Projecto de Física Computacional

Monografia

Cálculo das energias, estruturas e distâncias interatómicas de equilíbrio para alguns agregados de carbono, usando potenciais de tipo Tight-Binding, com base na teoria do Funcional da Densidade

Lara Vaz Pato nº34578
Pedro Cruz nº26920
Rui Caldeira nº32046

FCUL · Julho 2009

"Nunca desencoraje aquele que continuamente progride, não importa quão lentamente."

Platão (427AC - 347AC)

Conteúdo

1 Introdução

1.1 Tight-Binding e métodos de cálculo de energia

Há duas aproximações possíveis para o cálculo da energia total de agregados: uma é a via mais clássica e empírica, sendo muito rápida para estruturas bem conhecidas, mas falhando frequentemente para estruturas ausentes da sua base de dados (tem pouca "transferabilidade"); por outro lado, há cálculos rigorosos baseados nas teorias do funcional de densidade (DFT) e de *Hartree-Fock*, mas estes são um pouco lentos para a investigação de certos problemas.

Para tentar aproveitar o melhor dos dois mundos surgiram os métodos semi-empíricos, entre os quais o bem sucedido *Tight-Binding* (TB). Nos modelos semi-empíricos fazem-se em geral as seguintes aproximações:

- Só são considerados os electrões de valência;
- Tratam-se apenas integrais sobre duas funções de onda de cada vez (integrais de dois centros);
- Os elementos de matriz (hamiltoniana e de *overlap*) são aproximados por funções analíticas da separação interatómica, sendo os seus parâmetros escolhidos para reproduzir as características de sistemas de referência;
- A repulsão coulombiana núcleo-núcleo é substituída por uma função parametrizada.

Quanto à aproximação de TB ela é utilizada por exemplo em física do estado sólido para calcular a estrutura de bandas electrónica usando um conjunto de funções de onda aproximadas, baseadas na sobrepo-sição de funções de onda para átomos isolados localizados em cada posição atómica. O método está fortemente ligado ao LCAO usado para obter as orbitais moleculares.

O TB é especialmente eficaz para a optimização computacional. Queremos um modelo que, além de bem motivado fisicamente, seja o mais simples e com o menor número de parâmetros possível, mas suficientemente rigoroso para fornecer uma descrição qualitativa (ou até quantitativa) do sistema.

No artigo em que nos baseámos [1], os autores usam o TB não ortogonal (ou seja, que não despreza o efeito de *overlap* das orbitais), tendo atingido um compromisso entre um cálculo eficiente e suficientemente correcto para os fins propostos. Autodenominaram o seu método de *nonorthogonal DF-based TB scheme* (DF-TB) e usaram-no para achar estruturas (geometrias) e energias de equilíbrio de moléculas de carbono e hidrocarbonetos, entre outras características que não testámos.

Parametrizaram o problema de forma a ficar o mais próximo possível dos valores teóricos, baseando-se na DFT – aproximação de densidade local (LDA). Depois transpuseram o método para sistemas mais complexos, como o C_{60}, argumentando que o método é "transferível" para sistemas mais complexos de carbono.

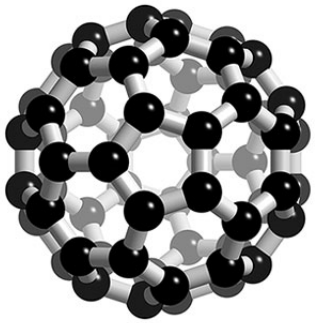

Figura 1.1: Fulereno C_{60} (imagem retirada de [4]).

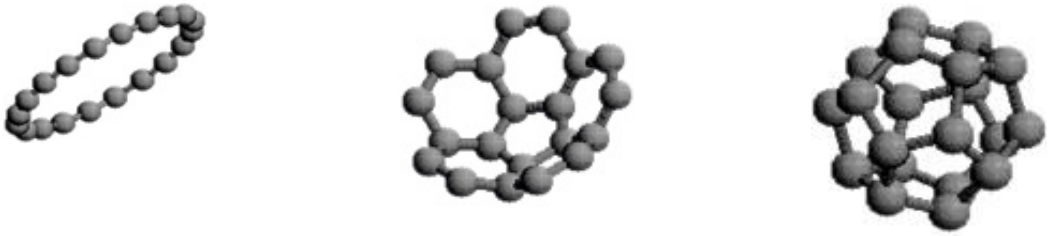

Figura 1.2: C_{20} em anel, taça e gaiola (imagem retirada de [5]).

1.2 Os Fulerenos

Os fulerenos são estruturas sólidas formadas por carbono. Os seus anéis pentagonais (ou heptagonais), permitem que a sua estrutura não seja plana.

O fulereno mais conhecido é o C_{60} (Figura 1.1), descoberto numa experiência em 1985 pelos professores Kroto, Curl e Smallery. Os seus átomos encontram-se nos vértices de um icosaedro truncado, com 12 pentágonos e 20 hexágonos. Os hexágonos são não regulares, visto que os lados que estão em contacto com pentágonos são maiores (ligação simples) do que os lados em contacto com hexágonos (ligação dupla).

O mais pequeno fulereno previsto pelos actuais modelos é o C_{20}, uma estrutura de 12 pentágonos, formando um dodecaedro. Apesar de ser teoricamente possível, é difícil produzir esta molécula, visto que a grande curvatura da sua superfície dá-lhe uma maior tendência para se abrir, e é muito reactiva. Há fontes que dizem que esta molécula já foi produzida [5], no entanto nunca numa forma estável. Para além desta estrutura (*cage*), também são previstas outras duas estruturas de equilíbrio para o C_{20}: em anel e em taça (Figura 1.2); diferentes modelos dão previsões distintas para a estrutura do estado fundamental do C_{20}, mas isso é algo que ainda não foi definitivamente estabelecido pela pela experiência.

2 Modelo e Implementação

O que o modelo apresentado no artigo [1] nos dá é uma forma de calcular a energia total da configuração de átomos, que depois utilizaremos para achar a estrutura de equilíbrio através do *simulated annealing*. Nesta secção é descrito esse modelo e a forma como foi implementado no nosso programa.

2.1 Elementos de Matriz

Vamos precisar das matrizes **H** e **S** (hamiltoniana e de sobreposição/*overlap* respectivamente), cujos elementos são dados por

$$\mathbf{S}_{\mu\nu} = \langle\phi_\mu|\phi_\nu\rangle \qquad \text{e} \qquad \mathbf{H}_{\mu\nu} = \langle\phi_\mu|\mathbf{H}|\phi_\nu\rangle, \tag{2.1}$$

em que ϕ_μ e ϕ_ν são as funções de onda correspondentes a duas orbitais electrónicas (uma corresponde à linha e outra à coluna). Em cada átomo de carbono há uma orbital de tipo **s** e três de tipo **p** (as orbitais de mais baixa energia) por onde se podem distribuir os quatro electrões de valência. Assim, em cada linha e coluna das matrizes, cada uma das quatro orbitais de cada átomo, ficando uma matriz $4\mathbf{N}\times4\mathbf{N}$, sendo **N** o número de átomos do agregado.

Para as interacções entre electrões no mesmo átomo, as expressões (2.1) são simples:

$$\mathbf{S}_{\mu\nu} = \int \phi_\mu\phi_\nu = \begin{cases} 1 & \mu = \nu \\ 0 & \mu \neq \nu \end{cases}, \tag{2.2}$$

$$\mathbf{H}_{\mu\nu} = \int \phi_\mu\mathbf{H}\phi_\nu = \begin{cases} \varepsilon_s & \mu = \nu = s \\ \varepsilon_p & \mu = \nu = p \\ 0 & \mu \neq \nu \end{cases}, \tag{2.3}$$

com ε_s e ε_p as energias das orbitais **s** e **p** no átomo livre.

No artigo dão-nos a expressão

$$V(\mathbf{r}) = \sum_{m=1}^{10} c_m T_{m-1}(y) - \frac{c_1}{2}, \qquad y = \frac{r - \frac{b+a}{2}}{\frac{b-a}{2}} \tag{2.4}$$

para calcular o potencial repulsivo entre dois corpos $V_{rep}(r)$ e os elementos $\mathbf{H}_{ss\sigma}(r)$, $\mathbf{H}_{sp\sigma}(r)$, $\mathbf{H}_{pp\sigma}(r)$, $\mathbf{H}_{pp\pi}(r)$, $\mathbf{S}_{ss\sigma}(r)$, $\mathbf{S}_{sp\sigma}(r)$, $\mathbf{S}_{pp\sigma}(r)$ e $\mathbf{S}_{pp\pi}(r)$ em função da distância r entre os átomos. $T_{m-1}(y)$ são os polinómios de Chebyshev, e os coeficientes para cada elemento a calcular, bem como os limites de validade da expressão, são dados em tabelas no artigo - ver Tabela 2 (página 20).

Depois passamos os elementos $\mathbf{X}_{ss\sigma}$, $\mathbf{X}_{sp\sigma}$, etc., em que **X** representa **H** ou **S**, para a base $\{s,p_x,p_y,p_z\}$ (em que estão os elementos das matrizes), através da tabela de Slater-Koster:

$$\begin{aligned} &\mathbf{X}_{ss} = \mathbf{X}_{ss\sigma}, \\ &\mathbf{X}_{sp_i} = c_i\mathbf{X}_{sp\sigma}, \\ &\mathbf{X}_{p_i,p_i} = c_i^2\mathbf{X}_{pp\sigma} + (1-c_i^2)\mathbf{X}_{pp\pi}, \\ &\mathbf{X}_{p_i,p_j} = c_ic_j(\mathbf{X}_{pp\sigma} - \mathbf{X}_{pp\pi}) \qquad (i \neq j), \end{aligned} \tag{2.5}$$

onde $x_1 = x$, $x_2 = y$, $x_3 = z$ e os cossenos directores c_i são dados por $c_i = \frac{x_{ib}-x_{ia}}{r_{ab}}$, com a e b os índices de cada átomo.

Além disso, como **H** e **S** representam observáveis, então terão de ser hermíticas, e como são também reais isso implica que serão simétricas, e como tal tivemos de ter em conta que $\mathbf{X}_{sp\sigma} = -\mathbf{X}_{ps\sigma}$.

2.2 Energia de Coesão do átomo

A primeira equação que surge no TB está relacionada com o cálculo da energia de Estrutura de Bandas (E_{BS}), e no caso não ortogonal consiste em:

$$\sum_{\nu} \mathbf{C}_{\nu i}(\mathbf{H}_{\mu\nu} - \varepsilon_i \mathbf{S}_{\mu\nu}) = 0, \qquad \forall \mu, i, \tag{2.6}$$

com ε_i a energia correspondente a uma orbital ψ_i passível de ser ocupada.

Esta equação é equivalente ao problema de valores próprios generalizado:

$$\mathbf{HC}_i - \varepsilon_i \mathbf{SC}_i = 0, \tag{2.7}$$

com ε_i e C_i os valores e vectores próprios (demonstração no final da secção).

Após resolver este problema ficamos com 4**N** valores próprios ε_i. Mas uma vez que em cada órbita ficam dois electrões e cada carbono tem quatro electrões de valência, queremos apenas as 2**N** energias mais baixas para usar na energia de coesão. Assim, a energia que vai manter os átomos juntos é a energia de estrutura de bandas E_{BS}, dada por:

$$E_{BS} = \sum_{i=1}^{2N} 2\varepsilon_i. \tag{2.8}$$

A energia potencial repulsiva, por sua vez, vem:

$$E_{rep} = \sum_{k=0}^{N-2} \sum_{l=k+1}^{N-1} V_{rep}(r_{kl}). \tag{2.9}$$

A energia total de coesão (*binding energy*) de uma dada configuração é a energia total menos a energia dos átomos livres, em que temos em cada átomo dois electrões em orbital **s** e dois em orbital **p** (configuração de menor energia):

$$E_{bind}(\overrightarrow{R}_1, \overrightarrow{R}_2, ...) = \sum_{i=1}^{2N} 2\varepsilon_i + \sum_{k=0}^{N-2} \sum_{l=k+1}^{N-1} V_{rep}(r_k l) - 2N(\varepsilon_s - \varepsilon_p). \tag{2.10}$$

A energia de ligação por átomo E_{bind}^{at} é este valor dividido por **N**. No artigo é feita ainda outra correcção que consiste em subtrair a este valor a energia de spin-polarização do átomo livre $-1.13eV$.

Nota: Demonstração da equivalência entre (2.6) e (2.7):
Partindo de (2.7) temos

$$\mathbf{HC}_i = \varepsilon_i \mathbf{SC}_i \quad \Leftrightarrow \quad \begin{pmatrix} \mathbf{H}_{11} & \mathbf{H}_{12} & \cdots \\ \mathbf{H}_{21} & \mathbf{H}_{22} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix} \begin{pmatrix} \mathbf{C}_1^i \\ \mathbf{C}_2^i \\ \vdots \end{pmatrix} = \varepsilon_i \begin{pmatrix} \mathbf{S}_{11} & \mathbf{S}_{12} & \cdots \\ \mathbf{S}_{21} & \mathbf{S}_{22} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix} \begin{pmatrix} \mathbf{C}_1^i \\ \mathbf{C}_2^i \\ \vdots \end{pmatrix} \Leftrightarrow$$

$$\Leftrightarrow \quad \begin{pmatrix} \mathbf{H}_{11}\mathbf{C}_1^i + \mathbf{H}_{12}\mathbf{C}_2^i \cdots \\ \mathbf{H}_{21}\mathbf{C}_1^i + \mathbf{H}_{22}\mathbf{C}_2^i \cdots \\ \vdots \\ \vdots \end{pmatrix} = \varepsilon_i \begin{pmatrix} \mathbf{S}_{11}\mathbf{C}_1^i + \mathbf{S}_{12}\mathbf{C}_2^i \cdots \\ \mathbf{S}_{21}\mathbf{C}_1^i + \mathbf{S}_{22}\mathbf{C}_2^i \cdots \\ \vdots \\ \vdots \end{pmatrix}$$

Para isto ser verdade, cada componente μ (cada elemento da coluna) dos dois vectores tem de se igualar

$$\mathbf{H}_{\mu 1}\mathbf{C}_1^i + \mathbf{H}_{\mu 2}\mathbf{C}_2^i \ldots = \varepsilon_i(\mathbf{S}_{\mu 1}\mathbf{C}_1^i + \mathbf{H}_{\mu 2}\mathbf{C}_2^i \ldots) \Leftrightarrow \sum_\nu \mathbf{H}_{\mu\nu}\mathbf{C}_\nu^i = \sum_\nu \varepsilon_i \mathbf{S}_{\mu\nu}\mathbf{C}_\nu^i, \qquad (2.11)$$

o que é claramente igual à equação (2.6), como queríamos demonstrar.

2.3 Simulated Annealing

O *Simulated Annealing* (SA) é um método de optimização fortissimo, que de um modo geral e com os parâmetros certos converge para a solução ideal, o que no nosso caso é o mínimo global da energia. A sua grande vantagem relativamente a outros métodos é que é muito mais difícil que o sistema fique preso num mínimo local.

Basicamente o SA é uma pequena mas decisiva variação do método de Metropolis Monte-Carlo. No nosso problema, um passeio de Metropolis Monte-Carlo tem N passos, cada um consistindo na alteração das coordenadas de um dos átomos do agregado aleatoriamente e comparação da energia do novo agregado (E_2) com a do anterior (E_1). Se E_2 for menor, então o sistema adopta a nova configuração e continua o seu "passeio". Caso contrário, então em princípio deveria rejeitar a nova configuração, visto que é menos estável (algoritmo *greedy*), mas isto tornaria dificílimo ao sistema sair de um mínimo local.

Para dar a volta ao problema, queremos que haja uma certa probabilidade de o programa tomar uma "má decisão", ou seja, escolher E_2 mesmo quando a sua energia é superior, mas claro que se a energia for exorbitantemente maior não faz muito sentido fazê-lo. Para modelar isto consideramos que o sistema se encontra a uma temperatura constante T (*ensemble* canónico), e como tal sabemos que a probabilidade de uma configuração com energia E ocorrer é proporcional ao factor de Boltzmann $e^{-\frac{E}{K_B T}}$, e como tal quanto maior a energia menor a probabilidade. A razão entre as probabilidades de o sistema se encontrar em E_2 ou E_1 estará entre 0 e 1, pelo que geramos um número aleatório entre 0 e 1 e comparamos os dois valores. Aceitamos a nova configuração, ou seja, tomamos uma "má decisão", se $e^{-\frac{E_2-E_1}{K_B T}}$ for inferior a esse valor, o que é mais provável se a nova energia não for muito superior e/ou se a temperatura for muito elevada.

O SA consiste num "passeio de passeios"de Metropolis Monte-Carlo, cada um a uma temperatura T constante, e em cada passo a temperatura desce um pouco. O que isto vai fazer é permitir inicialmente ao sistema explorar bem o espaço de fases, visto que praticamente qualquer configuração poderá ser aceite (o sistema está no estado líquido). À medida que a temperatura diminui, o agregado vai ficando cada vez mais "preso", até solidificar completamente no estado de mais baixa energia.

Os parâmetros mais determinantes para que o SA resulte são a razão entre duas temperaturas consecutivas α, a temperatura inicial T_{max}, geralmente tomada como superior à temperatura de fusão, a temperatura final T_{min} e o número de passos de Monte-Carlo (quantos mais melhor). Para obter os parâmetros do SA pretendidos foram necessárias bastantes experiências, e geralmente os parâmetros eram específicos para cada agregado. Em geral considerámos $\alpha = 0.995$, $T_{max} = 500K$, $T_{min} = 0.0001K$ e $\mathbf{N}\times 1000$ passos de Monte-Carlo, no entanto por exemplo para as estruturas mais pequenas os valores de α podiam perfeitamente ser inferiores que a precisão não se alterava.

2.4 Esquema do Programa

O esquema base de todas as versões do nosso programa foi sempre o mesmo:

- Tem-se um conjunto de **N** átomos de carbono, numa configuração inicial aleatória, com uma certa geometria à escolha (numa caixa ou numa esfera, por exemplo);
- Constroem-se os polinómios $\mathbf{H}^{CC}_{abc}(r)$, $\mathbf{S}^{CC}_{abc}(r)$, $V_{rep}(r)$ de acordo com a equação (2.4) e os coeficientes dados nas Tabelas 2 e 3, e tendo também em conta que $\mathbf{X}_{sp\sigma} = -\mathbf{X}_{ps\sigma}$.
- Passa-se de $\mathbf{H}^{CC}_{abc}(r)$ e $\mathbf{S}^{CC}_{abc}(r)$ para $\mathbf{H}^{CC}_{a'b'}(r)$ e $\mathbf{S}^{CC}_{a'b'}(r)$, de acordo com as fórmulas de Slater-Koster (2.5).
- Acham-se as matrizes **H**, **S**:
  - Os termos da diagonal $(i = j)$ são dados por (2.2) e (2.3), onde usámos os valores para as energias das orbitais **s** e **p** no átomo livre obtidas no cálculo *all-electron LDA PZ*, em hartrees (E_h):
    - $\varepsilon_s = -0.50097 E_h$
    - $\varepsilon_{px} = \varepsilon_{py} = \varepsilon_{pz} = -0.19930 E_h$
  - Os termos fora da diagonal para cada par de orbitais no mesmo átomo são nulos (pois funções de onda de orbitais distintas num mesmo átomo são ortogonais);
  - Os termos fora da diagonal para as restantes orbitais são dados por $\mathbf{H}^{CC}_{ia'jb'}(r_{ij})$ e $\mathbf{S}^{CC}_{ia'jb'}(r_{ij})$, com r_{ij} a distância entre os átomos i e j, a' as orbitais do átomo i e b' as orbitais do átomo j. Se r_{ij} estiver fora dos limites de validade (a, b) da Tabela 2 então o termo é nulo também.
- Com as matrizes construídas resolve-se o problema aos valores próprios generalizado, descrito na Secção 2.2, com a ajuda da secção *Real Generalized Nonsymmetric Eigensystems* da GSL, obtendo as 4N energias ε_i. Ordenam-se estes valores por ordem crescente (tendo considerado as energias como negativas) e escolhem-se os primeiros 2N valores para substituir em (2.8).
- Para o cálculo da energia repulsiva através da expressão 2.9 calcula-se $V_{rep}(r_{ij})$, se r_{ij} se encontrar dentro dos limites (a, b) da Tabela 3. Se a distância for superior considera-se a repulsão como sendo nula e se for inferior como sendo suficientemente grande para que essa configuração seja praticamente proibida ao sistema.
- Acha-se a energia total de coesão pela equação (2.10).
- Usa-se o método de SA, explicitado na secção 2.3, para achar a configuração óptima (de menor energia), partindo de uma temperatura suficientemente elevada e fazendo-a descer lentamente. Em cada passo de Monte-Carlo dentro do SA faz-se o cálculo da energia da forma descrita acima.

Por vezes foram feitas restrições à geometria do sistema (ver secções 3 e A.3), através de uma condição que fazia com que a energia repulsiva fosse muito elevada, levando à quase certa rejeição de uma estrutura fora dos limites impostos.

Em anexo encontra-se uma das versões do nosso programa, especificamente para o C_{20}, com uma determinada condição inicial e restrições geométricas. O programa em si é composto por dois ficheiros, uma biblioteca onde são construídos os instrumentos de trabalho e um outro onde se encontra o código do *Simulated Annealing*.

A biblioteca é composta por três classes, cada uma contendo os seus construtores, destrutores e métodos devidamente comentados no código. A classe `Vector3D`, como o nome indica, é um vector de três dimensões, com métodos *getter* e *setter*, e funções para obtenção da norma de vectores e de produtos internos no espaço euclidiano em coordenadas cartesianas. A classe `Atom` serve como representação de átomos no nosso programa, e não é mais que a extensão da classe `Vector3D` com a adição do campo "String Name", para a distinção entre tipos de átomos, e de algumas outras funções. Finalmente temos a classe `Cluster`, que replica um agregado de átomos com um `Atom[ ]`.

3 Resultados e Discussão

3.1 Do C_2 ao C_{10}

Começámos por comparar os resultados do programa com os do artigo para agregados do C_2 até C_{10}. Verificámos que o sistema tendia sempre naturalmente para a cadeia linear. Depois restringimos espacialmente o sistema de átomos, através de uma barreira de potencial que crescia de forma exponencial a partir de um certo raio, de forma a que não houvesse espaço suficiente para formar a cadeia linear, e obtivemos outras estruturas.

Observámos todas as estruturas referidas no artigo [1], no qual nos baseámos, e algumas delas também se encontravam no artigo [2], que compara vários modelos. As figuras em anexo (página 16) representam algumas das estruturas obtidas (não estão todas à mesma escala).

Com os resultados da Tabela 4, retirada de [1] (DF-TB), fizemos a Tabela 1, que os compara com aqueles que nós obtivemos (Lara-Pedro-Rui). Para que não fosse exaustiva, colocámos os dados das cadeias lineares apenas para os agregados mais pequenos e para o C_{10}, visto que os resultados foram sempre de encontro ao esperado segundo o artigo. Os dados apresentados para os comprimentos de ligação e ângulos são a média sobre os valores obtidos. Os símbolos D_{2h}, C_{8v}, etc, representam o *point group* a que pertence a molécula (referência [8]), e estão relacionados com a estrutura e simetria da molécula. Só para dar alguns exemplos, uma molécula com simetria D é planar, e estruturas do tipo $D_{\infty h}$ e $C_{\infty v}$ são cadeias lineares.

Pela Tabela 1 podemos observar uma consistência notável entre os resultados por nós obtidos, através do Simulated Annealing, e os obtidos pelos autores do artigo [1]. É no entanto de estranhar que para o C_2 os resultados não sejam exactamente iguais, visto que aqui não entrou ainda o método de optimização, basta fazer o gráfico da energia em função da distância interatómica para ver o mínimo. Pensamos que isto será devido aos naturais arredondamentos feitos pela máquina durante o cálculo da energia, o qual implica muitos passos e basta que não tenham usado as mesmas funcionalidades que nós para, por exemplo, resolver o problema aos valores próprios generalizado, para isso influenciar a precisão dos resultados obtidos.

Quando os resultados não coincidem, as divergências estão fundamentalmente na terceira casa decimal, no caso das distâncias interatómicas, e na primeira casa decimal

Tabela 1: Comparação entre os nossos resultados e os do artigo [1]

Agregado	Método	Simetria	Comp. ligação (Å)	Ângulo (º)	E^{at}_{bind} (eV)
C_2	LPR	$D_{\infty h}$	1,245	180,0	3,76
	DF-TB		1,244	180,0	3,70
C_3	LPR	$D_{\infty h}$	1,288	180,0	5,57
	DF-TB		1,288	180,0	5,50
C_4	LPR	$D_{\infty h}$	1,287 1,319	180,0	5,55
	DF-TB		1,288 1,321	180,0	5,50
	LPR	D_{2h}	1,442	70,7 109,3	5,11
	DF-TB		1,443	70,7	5,10
C_6	LPR	D_{3h}	1,346	100,1 139,9	5,91
	DF-TB		1,346	100,1	5,80
C_8	LPR	C_{8v}	1,347	120,2	6,25
	DF-TB		1,348	120,3	6,20
C_{10}	LPR	$D_{\infty h}$	1,247 1,342 1,270 1,309 1,284	180,0	6,57
	DF-TB		1,246 1,345 1,269 1,311 1,284	180,0	6,50
	LPR	D_{5h}	1,312	124,5 163,5	6,61
	DF-TB		1,311	125,3	6,50
	LPR	D_{5h}	1,312	124,5 163,5	6,61
	DF-TB		1,311	125,3	6,50

no caso dos ângulos e energias. As pequenas variações relativamente ao esperado que surgiram são no sentido positivo para as energias, mas quanto aos ângulos e distâncias de equilíbrio não se pode dizer que haja uma tendência para um lado ou para o outro. Apresentámos os nossos resultados da energia com duas casas decimais, e não uma como no artigo, precisamente pois verificámos que só em dois casos é que a primeira casa decimal difere, pelo que suspeitamos que os autores não terão feito o arredondamento convencional de subir a primeira casa de uma unidade quando a segunda é superior ou igual a cinco, e aí a coincidência de resultados seria quase total.

Reparámos também que nas estruturas em anel obtínhamos por vezes dois ângulos distintos, que se intercalavam. Na tabela do artigo só fazem referência ao valor mais baixo, no entanto seria impossível obter as estruturas propostas apenas com um ângulo, pelo que inserimos os dois valores na tabela.

Quanto às estruturas obtidas, achámos um pouco estranho verificar que as estruturas mais estáveis são consistentemente cadeias lineares, visto que há uma certa "quebra de simetria"entre os átomos das pontas e os do centro, ao contrário das outras estruturas obtidas, que são em anel. Trata-se de uma estrutura mais repulsiva do que o anel, visto que as pontas não se unem, pelo que seria de esperar que tivesse uma energia menor em módulo, o que até acontece no nosso modelo, mas a estrutura para a qual o sistema tende naturalmente é sempre a cadeia linear. É interessante observar, no entanto, que na cadeia linear (ver por exemplo o C_{10} $D_{\infty h}$) os comprimentos entre os átomos são sequencialmente maiores e mais pequenos, e que há uma simetria relativamente ao átomo central (por isso na tabela estão apenas as primeiras cinco distâncias, relativas a uma das metades da cadeia), e isto conferir-lhe-á o seu equilíbrio.

A comparação com modelos teóricos é feita no artigo, e como os resultados são praticamente iguais as nossas conclusões serão idênticas.

3.2 O C_{20}

No C_{20} havia três configurações que esperávamos obter, que já foram mencionadas anteriormente: gaiola, taça e anel (ver secção 1.2).

Ao deixarmos o sistema evoluir sem grandes restrições (apenas o suficiente para os átomos não fugirem para o "infinito") verificámos que ele tendia naturalmente para a estrutura em anel (ver Figura A.5). A energia de ligação atómica da estrutura que obtivemos foi de $-7.96eV$ (sem correcção de spin-polarização) e obtivemos dois comprimentos de ligação distintos, que mais uma vez se intercalavam, como nas cadeias lineares: 1.24Å e 1.37Å.

Conseguimos também obter a cage, mas foi necessário restringir o espaço de forma a impedir os átomos de formar uma estrutura anelar. Tomou a forma de um dodecaedro, como pode ser observado na Figura A.6, de energia de coesão por átomo $-7.91eV$, comprimento de ligação de 1.44Å em média e um ângulo médio de 109° (um pentágono perfeito tem um ângulo interno de 108°). Confirma-se assim que no nosso método a configuração de menor energia é a anelar.

Quanto à taça não conseguimos que fosse atingida naturalmente. Só forçando muito a configuração, colocando os átomos inicialmente nas posições certas e deixando o programa fazer pequenos ajustes, o que não tem grande significado. Encontrámos no entanto muitas outras estruturas, apesar de nenhuma das referências as mencionar.

3.3 O C_{60}

Quanto ao famoso C_{60}, nós simplesmente não tivemos acesso a máquinas suficientemente rápidas para fazer uma simulação condigna em menos de um mês, visto que mexer em matrizes 240×240 não é coisa simples... Mesmo assim seria duvidoso que conseguíssemos nesse espaço de tempo, visto que só o programa para o C_{20} nos demorava cerca de uma semana a correr na totalidade.

Assim, o que fizemos foi restringir os átomos a uma camada entre duas esferas de raios 6.6 e $6.8bohr$, sendo o raio do fulereno C_{60} de $6.7bohr$, conferindo uma energia gigantesca ao agregado sempre que um átomo saísse desta região e não calculando sequer a energia atractiva nestes casos, que é o que mais tempo consome. Dentro desta fina camada as posições iniciais dos átomos eram aleatórias, pelo que mesmo assim pudemos observar o fulereno 'a nascer' − Figura A.7.

Apesar de a estrutura não ter ficado perfeita, visto haver pentágonos ligados a pentágonos (o que mostra o que teria acontecido se não tivéssemos feito uma tão grande restrição), ainda assim está bastante próximo do que queremos, e podemos comparar a energia e as distâncias. A energia de ligação por átomo que obtivemos (sem correcção de spin-polarização) foi de $-8.60eV$, próxima da referida no artigo de $-8.85eV$, e os dois comprimentos de ligação distintos foram de cerca de 1.38Å e 1.46Å, já bastante parecidos com os $1.397\mathring{A}$ e $1.449\mathring{A}$ do artigo.

4 Conclusão

Ficámos de uma forma geral bastante satisfeitos com os resultados obtidos. Conseguimos implementar o método do artigo que nos foi apresentado e aproveitando o esquema do *simulated annealing* que tínhamos trabalhado nas aulas de Física Computacional conseguimos chegar a configurações e energias bastante próximas do esperado. Isto verificou-se também para o C_{20}, apesar de este sistema não ter sido trabalhado pelos autores.

Quanto ao C_{20} e ao C_{60}, gostaríamos de ter explorado melhor os agregados, mas a implementação do método demorou algum tempo, pelo que já não pudemos despender de muito para os estudar e para correr as morosas simulações.

Trata-se de uma área interdisciplinar e actual, pelo que foi uma experiência enriquecedora e gratificante, além de nos ter ensinado muito sobre paciência.

Referências

[1] D. Porezag, Th. Frauenheim, Th. Köhler, G. Seifert, R. Kaschner, *Construction of tight-binding-like potentials on the basis of density-functional theory: Application to Carbon*, Phys. Rev. B **51**, 12 947 (1995).

[2] R. O. Jones, *Density functional study of carbon clusters C2n - Structure and bonding in the neutral clusters*, J. Chem. Phys. **110**, 5189 (1999).

[3] Wikipedia, *Buckyball*, `http://en.wikipedia.org/wiki/Fullerene` (Julho 2009).

[4] Chemistry, Structures and 3D Molecules, C_{60}, `http://www.3dchem.com/imagesofmolecules/c60.jpg` (2008).

[5] Schlumberger Limited, *The Discovery of Fullerenes - The Smallest Fullerene: C_{20}*, `http://199.6.131.12/en/scictr/watch/fullerenes/smallest.htm` (2008).

[6] R. M. Martin, *Electronic Structure - Basic Theory and Practical Methods*, Cambridge University Press, 2004 (1a Edição).

[7] Arkady Krasheninnikov, *Introduction to electronic structure simulations - Lecture 5 - Semi-empirical and tight-binding methods*, University of Helsinki and Helsinki University of Technology, `http://tfy.tkk.fi/~asf/physics/lectures/PDF/lect5.pdf` (2008).

[8] Wikipedia, *Molecular Symmetry*, `http://en.wikipedia.org/wiki/Molecular_symmetry` (Julho 2009).

[9] J.C. Grossman, *Molecular Carbon*, `http://altair.physics.ncsu.edu/projects/c20/c20.html` (1995).

Lista de Figuras

Lista de Tabelas

A Anexos

A.1 Figuras

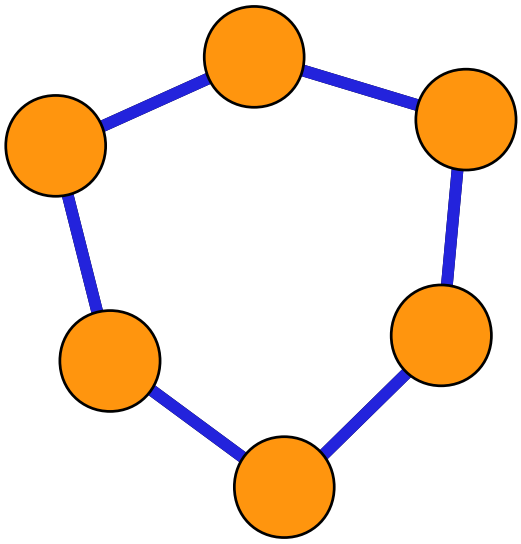

Figura A.1: Isómero D_{3h} do C_6.

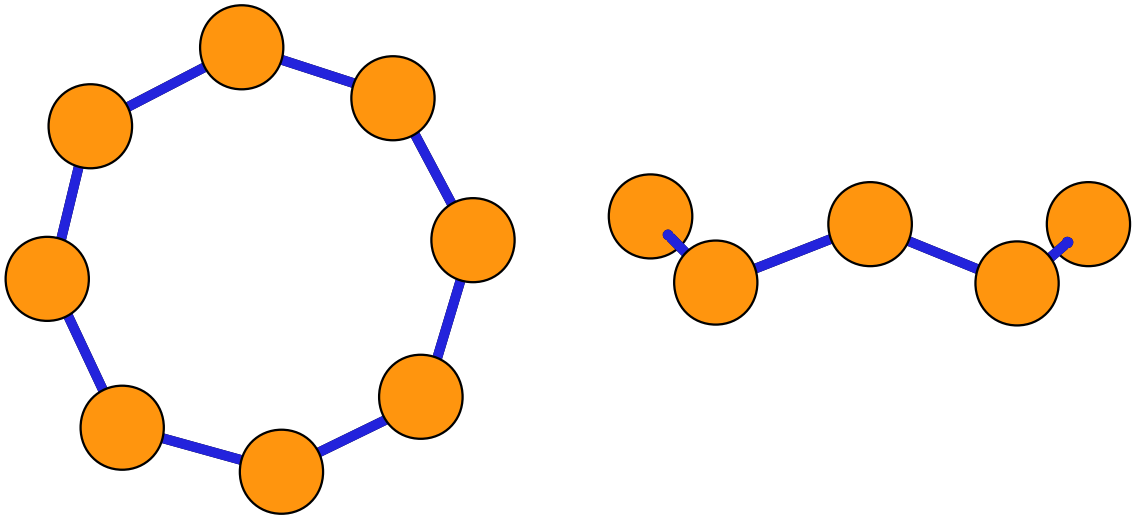

Figura A.2: Isómero D_{8v} do C_8, visto "de cima"e "de perfil".

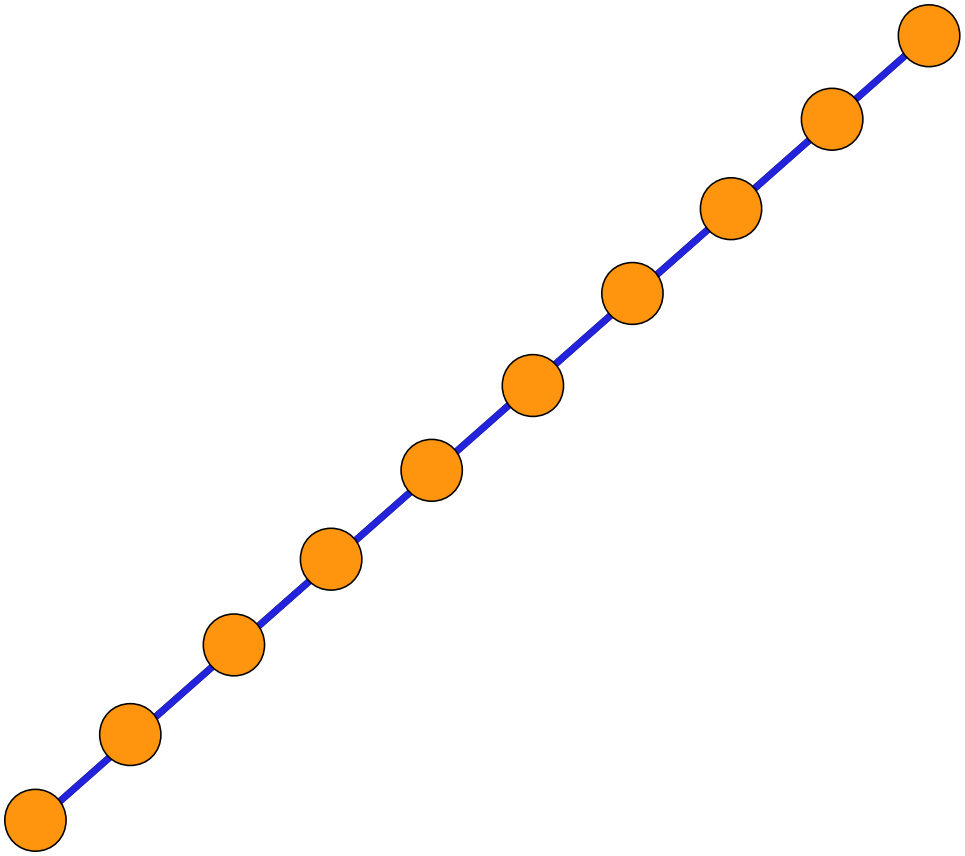

Figura A.3: Isómero linear ($D_{\infty h}$) do C_{10}

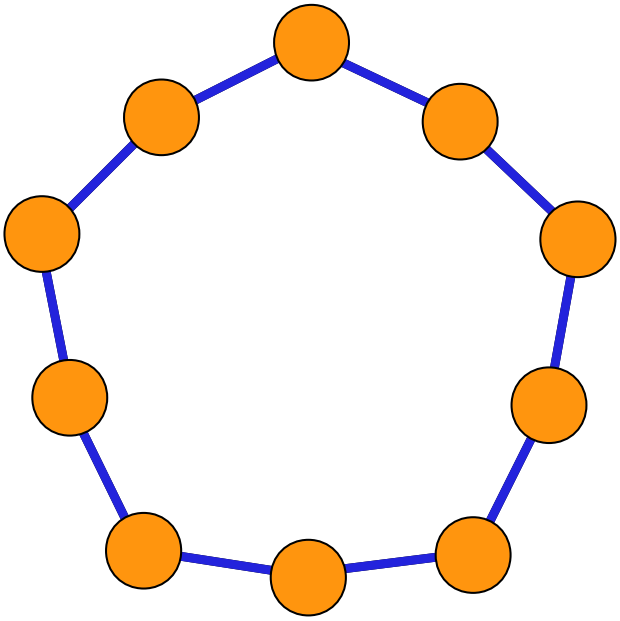

Figura A.4: Isómero D_{5h} do C_{10}.

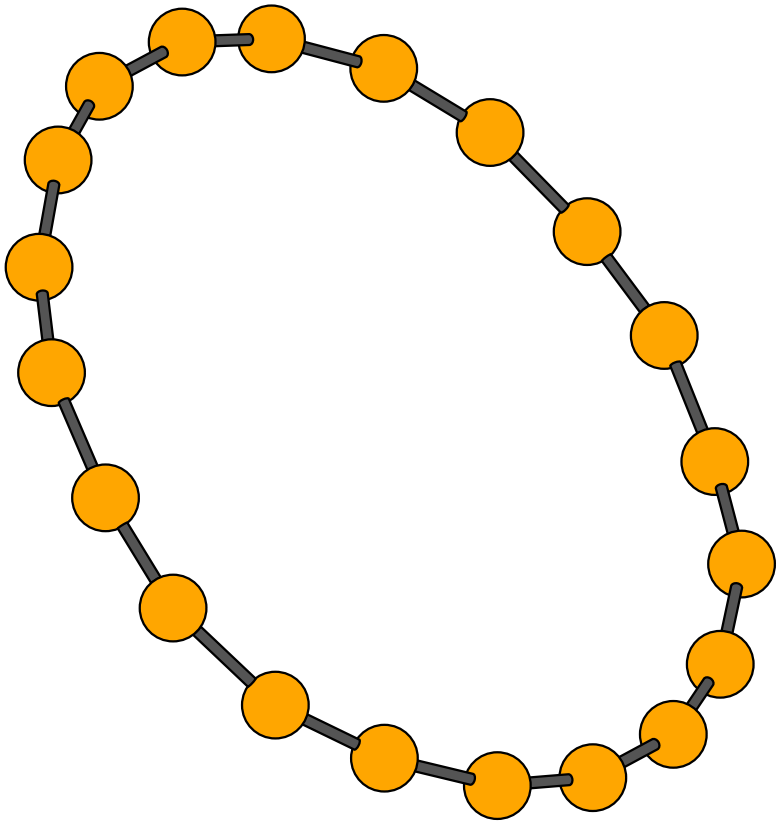

Figura A.5: C_{20} em configuração "anel".

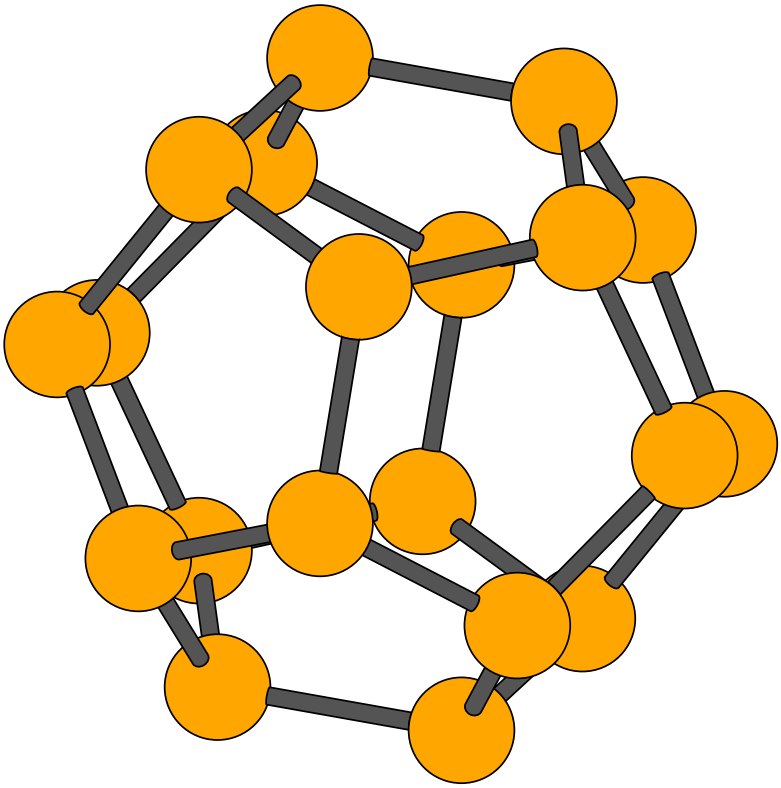

Figura A.6: C_{20} em configuração "gaiola".

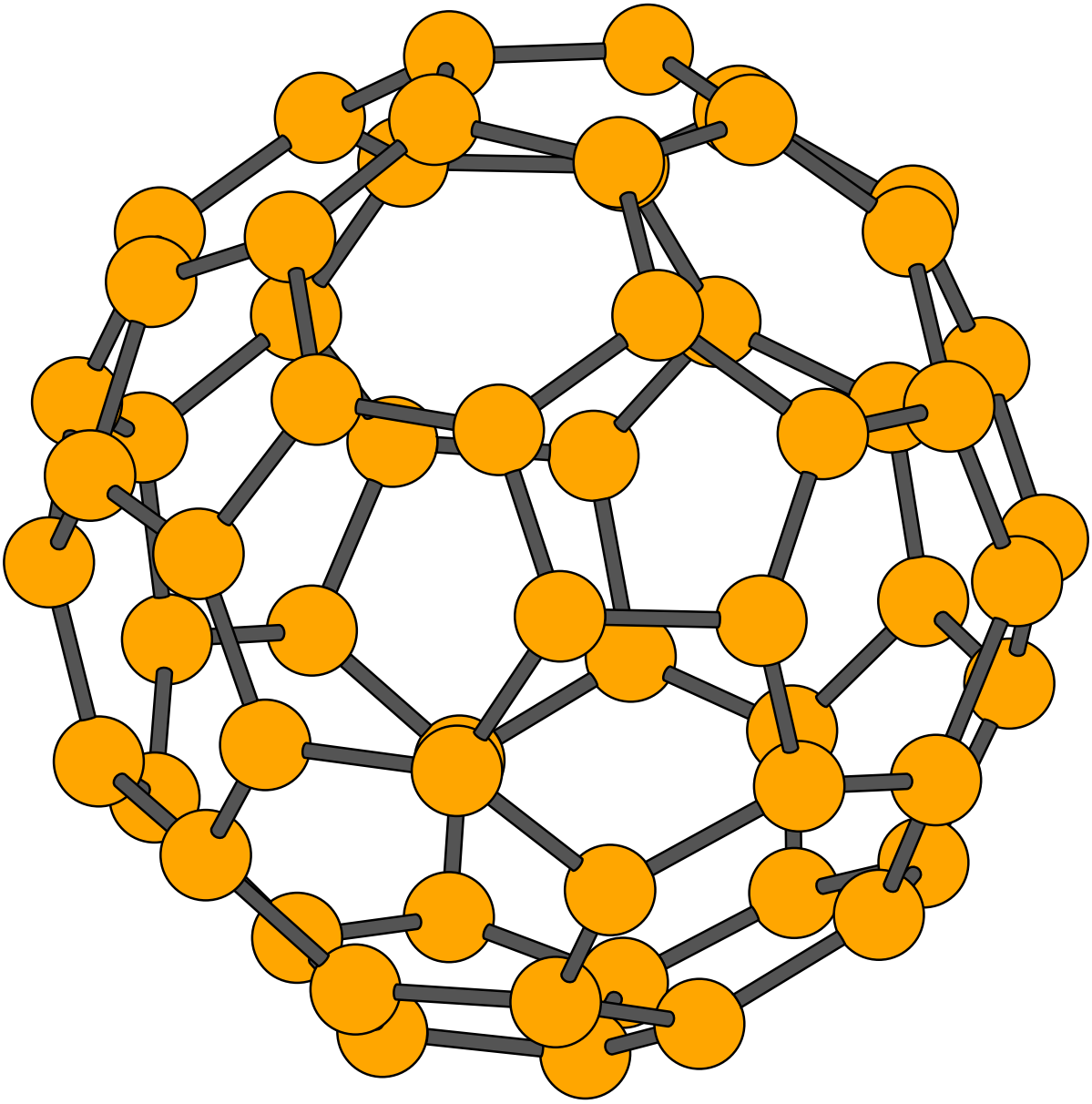

Figura A.7: C_{60} gaiola.

A.2 Tabelas

Tabela 2: Tabela retirada do artigo [1], mas corrigida (na tabela original $\mathbf{S}_{pp\sigma}$ e $\mathbf{S}_{pp\pi}$ estão trocados): coeficientes e fronteiras na expansão polinomial de Chebyshev (2.4) para os elementos da matriz Hamiltoniana e de Sobreposição, com $r \in (a, b)$, para ligações Carbono-Carbono. Valores para a e b em bohrs, valores dos coeficientes em hartrees.

Matrix	$\mathbf{c_1}$	$\mathbf{c_2}$	$\mathbf{c_3}$	$\mathbf{c_4}$	$\mathbf{c_5}$
(a,b)	$\mathbf{c_6}$	$\mathbf{c_7}$	$\mathbf{c_8}$	$\mathbf{c_9}$	$\mathbf{c_{10}}$
$\mathbf{H_{ss\sigma}}$	-0,4663805	0,3528951	-0,1402985	0,0050519	0,0269723
(1,0;7,0)	-0,0158810	0,0036716	0,0010301	-0,0015546	0,0008601
$\mathbf{H_{sp\sigma}}$	0,3395418	-0,2250358	0,0298224	0,0653476	-0,0605786
(1,0;7,0)	0,0298962	-0,0099609	0,0020609	0,0001264	-0,0003381
$\mathbf{H_{pp\sigma}}$	0,2422701	-0,1315258	-0,0372696	0,0942352	-0,0673216
(1,0;7,0)	0,0316900	-0,0117293	0,0033519	-0,0004838	-0,0000906
$\mathbf{H_{pp\pi}}$	-0,3793837	0,3204470	-0,1956799	0,0883986	-0,0300733
(1,0;7,0)	0,0074465	-0,0008563	-0,0004453	0,0003842	-0,0001855
$\mathbf{S_{ss\sigma}}$	0,4728644	-0,3661623	0,1594782	-0,0204934	-0,0170732
(1,0;7,0)	0,0096695	-0,0007135	-0,0013826	0,0007849	-0,0002005
$\mathbf{S_{sp\sigma}}$	-0,3662838	0,2490285	-0,0431248	-0,0584391	0,0492775
(1,0;7,0)	-0,0150447	-0,0010758	0,0027734	-0,0011214	0,0002303
$\mathbf{S_{pp\sigma}}$	0,3715732	-0,3070867	0,1707304	-0,0581555	0,0061645
(1,0;7,0)	0,0051460	-0,0032776	0,0009119	-0,0001265	-0,0000227
$\mathbf{S_{pp\pi}}$	-0,1359608	0,0226235	0,1406440	-0,1573794	0,0753818
(1,0;7,0)	-0,0108677	-0,0075444	0,0051533	-0,0013747	0,0000751

Tabela 3: Tabela retirada do artigo [1]: coeficientes e fronteiras na expansão polinomial de Chebyshev (2.4) para o potencial repulsivo de curta-distância, com $r \in (a, b)$, para ligações Carbono-Carbono. Valores para a e b em bohrs, valores dos coeficientes em hartrees.

V_{rep}	c_1	c_2	c_3	c_4	c_5
(a,b)	c_6	c_7	c_8	c_9	c_{10}
V_{rep}	2,2681036	-1,9157174	1,1677745	-0,5171036	0,1529242
(1,0;4,10)	-0,0219294	-0,0000002	-0,0000001	-0,0000005	0,0000009

Tabela 4: Tabela retirada do artigo [1], com os resultados obtidos pelos autores para os agregados de C_2 até C_{10}. A energia E^{at}_{bind} é o módulo da energia de ligação por átomo com a correcção de spin-polarização.

Agregado	Método	Simetria	Comp. ligação (Å)	Ângulo (º)	E^{at}_{bind} (eV)
C_2	DF-TB	$D_{\infty h}$	1,244	180,0	3,7
	CCD	$D_{\infty h}$	1,245	180,0	2,9
C_3	DF-TB	$D_{\infty h}$	1,288	180,0	5,5
	CCD	$D_{\infty h}$	1,278	180,0	4,2
C_4	DF-TB	$D_{\infty h}$	1,288 1,321	180,0	5,5
	CASSCF[a]	$D_{\infty h}$	1,306 1,287	180,0	
	DF-TB	D_{2h}	1,443	70,7	5,1
	CCD	D_{2h}	1,425	61,5	4,3
	CASSCF[a]	D_{2h}	1,432	64,5	
C_5	DF-TB	$D_{\infty h}$	1,257 1,315	180,0	6,2
	CCD	$D_{\infty h}$	1,271 1,275	180,0	4,8
C_6	DF-TB	$D_{\infty h}$	1,265 1,324 1,287	180,0	6,1
	DF-TB	D_{3h}	1,346	100,1	5,8
	CCD	D_{3h}	1,316	90,4	4,8
C_7	DF-TB	$D_{\infty h}$	1,245 1,337 1,280	180,0	6,4
	CCD	$D_{\infty h}$	1,270 1,280 1,264	180,0	5,0
C_8	DF-TB	$D_{\infty h}$	1,253 1,335 1,279 1,308	180,0	6,4
	DF-TB	C_{8v}	1,348	120,3	6,2
	CCD	D_{4h}	1,240 1,380	107,1	5,0
C_9	DF-TB	$D_{\infty h}$	1,240 1,350 1,263 1,302	180,0	6,6
	CCD	$D_{\infty h}$	1,269 1,283 1,261 1,269	180,0	5,2
C_{10}	DF-TB	$D_{\infty h}$	1,246 1,345 1,269 1,311 1,284	180,0	6,5
	DF-TB	D_{5h}	1,311	125,3	6,5
	CCD	D_{5h}	1,290	119,4	5,4

A.3 Código fonte

A.3.1 Programa principal

```
#include <cmath>
#include <string>
#include <iostream>
#include <libnum/libnum.h>
#include <sys/time.h>
using namespace std;
#include "objects.h"

int main()
{

        LIBNUM::RngInit();


        string movieFileName = "c20";
        string lastClusterFile = "last_c20";
        const double hartree = 27.2113845;
        int n = 20; // atomos
        int k = 1000*n; // iteracoes
        double T = 500.0;
        double Tmin = 1.0E-4;
        double coolingRate = 0.005; // = (1- alpha)
        double Kb = 8.617343E-5;
        double Rmax = 1.0;
        double RrateOfChange = 0.05; // variacao de Rmax
        int frameRate = k;
        double E1,E2,iatom,deltaE,oldx,oldy,oldz,cnt,xp,yp,zp,rej;

        struct timeval start, end;
        long mtime, seconds, useconds;

        Cluster cn(n,6.0); // c60 r = 6.7 , c20 r = 3.85 raios das cages
        cn.initializeBox(5.0);

        cn.createMovie(movieFileName);

        E1 = cn.getEnergy();

        //Simulated Annealing
        while(T>=Tmin)
        {
                cnt = 0;
                //Passeio de Metropolis-Monte Carlo
                for (int i = 0; i < k ; i++)
                {
                        cnt++;
                        iatom=LIBNUM::RngInt(n);
                        oldx=cn.getAtom(iatom).getX();
                        oldy=cn.getAtom(iatom).getY();
                        oldz=cn.getAtom(iatom).getZ();

                       cn.createPerturbation(Rmax,xp,yp,zp);
                        cn.getAtom(iatom).setX(oldx+xp);
                        cn.getAtom(iatom).setY(oldy+yp);
                        cn.getAtom(iatom).setZ(oldz+zp);

                        E2 = cn.getEnergy();

                        deltaE = E2-E1;

                        if (deltaE < 0.0)
                        {
                            E1 = E2;
                        }
                        else
                        {
                                if( LIBNUM::RngDouble() > exp(-deltaE/(Kb*T)) )
                                {
                                        cn.getAtom(iatom).setX(oldx);
                                        cn.getAtom(iatom).setY(oldy);
                                        cn.getAtom(iatom).setZ(oldz);
                                        rej++;
                                }
                                else
                                {
                                    E1 = E2;
                                }
                        }

                        if(cnt == 100)
                        {
                                if (rej > 50)
                                {
                                    Rmax = Rmax * (1.0 - RrateOfChange);
                                }
                                else
                                {
                                    Rmax = Rmax * (1.0 + RrateOfChange);
                                }

                                        rej = 0;
                                        cnt = 0;
                        }
```

```
                        if((i+1)%frameRate == 0)
                        {
                                cn.appendFrame(movieFileName);
                                cn.storeInFile(lastClusterFile);
                                gettimeofday(&end, NULL);
                                seconds  = end.tv_sec  - start.tv_sec;
                                useconds = end.tv_usec - start.tv_usec;
                                mtime = ((seconds) * 1000 + useconds/1000.0) + 0.5;

                                //a energia final vem dada em eV e já com a correcção de spin-polarização
                                cout << T << "␣" << E1*hartree << "␣" << (0.0415+(E1-n*(-(2.0*0.50097+2.0*0.19930)))/n
                                      )*hartree << "␣" << Rmax << "␣" << mtime << "\n";
                                gettimeofday(&start, NULL);
                        }
                }

                T = (1.0 - coolingRate)*T;
        }

        return 0;

}
```

A.3.2 Biblioteca

```
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_eigen.h>
#include <gsl/gsl_sort_vector.h>


#define PI 3.14159265

class Vector3D
{
    double x,y,z;

public:

    //Construtores:

    Vector3D(double x, double y, double z)
    {
        this-> x = x;
        this-> y = y;
        this-> z = z;
    }

    //Inicializa como vector nulo (0; 0; 0)
    Vector3D()
    {
        this-> x = 0.0;
        this-> y = 0.0;
        this-> z = 0.0;
    }

    //Metodos:

    double getX()
    {
        return x;
    }

    double getY()
    {
        return y;
    }

    double getZ()
    {
        return z;
    }

    //Devolve o quadrado da norma do vector.
    double NormSq()
    {
        return(x*x+y*y+z*z);
    }

    //Devolve a norma do vector.
    double Norm()
    {
        return(sqrt(NormSq()));
    }

    //Devolve o produto interno com o vector v.
    double dotProduct(Vector3D v)
    {
        return v.getX()*x+v.getY()*y+v.getZ()*z;
    }

    void setX(double x)
    {
        this->x = x;
    }

    void setY(double y)
    {
```

```
        this->y = y;
    }

    void setZ(double z)
    {
        this->z = z;
    }

    void setCoordinates(double x, double y, double z)
    {
        setX(x);
        setY(y);
        setZ(z);
    }
};


class Atom: public Vector3D
{
    string name;

public:

    //Construtores:

    Atom(string name, double x, double y, double z):Vector3D(x,y,z)
    {
        this-> name = name;
    }

    Atom():Vector3D()
    {
        this-> name = "NoName";
    }

    //Métodos:

    string getName()
    {
        return name;
    }

    //Devolve a distancia ao Atom at.
    double getDistanceTo(Atom at)
    {
        return  sqrt((at.getX()-getX())*(at.getX()-getX())+(at.getY()-getY())*(at.getY()-getY())+(at.getZ()-getZ())*(at.getZ()
            -getZ()));
    }

    void PrintCoordinates()
    {
        printf("%s %14lf %14lf %14lf\n",name.c_str(),getX(),getY(),getZ());
    }

    //Le atomos de um ficheiro com formato 'nome x y z'
    void ReadFromFile(FILE *fin)
    {
        char el[10];
        double x,y,z;
        fscanf(fin,"%s %lf %lf %lf\n", &el,&x, &y, &z);
        name = el;
        setCoordinates(x,y,z);
    }

};

class Cluster
{
    int n;
    Atom *atom;
    double potencialIncreaseRadius;
    struct timeval start, end;
    long mtime, seconds, useconds;

public:

    // Construtores:

    //Constroi um agregado de n atomos em que a sua energia aumenta se a
    //norma do vector posicao de cada atomo for maior que double
    //potencialIncreaseRadius, aumenta exponencialmente quanto maior for a
    //norma do vector posicao.
    Cluster(int n, double potencialIncreaseRadius)
    {
        this->n = n;
        atom = new Atom[n];
        this-> potencialIncreaseRadius = potencialIncreaseRadius;
    }

    //Constroi o mesmo tipo de agregado que o anterior construtor mas carrega
    //as coordenadas dos atomos guardadas num ficheiro.
    Cluster(int n, const char * filename, double potencialIncreaseRadius)
    {
        this->n = n;
        atom = new Atom[n];
        this-> potencialIncreaseRadius = potencialIncreaseRadius;
        FILE * pFile = fopen (filename,"r");
        for (int i=0; i<n; i++) {
            atom[i].ReadFromFile(pFile);
```

```
    }
    fclose (pFile);
}

//Destrutores:
~Cluster()
{
    delete[] atom;
}

//Metodos:

//Devolve referencia a atomo com indice int i.
Atom & getAtom(int i)
{
    return atom[i];
}

//Imprime coordenadas dos atomos.
void PrintCoordinates()
{
    for (int i=0; i<n; i++) {
        atom[i].PrintCoordinates();
    }
}

//Inicializa as coordenadas dos atomos a partir de um ficheiro criado por
//storeInFile(string filename)
void ReadFromFile(string FileName)
{
    FILE *fin = fopen(FileName.c_str(),"r");

    if (fin==NULL)
    {
        printf("FATAL␣Cluster::ReadFromFile,␣file␣%s␣not␣found\n",FileName.c_str());
        exit(1);
    }

    for (int i=0; i<n; i++)
    {
        atom[i].ReadFromFile(fin); // este ReadFromFile e' um metodo da class Atom !!!
    }

    fclose(fin);
}

//Devolve distancia entre atomos no indice i e j.
double getDistance(int i, int j) // metodo
{
    // Calculate the distance between atom[i] and atom[j]
    if (i==j)
    {
        printf("Warning␣Cluster::getDistance,␣i␣and␣j␣are␣the␣same!\n");
        return(0.0);
    }
    else
    {
        return atom[i].getDistanceTo(atom[j]);
    }
}

//Imprime distancias entre atomos.
void PrintDistances()
{
    for (int i=0; i<n; i++)
    {
        for (int j=i+1; j<n; j++)
        {
            printf("|r%2.2d-r%2.2d|␣=␣%14lf\n",j,i,getDistance(i,j));
        }
    }
}

//Guarda as coordenadas do agregado no ficheiro string filename.
void storeInFile(string filename)
{
    FILE *fout= fopen((filename+".dat").c_str(),"w");

    for ( int i = 0; i < n ; i++)
    {
        fprintf(fout,"%s␣%14lf␣%14lf␣%14lf\n",atom[i].getName().c_str(),atom[i].getX(),atom[i].getY(),atom[i].getZ());
    }
    fclose(fout);
}

//Cria ficheiro para filme XBS com nome string filename.
void createMovie(string filename)
{
    FILE *fout= fopen((filename+".bs").c_str(),"w");
    FILE *fout1= fopen((filename+".mv").c_str(),"w");

    fprintf(fout,"\n");
    fprintf(fout,"inc␣1.0");
    fprintf(fout,"\n\n");
    fprintf(fout,"spec␣C␣0.6␣Orange");
    fprintf(fout,"\n\n");
    fprintf(fout,"bonds␣C␣C␣0.0␣3.0␣0.1␣Blue");
    fprintf(fout,"\n\n");
```

```
    fprintf(fout1,"frame␣\n");

    for ( int i = 0; i < n ; i++)
    {
        fprintf(fout,"atom␣C␣%14lf␣%14lf␣%14lf\n",atom[i].getX(),atom[i].getY(),atom[i].getZ());
        fprintf(fout1,"%14lf␣%14lf␣%14lf\n",atom[i].getX(),atom[i].getY(),atom[i].getZ());
    }

    fclose(fout);
    fclose(fout1);
}

//Adiciona frame ao filme de nome string filename.
void appendFrame(string filename)
{
    FILE *fout= fopen((filename+".mv").c_str(),"a");
    fprintf(fout,"frame\n");
    for ( int i = 0; i < n ; i++)
    {
        fprintf(fout,"%14lf␣%14lf␣%14lf\n",atom[i].getX(),atom[i].getY(),atom[i].getZ());
    }

    fclose(fout);
}

//Altera os valores de (addx, addy, addz) para um ponto aleatorio na
//esfera de raio double Rmax.
double createPerturbation(double Rmax, double &addx,double &addy, double &addz)
{
    double t,theta,f,fi,r;
    t=LIBNUM::RngDouble();
    f=LIBNUM::RngDouble();
    r=(LIBNUM::RngDouble())*Rmax;
    theta=t*2.0*2.0*asin(1.0);
    fi=f*2.0*asin(1.0);

    addx=r*cos(theta)*sin(fi);
    addy=r*sin(theta)*sin(fi);
    addz=r*cos(fi);

}


//Calculo da Energia

double Chebyshev (double a,double b, double r,double c[10])
{

    double y = (r-(b+a)/2.0)/((b-a)/2.0);

    return c[0]/2.0 + c[1]*y + c[2]*(2.0*y*y-1.0) + c[3]*y*(4.0*y*y-3.0)
           + c[4]*(8.0*y*y*(y*y-1.0)+1.0) + c[5]*(4.0*y*y*(4.0*y*y*y-5.0*y)+5.0*y)
           + c[6]*(32.0*y*y*y*y*y*y-48.0*y*y*y*y+18.0*y*y-1.0)
           + c[7]*(64.0*y*y*y*y*y*y*y*y-112.0*y*y*y*y*y*y+56.0*y*y*y-7.0*y)
           + c[8]*(128.0*y*y*y*y*y*y*y*y*y-256.0*y*y*y*y*y*y*y+160.0*y*y*y*y*y-32.0*y*y+1.0)
           + c[9]*(256.0*y*y*y*y*y*y*y*y*y*y-576.0*y*y*y*y*y*y*y*y+432.0*y*y*y*y*y*y-120.0*y*y*y+9.0*y);

}

//Preenche gsl_matrix * mat com valores da matriz hamiltoniana do agregado.
void H(gsl_matrix * mat)
{
    double a1 = 1.0;
    double b1 = 7.0;

    double HCCssSig1_7[10] =
          {-0.4663805,0.3528951,-0.1402985,0.0050519,0.0269723,-0.015881,0.0036716,0.0010301,-0.0015546,0.0008601};
    double HCCspSig1_7[10] =
          {0.3395418,-0.2250358,0.0298224,0.0653476,-0.0605786,0.0298962,-0.0099609,0.0020609,0.0001264,-0.0003381};
    double HCCppSig1_7[10] =
          {0.2422701,-0.1315258,-0.0372696,0.0942352,-0.0673216,0.0316900,-0.0117293,0.0033519,-0.0004838,-0.0000906};
    double HCCppPi1_7[10] =
          {-0.3793837,0.320447,-0.1956799,0.0883986,-0.0300733,0.0074465,-0.0008563,-0.0004453,0.0003842,-0.0001855};


    for ( int a = 0; a < n ; a++)
    {
        for ( int b = 0; b < n ; b++)
        {
            if (a == b)
            {
                for (int i = 0 ; i < 4 ; i++)
                {
                    for (int j = 0 ; j < 4 ; j++)
                    {
                        if (i != j)
                        {
                            gsl_matrix_set(mat,a*4+i,b*4+j,0.0);
                        }
                        else if ( i == 0 )
                        {
                            gsl_matrix_set(mat,a*4+i,b*4+j,-0.50097);
                        }
                        else
                        {
                            gsl_matrix_set(mat,a*4+i,b*4+j,-0.19930);
                        }
                    }
                }
```

```
            }

            else
            {
                double r = getDistance(a,b);

                if (r>b1 || r<a1)
                {
                    for (int i = 0 ; i < 4 ; i++)
                    {
                        for (int j = 0 ; j < 4 ; j++)
                        {
                            gsl_matrix_set(mat,a*4+i,b*4+j,0.0);
                        }
                    }
                }
                else
                {
                    double c[3] = {(atom[b].getX()-atom[a].getX())/r,(atom[b].getY()-atom[a].getY())/r,(atom[b].getZ()-
                        atom[a].getZ())/r};

                    for (int i = 0 ; i < 4 ; i++)
                    {
                        for (int j = 0 ; j < 4 ; j++)
                        {
                            if (i == 0 && j == 0)
                            {
                                gsl_matrix_set(mat,a*4+i,b*4+j,Chebyshev(a1,b1,r,HCCssSig1_7));
                            }
                            else if ( i == 0 )
                            {
                                gsl_matrix_set(mat,a*4+i,b*4+j,c[j-1]*Chebyshev(a1,b1,r,HCCspSig1_7));
                            }
                            else if ( j == 0 )
                            {
                                gsl_matrix_set(mat,a*4+i,b*4+j,-c[i-1]*Chebyshev(a1,b1,r,HCCspSig1_7));
                            }
                            else if ( i == j )
                            {
                                gsl_matrix_set(mat,a*4+i,b*4+j,c[i-1]*c[i-1]*Chebyshev(a1,b1,r,HCCppSig1_7)+(1.0-c[i-1]*c[
                                    i-1])*Chebyshev(a1,b1,r,HCCppPi1_7));
                            }
                            else
                            {
                                gsl_matrix_set(mat,a*4+i,b*4+j,c[i-1]*c[j-1]*(Chebyshev(a1,b1,r,HCCppSig1_7) - Chebyshev(
                                    a1,b1,r,HCCppPi1_7)));
                            }

                        }
                    }
                }
            }
        }
    }
}

//Preenche gsl_matrix * mat com valores da matriz de sobreposicao do agregado
void S(gsl_matrix * mat)
{
    double a1 = 1.0;
    double b1 = 7.0;

    double SCCssSig1_7[10] =
          {0.4728644,-0.3661623,0.1594782,-0.0204934,-0.0170732,0.0096695,-0.0007135,-0.0013826,0.0007849,-0.0002005};
    double SCCspSig1_7[10] =
          {-0.3662838,0.2490285,-0.0431248,-0.0584391,0.0492775,-0.0150447,-0.0010758,0.0027734,-0.0011214,0.0002303};
    double SCCppPi1_7[10] =
          {0.3715732,-0.3070867,0.1707304,-0.0581555,0.0061645,0.0051460,-0.0032776,0.0009119,-0.0001265,-0.0000227};
    double SCCppSig1_7[10] =
          {-0.1359608,0.0226235,0.1406440,-0.1573794,0.0753818,-0.0108677,-0.0075444,0.0051533,-0.0013747,0.0000751};


    for ( int a = 0; a < n ; a++)
    {
        for ( int b = 0; b < n ; b++)
        {
            if (a == b)
            {
                for (int i = 0 ; i < 4 ; i++)
                {
                    for (int j = 0 ; j < 4 ; j++)
                    {
                        if ( i == j)
                        {
                            gsl_matrix_set(mat,a*4+i,b*4+j,1.0);
                        }
                        else
                        {
                            gsl_matrix_set(mat,a*4+i,b*4+j,0.0);
                        }
                    }
                }
            }
            else
            {
                double r = getDistance(a,b);

                if (r>b1 || r<a1)
                {
```

```
                    for (int i = 0 ; i < 4 ; i++)
                    {
                        for (int j = 0 ; j < 4 ; j++)
                        {
                            gsl_matrix_set(mat,a*4+i,b*4+j,0.0);
                        }
                    }
                }
                else
                {
                    double c[3] = {(atom[b].getX()-atom[a].getX())/r,(atom[b].getY()-atom[a].getY())/r,(atom[b].getZ()-
                        atom[a].getZ())/r};

                    for (int i = 0 ; i < 4 ; i++)
                    {
                        for (int j = 0 ; j < 4 ; j++)
                        {
                            if (i == 0 && j == 0)
                            {
                                gsl_matrix_set(mat,a*4+i,b*4+j,Chebyshev(a1,b1,r,SCCssSig1_7));
                            }
                            else if ( i == 0 )
                            {
                                gsl_matrix_set(mat,a*4+i,b*4+j,c[j-1]*Chebyshev(a1,b1,r,SCCspSig1_7));
                            }
                            else if ( j == 0 )
                            {
                                gsl_matrix_set(mat,a*4+i,b*4+j,-c[i-1]*Chebyshev(a1,b1,r,SCCspSig1_7));
                            }
                            else if ( i == j )
                            {
                                gsl_matrix_set(mat,a*4+i,b*4+j,c[i-1]*c[i-1]*Chebyshev(a1,b1,r,SCCppSig1_7)+(1.0-c[i-1]*c[
                                    i-1])*Chebyshev(a1,b1,r,SCCppPi1_7));
                            }
                            else
                            {
                                gsl_matrix_set(mat,a*4+i,b*4+j, c[i-1]*c[j-1]*(Chebyshev(a1,b1,r,SCCppSig1_7) - Chebyshev(
                                    a1,b1,r,SCCppPi1_7)));
                            }

                        }
                    }
                }
            }
        }
    }
}

//Imprime matriz com dimensoes d1 x d2.
void printmatrix(gsl_matrix * mat, int d1, int d2)
{
    for (int i = 0 ; i < d1 ; i++)
    {
        for (int j = 0 ; j < d2 ; j++)
        {
            printf("%14f ", gsl_matrix_get (mat, i, j));
        }
        cout << "\n";
    }
}

//Devolve energia de coesao das orbitais mais a energia de um atomo livre.
double getEnergyAtractive()
{
    double energy = 0.0;
    gsl_vector_complex * alpha = gsl_vector_complex_calloc(n*4);
    gsl_vector * beta = gsl_vector_calloc(n*4);
    gsl_vector * energies = gsl_vector_calloc(n*4);
    gsl_matrix * HH = gsl_matrix_calloc (n*4, n*4);
    gsl_matrix * SS = gsl_matrix_calloc (n*4, n*4);
    gsl_eigen_gen_workspace * workH = gsl_eigen_gen_alloc (n*4);

    H(HH);
    S(SS);
    gsl_eigen_gen (HH,SS, alpha, beta, workH);

    for (int i=0; i<n*4; i++)
    {
        gsl_vector_set(energies,i,GSL_REAL(gsl_vector_complex_get(alpha,i))/gsl_vector_get(beta,i));
    }

    gsl_sort_vector(energies);

    for (int i = 0 ; i < n*2 ; i++)
    {
        energy += 2.0*gsl_vector_get(energies,i);
    }

    gsl_matrix_free(HH);
    gsl_matrix_free(SS);
    gsl_vector_complex_free(alpha);
    gsl_vector_free(beta);
    gsl_vector_free(energies);
    gsl_eigen_gen_free (workH);

    return energy;
}

//Devolve a energia de repulsao entre atomos.
```

```
double getEnergyRepulsive()
{
    double r;
    double a1 = 1.0;
    double b1 = 4.1;
    double energy = 0.0;
    double VCCrep[10] =
          {2.2681036,-1.9157174,1.1677745,-0.5171036,0.1529242,-0.0219294,-0.0000002,-0.0000001,-0.0000005,0.0000009};

    for ( int a = 0; a < n ; a++)
    {
        double radius = atom[a].Norm();

        if (radius > potencialIncreaseRadius)
        {
            energy += 0.0001*exp(6.0*radius/potencialIncreaseRadius);
        }

        for ( int b = 0; b < a ; b++)
        {
            double r = getDistance(a,b);

            if (r <= b1)
            {
                if (r < a1)
                {
                    energy += 1.0E20;
                }
                else
                {
                    energy += Chebyshev(a1,b1,r,VCCrep);
                }
            }
        }
    }

    return energy;
}

//Devolve a energia total do sistema.
double getEnergy()
{
    double Erep = getEnergyRepulsive();

    if (Erep >= 1.0E10)
    {
        return Erep;
    }
    else
    {
        return Erep+getEnergyAtractive();
    }

    //return getEnergyAtractive()+getEnergyRepulsive();
}

//Devolve a energia de ligacao por atomo.
double getEnergyBondPerAtom()
{

    double Etot, Eint;
    double Espin = 0.0415; //hartree

    Etot=getEnergyAtractive()+getEnergyRepulsive();
    Eint=-(2.0*0.50097+2.0*0.19930);
    return Espin+(Etot-n*Eint)/n;
}


//Centra o agregado na origem
double centerAtoms()
{
    for ( int a = 1; a < n ; a++)
    {
        atom[a].setX(atom[a].getX()-atom[0].getX());
        atom[a].setY(atom[a].getY()-atom[0].getY());
        atom[a].setZ(atom[a].getZ()-atom[0].getZ());
    }

    atom[0].setX(0.0);
    atom[0].setY(0.0);
    atom[0].setZ(0.0);
}


//Imprime os raios e os angulos entre os int neighbors primeiros vizinhos.
void geometry(int neighbors)
{

    double atomindex[n][neighbors];

    for (int i = 0; i < n ; i++)
    {
        gsl_vector * vectOrdered = gsl_vector_calloc(n);
        gsl_vector * vect= gsl_vector_calloc(n);

        for (int j = 0; j < n ; j++)
        {
            double data = round(1000.0*getDistance(i,j))/1000.0;
```

```
            gsl_vector_set(vectOrdered,j,data);
            gsl_vector_set(vect,j,data);
        }

        gsl_sort_vector(vectOrdered);

        int idx = 0;
        for (int j = 1; j <= neighbors ; j++)
        {

            while ( gsl_vector_get(vectOrdered,j) != gsl_vector_get(vect,idx))
            {
                idx++;

                if (idx == n)
                {
                    idx = 0;
                }
            }

            atomindex[i][j-1] = idx;
            cout << "Atom " << i+1 << " Vizinho " << j << " raio " << gsl_vector_get(vect,idx) << " bohrs\n";

            idx++;
            if (idx == n)
            {
                idx = 0;
            }
        }

        gsl_vector_free(vect);
        gsl_vector_free(vectOrdered);
    }


    for (int i = 0; i < n ; i++)
    {

        for (int l = 0; l < neighbors ; l++)
        {
            for (int m = 0; m < l ; m++)
            {
                Vector3D vect1(getAtom(i).getX()-getAtom(atomindex[i][l]).getX(),getAtom(i).getY()-getAtom(atomindex[i][l
                    ]).getY(),getAtom(i).getZ()-getAtom(atomindex[i][l]).getZ());

                Vector3D vect2(getAtom(i).getX()-getAtom(atomindex[i][m]).getX(),getAtom(i).getY()-getAtom(atomindex[i][m
                    ]).getY(),getAtom(i).getZ()-getAtom(atomindex[i][m]).getZ());

                cout << "Atom " << i+1 << " AnguloVizinhos(" << l+1 << "," << m+1 << ") " << 360.0*acos(vect1.dotProduct(
                    vect2)/(vect1.Norm()*vect2.Norm()))/(2.0*PI) << "\n";
            }
        }
    }
}

//Possiveis configuracoes iniciais:

//Inicializa as coordenadas dos atomos numa superficie esferica de raio
//double raio
void initializeSpherical(double raio)
{
    double x,y,z;
    for (int i = 0 ; i < n ; i++)
    {
        x = 2.0*(LIBNUM::RngInt(2)-0.5)*raio*LIBNUM::RngDouble();
        y = 2.0*(LIBNUM::RngInt(2)-0.5)*raio*LIBNUM::RngDouble();
        z = 2.0*(LIBNUM::RngInt(2)-0.5)*sqrt(raio*raio-y*y-x*x);


        while ( (x*x+y*y) > raio*raio)
        {
            x = 2.0*(LIBNUM::RngInt(2)-0.5)*raio*LIBNUM::RngDouble();
            y = 2.0*(LIBNUM::RngInt(2)-0.5)*raio*LIBNUM::RngDouble();
            z = 2.0*(LIBNUM::RngInt(2)-0.5)*sqrt(raio*raio-y*y-x*x);
        }
        atom[i].setCoordinates(x, y, z);
    }
}

//Inicializa as coordenadas dos atomos num cubo de lado double side
void initializeBox(double side)
{
    double x,y,z;
    for (int i = 0 ; i < n ; i++)
    {
        x = side*LIBNUM::RngDouble();
        y = side*LIBNUM::RngDouble();
        z = side*LIBNUM::RngDouble();

        atom[i].setCoordinates(x, y, z);
    }
}

//Inicializa as coordenadas dos atomos numa linha de comprimento double
//length
void initializeChain(double length)
{
    double x,y,z;
    for (int i = 0 ; i < n ; i++)
```

```
    {

        x = length*i/n;;
        y = 0.0;
        z = 0.0;

        atom[i].setCoordinates(x, y, z);
    }
}

//Inicializa as coordenadas dos atomos num anel de raio double raio
void initializeRing(double raio)
{
    double x,y,z,tetta;
    for (int i = 0 ; i < n ; i++)
    {
        tetta = i*2.0*PI/n;

        x = raio*sin(tetta);
        y = raio*cos(tetta);
        z = 0.0;

        atom[i].setCoordinates(x, y, z);
    }
}

//Inicializa as coordenadas dos atomos num dodecaedro com distancia ao
//primeiro vizinho double r. Tem de ser utilizado com um agregado de 20
//atomos.
void initializeDodecahedron(double r)
{
    double gamma = (1.0+sqrt(5.0))/2.0;
    double gamma_inv = 2.0/(1.0+sqrt(5.0));

    r = r*gamma/2.0;

    atom[0].setCoordinates(r,r,r);
    atom[1].setCoordinates(r,r,-r);
    atom[2].setCoordinates(r,-r,r);
    atom[3].setCoordinates(r,-r,-r);
    atom[4].setCoordinates(-r,r,r);
    atom[5].setCoordinates(-r,r,-r);
    atom[6].setCoordinates(-r,-r,r);
    atom[7].setCoordinates(-r,-r,-r);

    atom[8].setCoordinates(0.0,r*gamma_inv,r*gamma);
    atom[9].setCoordinates(0.0,r*gamma_inv,-r*gamma);
    atom[10].setCoordinates(0.0,-r*gamma_inv,r*gamma);
    atom[11].setCoordinates(0.0,-r*gamma_inv,-r*gamma);

    atom[12].setCoordinates(r*gamma_inv,r*gamma,0.0);
    atom[13].setCoordinates(r*gamma_inv,-r*gamma,0.0);
    atom[14].setCoordinates(-r*gamma_inv,r*gamma,0.0);
    atom[15].setCoordinates(-r*gamma_inv,-r*gamma,0.0);

    atom[16].setCoordinates(r*gamma,0.0,r*gamma_inv);
    atom[17].setCoordinates(r*gamma,0.0,-r*gamma_inv);
    atom[18].setCoordinates(-r*gamma,0.0,r*gamma_inv);
    atom[19].setCoordinates(-r*gamma,0.0,-r*gamma_inv);

}

//Inicializa as coordenadas dos atomos num icosaedro truncado (gaiola de
//C60) com distancia ao primeiro vizinho double r. Tem de ser utilizado com
//um agregado de 60 atomos
void intializeTruncatedIcosahedron(double r)
{
    double gamma = (1.0+sqrt(5.0))/2.0;
    r = r/2.0;

    atom[0].setCoordinates(0.0,r,3.0*r*gamma);
    atom[1].setCoordinates(0.0,r,-3.0*r*gamma);
    atom[2].setCoordinates(0.0,-r,3.0*r*gamma);
    atom[3].setCoordinates(0.0,-r,-3.0*r*gamma);

    atom[4].setCoordinates(r,3.0*r*gamma,0.0);
    atom[5].setCoordinates(r,-3.0*r*gamma,0.0);
    atom[6].setCoordinates(-r,3.0*r*gamma,0.0);
    atom[7].setCoordinates(-r,-3.0*r*gamma,0.0);

    atom[8].setCoordinates(3.0*r*gamma,0.0,r);
    atom[9].setCoordinates(3.0*r*gamma,0.0,-r);
    atom[10].setCoordinates(-3.0*r*gamma,0.0,r);
    atom[11].setCoordinates(-3.0*r*gamma,0.0,-r);

    atom[12].setCoordinates(r*2.0,r*(1.0+2.0*gamma),r*gamma);
    atom[13].setCoordinates(r*2.0,r*(1.0+2.0*gamma),-r*gamma);
    atom[14].setCoordinates(r*2.0,-r*(1.0+2.0*gamma),r*gamma);
    atom[15].setCoordinates(r*2.0,-r*(1.0+2.0*gamma),-r*gamma);
    atom[16].setCoordinates(-r*2.0,r*(1.0+2.0*gamma),r*gamma);
    atom[17].setCoordinates(-r*2.0,r*(1.0+2.0*gamma),-r*gamma);
    atom[18].setCoordinates(-r*2.0,-r*(1.0+2.0*gamma),r*gamma);
    atom[19].setCoordinates(-r*2.0,-r*(1.0+2.0*gamma),-r*gamma);

    atom[20].setCoordinates(r*(1.0+2.0*gamma),r*gamma,r*2.0);
    atom[21].setCoordinates(r*(1.0+2.0*gamma),r*gamma,-r*2.0);
    atom[22].setCoordinates(r*(1.0+2.0*gamma),-r*gamma,r*2.0);
    atom[23].setCoordinates(r*(1.0+2.0*gamma),-r*gamma,-r*2.0);
    atom[24].setCoordinates(-r*(1.0+2.0*gamma),r*gamma,r*2.0);
```

```
        atom[25].setCoordinates(-r*(1.0+2.0*gamma),r*gamma,-r*2.0);
        atom[26].setCoordinates(-r*(1.0+2.0*gamma),-r*gamma,r*2.0);
        atom[27].setCoordinates(-r*(1.0+2.0*gamma),-r*gamma,-r*2.0);

        atom[28].setCoordinates(r*gamma,r*2.0,r*(1.0+2.0*gamma));
        atom[29].setCoordinates(r*gamma,r*2.0,-r*(1.0+2.0*gamma));
        atom[30].setCoordinates(r*gamma,-r*2.0,r*(1.0+2.0*gamma));
        atom[31].setCoordinates(r*gamma,-r*2.0,-r*(1.0+2.0*gamma));
        atom[32].setCoordinates(-r*gamma,r*2.0,r*(1.0+2.0*gamma));
        atom[33].setCoordinates(-r*gamma,r*2.0,-r*(1.0+2.0*gamma));
        atom[34].setCoordinates(-r*gamma,-r*2.0,r*(1.0+2.0*gamma));
        atom[35].setCoordinates(-r*gamma,-r*2.0,-r*(1.0+2.0*gamma));

        atom[36].setCoordinates(r*1.0,r*(2.0+gamma),r*2.0*gamma);
        atom[37].setCoordinates(r*1.0,r*(2.0+gamma),-r*2.0*gamma);
        atom[38].setCoordinates(r*1.0,-r*(2.0+gamma),r*2.0*gamma);
        atom[39].setCoordinates(r*1.0,-r*(2.0+gamma),-r*2.0*gamma);
        atom[40].setCoordinates(-r*1.0,r*(2.0+gamma),r*2.0*gamma);
        atom[41].setCoordinates(-r*1.0,r*(2.0+gamma),-r*2.0*gamma);
        atom[42].setCoordinates(-r*1.0,-r*(2.0+gamma),r*2.0*gamma);
        atom[43].setCoordinates(-r*1.0,-r*(2.0+gamma),-r*2.0*gamma);

        atom[44].setCoordinates(r*(2.0+gamma),r*2.0*gamma,r*1.0);
        atom[45].setCoordinates(r*(2.0+gamma),r*2.0*gamma,-r*1.0);
        atom[46].setCoordinates(r*(2.0+gamma),-r*2.0*gamma,r*1.0);
        atom[47].setCoordinates(r*(2.0+gamma),-r*2.0*gamma,-r*1.0);
        atom[48].setCoordinates(-r*(2.0+gamma),r*2.0*gamma,r*1.0);
        atom[49].setCoordinates(-r*(2.0+gamma),r*2.0*gamma,-r*1.0);
        atom[50].setCoordinates(-r*(2.0+gamma),-r*2.0*gamma,r*1.0);
        atom[51].setCoordinates(-r*(2.0+gamma),-r*2.0*gamma,-r*1.0);

        atom[52].setCoordinates(r*2.0*gamma,r*1.0,r*(2.0+gamma));
        atom[53].setCoordinates(r*2.0*gamma,r*1.0,-r*(2.0+gamma));
        atom[54].setCoordinates(r*2.0*gamma,-r*1.0,r*(2.0+gamma));
        atom[55].setCoordinates(r*2.0*gamma,-r*1.0,-r*(2.0+gamma));
        atom[56].setCoordinates(-r*2.0*gamma,r*1.0,r*(2.0+gamma));
        atom[57].setCoordinates(-r*2.0*gamma,r*1.0,-r*(2.0+gamma));
        atom[58].setCoordinates(-r*2.0*gamma,-r*1.0,r*(2.0+gamma));
        atom[59].setCoordinates(-r*2.0*gamma,-r*1.0,-r*(2.0+gamma));

    }
};
```