712 ;
    1.1448084       0.088139832     0.22121374 ;
    0.91885328      -0.8722989      0.060780969 ;
    -0.024873432    -1.1523596      0.21425585 ;
    -0.70879298     -0.76613259     -0.40950736 ;
    -1.2276102      0.016688809     -0.066106252 ;
    -0.7394433      0.72004747      0.45827281] %BAGPII0
```

Las curvas que obtenemos son:


u=0
u=1
u=2
u=3
u=4
u=5
u=6
u=7

Haciendo zoom:



En este caso, los mínimos pueden estar en $u = 0$ y $u = 7$.

Matlab nos da:

```
u=0: dmin= 0.031164, condicion=5
u=1: dmin= 0.36664, condicion=0
u=2: dmin= 0.22984, condicion=7
u=3: dmin= 0.35627, condicion=2
u=4: dmin= 0.24274, condicion=0
u=5: dmin= 0.35698, condicion=5
u=6: dmin= 0.033987, condicion=2
u=7: dmin= 0.36823, condicion=7
```

El mínimo global es dmin= 0.031164, y se da en $u = 0$, condicion=5, o lo que es lo mismo, $s = 1, v = 1, a = 0, b = 1$.

En el fichero de detalle que nos da nuestro algoritmo encontramos:

```
0 1:AMCOCA0 BAGPII0;     d: 0.030904, s=1, v=1, a=0, b=1,
gamma=4.700941
```

Por lo tanto, podemos asegurar que nuestro programa funciona correctamente para cualquier valor de N.

Referencias

[1] Parabolic interpolation and brent's method in one dimension. Numerical Recipes: The Art of Scientific Computing, Third Edition. *Section 10.2*,2003.

[2] K Nolsøe, M Kessler, J Pérez y H Madsen, Bayesian conformational analysis of ring molecules through Reversible Jump MCMC. *J. Chemometrics.*, B19:412–426, 2005.

[3] F.H Allen. The Cambridge Structural Database: a quarter of a million crystal structures and rising. *Acta Cryst.*, B58:380–389, 2002.

[4] F.H Allen, M. J Doyle, and R Taylor. Automated conformational analysis from crystallographic data. 1. A symmetry-modified Single-Linkage clustering algorithm for three-dimensional pattern recognition. *Acta Cryst.*, 47:29–40, 1991.

[5] F.H Allen, M. J Doyle, and R Taylor. Automated conformational analysis from crystallographic data. 2. Symmetry-modified Jarvis-Patrick and Complete-Linkage clustering algorithm for three-dimensional pattern recognition. *Acta Cryst.*, B47:41–49, 1991.

[6] F.H Allen, J.A.K Howard, and N.A Pitchford. Symmetry-modified conformational mapping and classification of the medium rings from crystallographic data. IV. Cyclooctane and related eight-membered rings. *Acta Cryst.*, B52:882–891, 1996.

[7] D Cremer and J.A Pople. A general definition of ring puckering coordinates. *Journal of the American Chemical Society*, 97(6):1354–1358, 1975.

[8] C Giacovazzo, H.L Monaco, G Artioli, D Viterbo, G Ferraris, G Gilli, G Zanotti, and M Catti. Fundamentals of crystallography. 2002.

[9] Carlos F. Gonzalez. *Mecánica del Sólido Rígido*. Ariel Editorial, Barcelona, 2003.

[10] J.B Hendrickson. Molecular geometry. VII. Modes of interconversion in the medium rings. *J. Am. Chem. Soc.*, 89:7047–7054, 1967.

[11] M Kessler, M. C Bueso, and J Pérez. Model-based conformational clustering of ring molecules. *J. Chemometrics.*, 21:53–64, 2007.

[12] M Kessler and J Pérez. Molecular superposition of n-membered rings: application to conformational analysis.Prepublicaciòn.Sometido.

[13] J Pérez, KÑolsøe, M Kessler, L García, E Pérez, and Serrano J. L. Bayesian methods for the conformational classification of eight-membered rings. *Acta Cryst.*, B61:585–594, 2005.

[14] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2008. ISBN 3-900051-07-0.

[15] H Schildt. C. Manual de Referencia. McGraw-Hill, 2001.

6. Anexo II: Código en C completo

En las siguientes páginas se incluye el código completo del programa en C desarrollado en este proyecto.

```
/*    Implementación de un algoritmo de superposición 3D de fragmentos químicos asociados
      moléculas en forma de anillo

      Roberto Castellano Sánchez
      Escuela Técnica Superior de Ingenieros Industriales
      Universidad Politécnica de Cartagena. Marzo 2010.

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>


/******************    LIBRERIA DE CÁLCULO MATRICIAL             **********************

///////   Macros para vectores de tres componentes  ///////

// Producto vectorial: a = b x c
#define crossProduct(a,b,c) \
      (a)[0] = (b)[1] * (c)[2] - (c)[1] * (b)[2]; \
      (a)[1] = (b)[2] * (c)[0] - (c)[2] * (b)[0]; \
      (a)[2] = (b)[0] * (c)[1] - (c)[0] * (b)[1];

// Producto escalar: a = b · c
#define dotProduct(a,b,c) \
      (a)= (b)[0] * (c)[0] + (b)[1] * (c)[1] + (b)[2] * (c)[2];

// Normalización de un vector: v_n = v / |v|
#define normalize(v_n,v) \
      (v_n)[0] = (v)[0] / pow((v)[0]*(v)[0]+(v)[1]*(v)[1]+(v)[2]*(v)[2],0.5); \
      (v_n)[1] = (v)[1] / pow((v)[0]*(v)[0]+(v)[1]*(v)[1]+(v)[2]*(v)[2],0.5); \
      (v_n)[2] = (v)[2] / pow((v)[0]*(v)[0]+(v)[1]*(v)[1]+(v)[2]*(v)[2],0.5);

// Multiplicación de un vector por un escalar: a = constant * b
#define scalar(a,constant,b) \
      (a)[0] = constant*(b)[0]; \
      (a)[1] = constant*(b)[1]; \
      (a)[2] = constant*(b)[2];

// Resta de dos vectores: a = b - c
#define subtraction(a,b,c) \
      (a)[0] = (b)[0] - (c)[0]; \
      (a)[1] = (b)[1] - (c)[1]; \
      (a)[2] = (b)[2] - (c)[2];

///// Fin de Macros para vectores de tres componentes /////




///////   Funciones auxiliares para matrices  ///////

// Multiplica dos matrices:  a1[Y][X] * a2[X][Z] = a3[Y][Z]
void multYXZ(float *a1, float *a2, float *a3, int Y, int X, int Z); //a1*a2=a3


/******************    FIN DE LIBRERIA DE CÁLCULO MATRICIAL                 ***********
```

```
68: /******************    RUTINA PARA CALCULAR EL MINIMO DE UNA FUNCION   *****************
69:
70: // Golden Search: Búsqueda del mínimo de f a partir de la distancia aúrea
71: float brent(float ax, float bx, float cx, float (*f)(float), float tol, float *xmin);
72:
73: //Macros necesarias para brent:
74: #include "nrutil.h"
75: #define ITMAX 100        // Máximo número de iteraciones
76: #define CGOLD 0.3819660  // Distancia aúera
77: #define ZEPS 1.0e-10     // Precisión (cero)
78:
79: #define SHFT(a,b,c,d) (a)=(b);(b)=(c);(c)=(d);
80: /*******************************************************************************
81:
82:
83:
84:
85: /******************    VARIABLES GLOBALES      ***************************************
86: float pi=3.1415926;
87: int i,j,t,k,u,s;  // Variables contador
88: int leidas; // Numero de moleculas leidas
89:
90: // int v,a,b;
91:
92: int N;                    // Número de átomos de la molécula
93: int max;                  // Numero maximo de moleculas a procesar
94: int long_max_nombre=212;  // Maximo numero de caracteres en el nombre
95: int size_s;               // Numero de posibles valores de s
96:
97: int *S;                   // Vector con los valores de S
98: char *nombres;            // Nombres de las moléculas
99: float *fractional;        // Coordenadas fraccionarias
100: float *cartesian;         // Coordenadas cartesianas
101: float *intrinsic;         // Intrínsecas respecto de S' (sin escalar)
102: float *If;                // Intrinsecas del fragmento a escala
103:
104: float *p_If1;         // Apunta a la primera molecula de la pareja cuya distancia se
105:                       // calculara a traves de la funcion d_gamma
106:
107: ////////////////// Variables globales para la función cambio/////////////////////////
108: float *D;                                                                          //
109: float M[3][3];                                                                    //
110: float J[3][3];                                                                    //
111:                                                                                  //
112: float *mult_temp1;    // If2*M o D*If2 dependiendo de los valores de {v,a,b}      //
113: float *mult_temp2;    // D*If2*M*J                                                //
114: float *Ifpre_inicial; // D^v * If2 * M^a * J^b                                   //
115:                                                                                  //
116: float *If2cond;    // T^u * D^v * If2 * M^a * J^b                               //
117: ////////////////////////////////////////////////////////////////////////////////////
118:
119: float *If2cond_gamma; // (T^u * D^v * If2 * M^a * J^b)*R(gamma)
120:
121:
122: /* Valores de mínimos que sólo se utilizarán en la versión del programa para comprobar la
123: matriz de distancias, donde se imprimen los valores de u,v,a,b asociados al mínimo.
124: Poner estas variables dentro de la funcion distancia_d en la version final del programa *
125: int umin,vmin,amin,bmin;         //Valores de [u,v,a,b] que minimizan d*
126: float dmin, gamma_min;
127:
128: /******************    FIN DE VARIABLES GLOBALES     ********************************
129:
130:
131:
132: /******************    FUNCIONES AUXILIARES  ***************************************
133: void leer_archivo(); //Lee el fichero de origen y crea la matriz If
134:
```

```
135:      // Halla las coordenadas cartesianas
136:      void fractian2cartesian(float p_celda[], float *fractional);
137:
138:      // Cambia el origen de coordenadas al centro geométrico de la molécula
139:      void centro_geometrico(float *cartesian);
140:
141:      // Pasa de coordenadas cartesianas a intrinsecas
142:      void cartesian2intrinsic(float *cartesian,int k);
143:
144:      // Divide las coordenadas intrinsecas entre la distancia media interatómica
145:      void escalar_molecula(float *in,int k);
146:
147: void escribir_archivo();
148:        float distancia_d(float *If1, float *If2);
149:              void cambio(int u, int v, int a, int b, float *If2, float *If2_cambiado);
150:              float d_gamma(float gamma); //Calcula d
151:                    void giro(float *m, float gamma, float *m_girada);
152:                    float norma(float *a,float *b);
153:
154: /******************    FIN DE FUNCIONES AUXILIARES  **********************************
155:
156:
157: int main(int argc, char *argv[])
158: {
159: system("CLS");
160: printf("################################################################################\n");
161: printf("#                                                                              #\n");
162: printf("#           Implementacion de un algoritmo de superposicion 3D                 #\n");
163: printf("#         de fragmentos químicos asociados a moléculas en forma de anillo      #\n");
164: printf("#                                                                              #\n");
165: printf("#                                                                              #\n");
166: printf("#           Escuela Tecnica Superior de Ingenieros Industriales                #\n");
167: printf("#            Universidad Politecnica de Cartagena. Febrero 2010.               #\n");
168: printf("#                                                                              #\n");
169: printf("################################################################################\n");
170:
171: /* Lo primero que debe hacer el usuario es introducir los parámetros que va a utilizar
172:    el programa */
173:
174: printf("\n\nIntroduzca por favor los parametros a utilizar:\n\n\n");
175:
176: printf("Numero de atomos de las moleculas: ");
177: scanf("%d",&N);
178:
179: printf("\n\nMaximo numero de moleculas a procesar: ",max);
180: scanf("%d",&max);
181:
182: printf("\n\nNumero de valores posibles de posicion inicial s:  ");
183: scanf("%d",&size_s);
184:
185: S=(int *) malloc(sizeof(int)*size_s);  /* Asignar al puntero S una direccion de memoria
186:                                            donde guardara las posibles posiciones s */
187:
188: printf("\n1er valor posible de s: ");
189: scanf("%d",S);
190: for(i=2;i<=size_s;i++){
191:        printf("%do valor posible de s: ",i);
192:        scanf("%d",S+i-1);}
193:
194: /* El usuario ya ha definido todos los parámetros del programa */
195:
196:
197:
198:
199: // Definición de las variables globales que se utilizan en cambio() ////////////////////
200:
201: D=(float *)malloc(sizeof(float)*N*N);  // Asignar al puntero D una direccion de memoria
```

```
202:
203: for(i=0; i<N; ++i)           //  Definición de D
204:         for(j=0; j<N; ++j)    //
205:                 *(D+i*N+j)=0;  //
206: int D00=1;                    //
207: *D=D00;                       //
208: for(i=1; i<N; ++i)           //
209:         *(D+(N-i)*N+i)=1;     //
210:
211: for(i=0; i<3; ++i)                          //  Definición de M
212:         for(j=0; j<3; ++j)                  //
213:                 if((i==j)&(i!=3)) M[i][j]=1;   //
214:                 else M[i][j]=0;             //
215: M[2][2]=-1;                                 //
216:
217: for(i=0; i<3; ++i)          //  Definición de J
218:         for(j=0; j<3; ++j)   //
219:                 J[i][j]=0;    //
220: J[0][1]=J[1][0]=1;          //
221: J[2][2]=-1;                 //
222:
223: mult_temp1=(float *) malloc(sizeof(float)*N*3); // If2*M o D*If2
224: mult_temp2=(float *) malloc(sizeof(float)*N*3); // D*If2*M
225:
226: // D^v * If2 * M^a * J^b
227: Ifpre_inicial=(float *) malloc(sizeof(float)*N*N);
228:
229: // (T^u * D^v * If2 * M^a * J^b)*R(gamma)
230: If2cond_gamma=(float *) malloc(sizeof(float)*N*3);
231:
232: // T^u * D^v * If2 * M^a * J^b
233: If2cond=(float *) malloc(sizeof(float)*N*3);
234:
235: // Fin de Definición de las variables globales que se utilizan en cambio() //////////////
236:
237:
238: /* Ahora el programa leera el archivo de origen de los datos del programa */
239: leer_archivo();
240:
241: printf("Moleculas leidas del archivo: %d\n\n",(leidas));
242:
243: /* Y por último, a partir de los datos leídos, calculará la matriz y la guardará en otro
244:    archivo de texto. Hay dos posibilidades, con o sin los valores: s,a,b,v,gamma */
245:
246: escribir_archivo();
247:
248: getchar();
249:
250:
251: /* Al finalizar el programa se libera la memoria utilizada */
252:
253: free(S);
254: free(nombres);
255: free(cartesian);
256: free(intrinsic);
257: free(If);
258: free(fractional);
259:
260: free(D);
261: free(mult_temp1);
262: free(mult_temp2);
263: free(If2cond);
264: free(If2cond_gamma);
265:
266:
267:
268: printf("\n\n\t\t Fin de programa");
```

```
269: getchar();
270:
271:
272: } //Fin de main
273:
274:
275: /******************    FUNCIONES AUXILIARES A MAIN ************************************
276:
277:
278: void leer_archivo()
279: {
280:
281: /* Primero se asigna una dirección en memoria a las matrices que se van a utilizar */
282:
283: nombres=(char *) malloc(sizeof(char)*max*long_max_nombre); // Nombres de las moléculas
284: fractional=(float *) malloc(sizeof(float)*N*3);  // Coordenadas fraccionarias
285: cartesian=(float *) malloc(sizeof(float)*N*3);  // Coordenadas cartesianas
286: intrinsic=(float *) malloc(sizeof(float)*N*3);  // Coordenadas "intrínsecas"
287: If=(float *) malloc(sizeof(float)*N*max*3);     /* Coordenadas "intrínsecas" referidas al
288:                                                    centro geométrico */
289:
290: k=0;  // Molécula que se lee
291: float celda[6];  // Parámetros de celda
292:
293: /* Se debe introducir por pantalla el archivo de origen de los datos */
294: FILE *entrada;
295: char fichero_entrada[241];
296:
297: printf("\n\nFichero de datos de entrada: ");
298: scanf("%s",fichero_entrada);
299:
300: /* Comprobamos que el fichero introducido por el usuario, se encuentra en el mismo
301:    directorio que el programa */
302:
303: while ((entrada=fopen(fichero_entrada,"r"))==NULL){
304:    printf("\nEl archivo no existe. Especifique otro: ");
305:    scanf("%s",fichero_entrada);
306:    }
307:
308: // Una vez comprobado que el fichero de entrada de datos existe, se pasa a leerlo
309:
310: while(!feof(entrada))
311: {
312: if(entrada==NULL)
313:       printf("Error al abrir el fichero");
314:
315: char caracter;       //Variable temporal de lectura de nombre
316:
317: int u=0;
318: caracter='0';
319:
320:
321: while(caracter !=';')                                    //
322: {                                                        //
323: caracter=getc(entrada);                                 //
324: *(nombres+k*long_max_nombre+u)=caracter;                 //
325: u++;                                                     //
326: }                                                        //
327:                                                         // Extrae datos del fichero
328: for(i=0;i<5;++i)                                        // origen, copia el nombre
329:        fscanf(entrada, "%f %*c",&celda[i]);             // y cambia a coordenadas
330: fscanf(entrada, "%f", &celda[5]);                       // cartesianas
331:                                                         //
332: for(i=0;i<N;++i)                                        //
333:        for(j=0;j<3;++j)                                 //
334:               fscanf(entrada, "%*c %f ", fractional+i*3+j);
335:
```

```
336: /* A partir de los parámetros de celda y las coordenadas fraccionarias,
337: hallamos las cartesianas */
338: fractian2cartesian(celda,fractional);
339:
340: // Cambiamos el origen de las coordenadas cartesianas al centro geométrico
341: centro_geometrico(cartesian);
342:
343: // Hallamos las coordenadas "intrínsecas"
344: cartesian2intrinsic(cartesian,k);
345:
346: // Dividimos las coordenadas intrinsecas entre la distancia media interatómica */
347: escalar_molecula(intrinsic,k);
348:
349:
350: k++;
351: leidas=k;
352: }
353:
354: fclose(entrada);
355:
356: }  // Fin de leer_archivo
357:
358:
359:
360: /****  Cambia de coordenadas fraccionarias a cartesianas ****/
361: void fractian2cartesian(float p_celda[], float *fractional)
362: {
363:                                                          // Parámetros  de la
364: float cos_alfa=cos(p_celda[3]/180*pi);                   // matriz de cambio de
365: float cos_beta=cos(p_celda[4]/180*pi);                   //  coordenadas  M
366: float cos_gamma=cos(p_celda[5]/180*pi);                  //
367:                                                          //
368:                                                          //
369: float sin_gamma=sin(p_celda[5]/180*pi);                  //
370:                                                          //
371: float v= sqrt( 1 - pow(cos_alfa,2) - pow(cos_beta,2) -   //
372:       pow(cos_gamma,2) + 2*cos_alfa*cos_beta*cos_gamma); //
373:
374:
375: float M01=p_celda[1]*cos_gamma;                                        //
376: float M02=p_celda[2]*cos_beta;                                         //
377: float M11=p_celda[1]*sin_gamma;                                        // Matriz M
378: float M12=(p_celda[2]*(cos_alfa-cos_beta*cos_gamma))/sin_gamma;       //
379: float M22=p_celda[2]*v/sin_gamma;                                      //
380:
381:
382:
383: //Cambio de coordenadas: M x [x;y;z]
384: for (i=0;i<N;++i)
385:        {
386:        *(cartesian+i*3+0)= p_celda[0]*(*(fractional+i*3+0))+M01*(*(fractional+i*3+1)) +
387:                            M02*(*(fractional+i*3+2));
388:        *(cartesian+i*3+1)= M11*(*(fractional+i*3+1)) + M12*(*(fractional+i*3+2));
389:        *(cartesian+i*3+2)= M22*(*(fractional+i*3+2));
390:        }
391: }
392:
393: /*    Cambia el origen de coordenadas al centro geométrico de la molécula     */
394: void centro_geometrico(float *cartesian)
395: {
396: float x=0,y=0,z=0;
397: float xc, yc, zc;   //Coordenadas del centro de gravedad
398:
399: for(i=0;i<N;++i)     //Cálculo del c.d.g.
400:        {
401:        x=x+(*(cartesian+i*3+0));
402:        y=y+(*(cartesian+i*3+1));
```

```
403:        z=z+(*(cartesian+i*3+2));
404:        }
405: xc=x/N;
406: yc=y/N;
407: zc=z/N;
408:
409: for(i=0;i<N;++i)     //Cambio de origen del coordenadas al c.d.g.
410:        {
411:        (*(cartesian+i*3+0))=(*(cartesian+i*3+0))-xc;
412:        (*(cartesian+i*3+1))=(*(cartesian+i*3+1))-yc;
413:        (*(cartesian+i*3+2))=(*(cartesian+i*3+2))-zc;
414:        }
415: }
416:
417: /*** Pasa de coordenadas cartesianas a intrinsecas  ***/
418: void cartesian2intrinsic(float *cartesian,int k)
419: {
420: float R1_0[3]={0,0,0},R2_0[3]={0,0,0},R2_1[3],R0[3],k1;  // Variables temporales
421: float n[3],R1[3],R2[3];     // Ejes del sistema de coordenadas
422:
423: for(i=0;i<N;++i)
424:        for(j=0;j<3;++j)
425:        {
426:               R1_0[j]=R1_0[j]+(*(cartesian+i*3+j))*sin(2*pi*i/N);
427:               R2_0[j]=R2_0[j]+(*(cartesian+i*3+j))*cos(2*pi*i/N);
428:        }
429:
430:
431: normalize(R1,R1_0);
432: dotProduct(k1,R1,R2_0);
433: scalar(R0,k1,R1);
434: subtraction(R2_1,R2_0,R0);
435: normalize(R2,R2_1);
436:
437: crossProduct(n,R1,R2)
438:
439:
440: float G[3][3]; //Matriz de cambio de coordenadas
441: for(i=0;i<3;++i)
442:        {
443:        G[0][i]=R1[i];
444:        G[1][i]=R2[i];
445:        G[2][i]=n[i];
446:        }
447:
448: for(i=0;i<N;++i)
449:        for(j=0;j<3;++j)
450:               *(intrinsic+i*3+j)=G[j][0]*(*(cartesian+i*3+0))+G[j][1]*(*(cartesian+i*3+1))
451:                                  +G[j][2]*(*(cartesian+i*3+2));
452: }
453:
454:
455: /*     Divide las coordenadas intrinsecas entre la distancia media interatómica */
456: void escalar_molecula(float *in,int k)
457: {
458: float temp;  //Variable temporal
459: float dist=0;     //Distancia temporal entre dos atomos consecutivos
460: float dist_media; //Distancia media entre los N atomos
461:
462: for(i=0;i<(N-1);i++)
463:        {
464:        temp=0;
465:        for(j=0;j<3;j++)
466:               temp=temp+pow((*(in+i*3+j))-(*(in+(i+1)*3+j)),2);
467:        dist=dist+sqrt(temp);
468:        }
469:
```

```
470: temp=0;
471: for(j=0;j<3;j++)
472:         temp=temp+pow((*(in+0*3+j))-(*(in+(N-1)*3+j)),2);
473:
474:
475: dist=dist+sqrt(temp);
476: dist_media=dist/N;
477:
478: for(i=0;i<N;i++)
479:         for(j=0;j<3;j++)
480:                 *(If+k*3*N+i*3+j)=(*(in+i*3+j))/dist_media;
481:
482: }
483: /* Fin de leer_archivo y sus funciones auxiliares */
484:
485:
486:
487: void escribir_archivo()
488: {
489: /* El usuario debe introducir un el nombre del archivo donde quiere guardar la matriz de
490:    distancias. Éste debe terminar en .txt si se quiere generar directamente el archivo de
491:    texto */
492:
493: FILE *salida; // Archivo en el que se escribirá únicamente la matriz de distancias
494:
495: char fichero_salida[228];
496: printf("Fichero de escritura de la matriz de distancias (sin extension): ");
497: scanf("%s",fichero_salida);
498: strcat(fichero_salida,".txt"); // Añade la extension por defecto .txt
499:
500:
501: // Comprobamos que no existe un fichero de salida con el mismo nombre
502: while ((salida=fopen(fichero_salida,"r"))!=NULL){
503:    printf("\nEl archivo ya existe. Especifique otro: ");
504:    scanf("%s",fichero_salida);
505:    strcat(fichero_salida,".txt");
506:    }
507:
508: salida=fopen(fichero_salida,"w");
509:
510:
511: /* El programa genera automaticamente otro archivo, que se llama como el definido por el
512:    usuario, pero añadiendo _detalle.txt al final. Este archivo grabará, además de la
513:    matriz,los valores de [s,a,v,b,gamma] de cada mínimo */
514:
515: FILE *salida_detalle; // Archivo en el que se escribirán los nombres de las moleculas,
516:                       // su distancia mínima y [s,a,v,b,gamma]
517:
518: char fichero_salida_detalle[241];
519: strcpy(fichero_salida_detalle,fichero_salida);
520: strcpy(strstr(fichero_salida_detalle,".txt"),"_detalle.txt");
521:
522: salida_detalle=fopen(fichero_salida_detalle,"w");
523:
524: int m1,m2; // Moleculas que se van a comparar
525: float dist; // d
526:
527: for(m1=0;m1<leidas;m1++){
528:         for(m2=m1+1;m2<leidas;m2++) // Barre todas las moleculas
529:                 {
530:                 dist=distancia_d(If+m1*N*3,If+m2*N*3); // Calcula la distancia entre m1 y m2
531:
532:                 fprintf(salida,"%.5f\n",dist); // Escribe la distancia entre m1 y m2
533:
534:                 /* Se escribe en el fichero de detalle la información complementaria */
535:                 fprintf(salida_detalle,"%d %d:",m1,m2); // Subíndice dentro de la matriz
536:
```

```
537:                t=0; // El nombre de la primera molécula
538:                while(*(nombres+m1*long_max_nombre+t) !=';'){
539:                       fprintf(salida_detalle,"%c",*(nombres+m1*long_max_nombre+t));
540:                       t++;}
541:
542:                fprintf(salida_detalle," ");  // Un espacio intermedio
543:
544:                t=0; // El nombre de la segunda molécula
545:                while(*(nombres+m2*long_max_nombre+t) !=';'){
546:                       fprintf(salida_detalle,"%c",*(nombres+m2*long_max_nombre+t));
547:                       t++;}
548:
549:                fprintf(salida_detalle,";\t"); // Un tabulador intermedio
550:
551:                /* Y por último, los valores de [s,a,v,b,gamma] para la distancia mínima */
552:                fprintf(salida_detalle,"d: %.5f, s=%d, v=%d, a=%d, b=%d, gamma=%.4f\n\n",
553:                dist,umin+1,vmin,amin,bmin,gamma_min);
554:
555:                }
556: printf(".");}  /* Hace que se vayan escribiendo puntos en pantalla mientras
557:                                  corre el programa */
558:
559: } //Fin de escribir_archivo
560:
561:
562: /***   Calcula la distancia d        *********************/
563: float distancia_d(float *If1, float *If2)
564: {
565: int u,v,a,b;     // Variables contador. Nota: u = (s-1)
566:
567: p_If1=If1;       /* Asigno al puntero que se utilizara en d_gamma() el valor del If1
568:                     que paso a la funcion distancia_d() */
569:
570: dmin=10000; /* Punto inicial de iteración. La distancia va decreciendo,
571:                de modo que se empieza por un valor muy alto */
572:
573: float dcond;     // Minima distancia d(gamma) para esas condiciones: u,v,a,b
574:
575: float *p_min_temp;    // Puntero donde escribirá golden el mínimo
576: float min_temp;       // Mínimo de la función devuelto por golden
577: p_min_temp=&min_temp; // Asignación de p_min_temp a una direccion de memoria
578:
579:
580: float (*p_d_gamma)(float gamma);  // Puntero a la funcion d_gamma
581: p_d_gamma=d_gamma; // Asigna al puntero la dirección de la funcion
582:
583:
584: for(u=0; u<size_s; ++u)
585:        for(v=0; v<=1; ++v)
586:               for(a=0; a<=1; ++a)
587:                      for(b=0; b<=1; ++b)
588:                             {
589:                             cambio((*(S+u)-1),v,a,b,If2,If2cond);
590:                             // Escribe en la matriz If2_cond: T^(s-1)*D^v*If2*M^a*J^b
591:                             dcond=brent(-0.0001,2.4,6.2833,p_d_gamma,0.000001,p_min_temp);
592:
593:                             if(dcond<dmin)
594:                                    {
595:                                    dmin=dcond;
596:                                    umin=(*(S+u)-1);
597:                                    vmin=v;
598:                                    amin=a;
599:                                    bmin=b;
600:                                    gamma_min=min_temp;
601:                                    }
602:                             }
603: return(dmin);
```

```
604: }
605:
606:
607: /*** Reorienta la molécula según [u,v,a,b]*********************/
608: /*** If2 => If2cond  ***/
609:
610: /*  Según los valores {u,v,a,b}, toma la molecula apuntada por If2, calcula
611:     T^u*D^v*If2*M^a*J^b, y el resultado lo guarda en la variable gloabl If2cond,
612:     cuyo primer elemento está apuntado por If2_cambiado */
613: void cambio(int u, int v, int a, int b, float *If2, float *If2_cambiado)
614: {
615:
616: float *Ifpre=Ifpre_inicial;  // Se asigna a Ifpre una dirección en memoria
617:
618: // Cálculo de Ifpre = D^v * If2 * M^a * J^b
619: if (v==0)
620:        if (a==0)
621:               if (b==0)
622:                       Ifpre=If2;
623:
624:               else // b=1
625:                       multYXZ(If2,&J[0][0],Ifpre,N,3,3);
626:        else // a=1
627:               if (b==0)
628:                       multYXZ(If2,&M[0][0],Ifpre,N,3,3);
629:
630:               else{ // b=1
631:                       multYXZ(If2,&M[0][0],mult_temp1,N,3,3);
632:                       multYXZ(mult_temp1,&J[0][0],Ifpre,N,3,3);}
633: else // v=1
634:      if (a==0)
635:               if (b==0)
636:                       multYXZ(D,If2,Ifpre,N,N,3);
637:
638:               else{// b=1
639:                       multYXZ(D,If2,mult_temp1,N,N,3);
640:                       multYXZ(mult_temp1,&J[0][0],Ifpre,N,3,3);}
641:        else // a=1
642:               if (b==0){
643:                       multYXZ(D,If2,mult_temp1,N,N,3);
644:                       multYXZ(mult_temp1,&M[0][0],Ifpre,N,3,3);}
645:
646:               else{ // b=1
647:                       multYXZ(D,If2,mult_temp1,N,N,3);
648:                       multYXZ(mult_temp1,&M[0][0],mult_temp2,N,3,3);
649:                       multYXZ(mult_temp2,&J[0][0],Ifpre,N,3,3);}
650: // Fin de cálculo de Ifpre = D^v * If2 * M^a * J^b
651:
652: if (u==0)
653:        If2cond=Ifpre;
654: else{
655:        float Tcond[N][N];                  //  Definición de T^u
656:        for(i=0; i<N; ++i)                  //
657:                 for(j=0; j<N; ++j)         //
658:                          Tcond[i][j]=0;    //
659:                 for(j=0; j<N-u; ++j)       //
660:                          Tcond[j][j+u]=1;  //
661:        for(i=0; i<u; ++i)                  //
662:                 Tcond[N-u+i][i]=1;         //
663:
664:        multYXZ(&Tcond[0][0],Ifpre,If2_cambiado,N,N,3);
665:      }
666:
667: }// Fin de cambio
668:
669:
670: /* Para las moleculas apuntadas por p_If1, If2cond, calcula su distancia después de girar
```

```
671:    If2_cond, es decir: || If1-If2_cond*R(gamma) || */
672: float d_gamma(float gamma)
673: {
674:
675: giro(If2cond,gamma,If2cond_gamma);
676: float distancia=norma(p_If1,If2cond_gamma);
677: return(distancia);
678: }
679:
680: /***  Gira la matriz m un ángulo gamma y lo guarda en m_girada    ***/
681: void giro(float *m, float gamma, float *m_girada)
682: {
683:
684: float matriz_giro[3][3]={
685:        cos(gamma),-sin(gamma),0,
686:        sin(gamma),cos(gamma), 0,
687:        0,         0,          1};
688:
689: multYXZ(m,&matriz_giro[0][0],m_girada,N,3,3);
690: }
691: /* Fin de escribir_archivo y sus funciones auxiliares */
692:
693: /***  Norma de la resta de las matrices: a - b     ***/
694: float norma(float *a,float *b)
695: {
696:
697: float suma2;     // Suma de (a[i][j]-b[i][j])^2 para un i fijo, de las tres componentes
698: float norma_i=0;   // Norma del átomo [i]
699:
700: for(i=0; i<N; ++i)
701:        {
702:        suma2=0;
703:        for(j=0; j<3; ++j)
704:               suma2=suma2+pow((*(a+i*3+j)-*(b+i*3+j)),2);
705:        norma_i=norma_i+pow(suma2,0.5);
706:        }
707: float norma_resta=norma_i/N;
708:
709: return(norma_resta);
710: }
711:
712:
713:
714:
715:
716: /*  Definicion de Brent: Búsqueda del mínimo de f a partir de la distancia aúrea */
717:
718: float brent(float ax, float bx, float cx, float (*f)(float), float tol, float *xmin)
719:
720: /* Dada una función f, y tres puntos de inicio: ax<bx<cx, esta rutina encuentra el mínimo
721: con una precisión aproximada de tol, usando el método de Brent (interpolación parabólica)
722: La abscisa del mínimo es xmin y el valor mínimo es el float devuelto por la funcion */
723:
724:
725: {
726: int iter;
727: float a,b,d,etemp,fu,fv,fw,fx,p,q,r,tol1,tol2,u,v,w,x,xm;
728: float e=0.0;
729: a=(ax < cx ? ax : cx);
730: b=(ax > cx ? ax : cx);
731: x=w=v=bx;
732: fw=fv=fx=(*f)(x);
733: for (iter=1;iter<=ITMAX;iter++)
734:     {
735:     xm=0.5*(a+b);
736:     tol2=2.0*(tol1=tol*fabs(x)+ZEPS);
737:
```

```
738:     if (fabs(x-xm) <= (tol2-0.5*(b-a))) {
739:        *xmin=x;
740:        return fx;}
741:
742:     if (fabs(e) > tol1) {
743:        r=(x-w)*(fx-fv);
744:        q=(x-v)*(fx-fw);
745:        p=(x-v)*q-(x-w)*r;
746:        q=2.0*(q-r);
747:        if (q > 0.0) p = -p;
748:        q=fabs(q);
749:        etemp=e;
750:        e=d;
751:        if (fabs(p) >= fabs(0.5*q*etemp) || p <= q*(a-x) || p >= q*(b-x))
752:        d=CGOLD*(e=(x >= xm ? a-x : b-x));
753:        else {
754:             d=p/q;
755:             u=x+d;
756:             if (u-a < tol2 || b-u < tol2)
757:                d=SIGN(tol1,xm-x);
758:             }
759: } else {
760:   d=CGOLD*(e=(x >= xm ? a-x : b-x));
761: }
762: u=(fabs(d) >= tol1 ? x+d : x+SIGN(tol1,d));
763: fu=(*f)(u);
764: if (fu <= fx) {
765:    if (u >= x) a=x; else b=x;
766:    SHFT(v,w,x,u)
767:    SHFT(fv,fw,fx,fu)
768: } else {
769:        if (u < x) a=u; else b=u;
770:        if (fu <= fw || w == x) {
771:           v=w;
772:           w=u;
773:           fv=fw;
774:           fw=fu;
775: } else if (fu <= fv || v == x || v == w) {
776:        v=u;
777:        fv=fu;
778: }
779: }
780: }
781: printf("Too many iterations in brent");
782: }
783: /*  Fin de Brent */
784:
785:
786:
787: /******************    FUNCIONES AUXILIARES PARA MATRICES   ***************************
788:
789: /**** Multiplica dos matrices:  a1[Y][X] * a2[X][Z] = a3[Y][Z] =  ****/
790: void multYXZ(float *a1, float *a2, float *a3, int Y, int X, int Z) //a1*a2=a3
791: {
792: int i,j,k;
793: float temp;
794:        for(i=0; i<Y; i++) //Filas de la izquierda
795:               for(j=0; j<Z; j++) //Columnas de la derecha
796:                      {temp=0;
797:                      for(k=0; k<X; k++) //Columnas de la izq=Filas de la derecha
798:                             temp+=(*(a1+i*X+k))*(*(a2+k*Z+j));
799:                      *(a3+i*Z+j)=temp;
800:                      }
801: }
802:
803:
804:
```