Possible Permutations of the Carbon-Carbon Distance in Annulen Molecules

Clayton Dagler
University of California, Davis
Division of Mathematics

December 21, 1998

Abstract

This article investigates the pattern of inneratomic distances in annulenes by taking a given set of distances and finding the ordering of these distances which minimizes its energy (given by the Hückel model). Our hypothesis is that there exists a specific permutation π_{max} that minimizes this energy function. The only possible periods of π_{max} are 1, 2, or the number of carbons. To determine the pattern of the inneratomic distances we have developed and implemented computer programs that calculate the energy function produced by π_{max} and compare it to the values produced by the other permutations. Our results are entirely consistent with the hypothesis. Thus the only possible periodicities of the inneratomic distances in annulenes are 1, 2, or the number of carbons. This results gives us a better understanding on the structure of annulenes and the behavior of the electrons which orbit the carbons in this class of molecules.

Introduction:

Determining the interatomic distances in a molecule is an important question in quantum chemistry. Quantum physics has been used to gain an answer to this question. This paper considers the interatomic distances in annulene molecules, i.e., ring-shaped molecules $(CH)_{2k}$, $k = 2, 3, \cdots$. An example of this class of molecules is Benzene (k=3), which contains 6 carbon and 6 hydrogen atoms (fig 1); however, this article only considers the case when k is even, i.e. when the number of carbons is a multiple of 4.

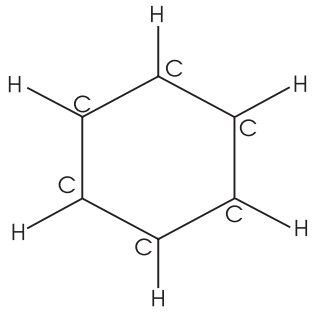

fig 1

It is an experimental fact that, for even k, the carbon-carbon distances are not all equal; rather, they form a pattern of alternating long and short bonds, this is termed dimerization. This does not imply that the long bonds and the short bonds are all equal; in fact, it is unknown what configurations may occur. Our objective is to determine what periodicities of the carbon-carbon distances are possible. Within the model we studied (the Hückel model) we found evidence that the only possibilities are:

i) The carbon-carbon distances are equal.

ii) The distances are arranged in a periodic pattern of period 2.

iii) The pattern is completely irregular, i.e., the period is the number of carbons in the molecule.

To determine the interatomic distances, it is well established that one must minimize the energy of the molecules; therefore, it is necessary to have an expression of energy as a function of the interatomic distances. In the Hückel model, this function consists of two terms, given by

$E(\boldsymbol{d}) = E_1(\boldsymbol{d}) + E_2(\boldsymbol{d})$

where d_i denotes the distance between the i^{th} and $i^{th} + 1^{st}$ carbon atom, and $\boldsymbol{d} = (d_1, d_2, \cdots, d_n)$ for $n = 2 \cdot k$. E_1 represents the energy of the n pi electrons in the $(CH)_n$ molecule and is given by the negative of the L^1 norm of the tridiagonal matrix

$$\boldsymbol{A}_{\boldsymbol{\delta}} = \begin{bmatrix} 0 & \delta_1 & 0 & \cdots & 0 & \delta_n \\ \delta_1 & 0 & \delta_2 & 0 & \cdots & 0 \\ 0 & \delta_2 & 0 & \ddots & \ddots & \vdots \\ \vdots & 0 & \ddots & \ddots & \delta_{n-2} & 0 \\ 0 & \cdots & \ddots & \delta_{n-2} & 0 & \delta_{n-1} \\ \delta_n & 0 & \cdots & 0 & \delta_{n-1} & 0 \end{bmatrix} \tag{1}$$

where $\boldsymbol{\delta} = (\delta_1, \delta_2, \cdots, \delta_n)$. Recall that the L^1 norm of a hermitian matrix $\boldsymbol{A}_{\boldsymbol{\delta}}$ is the sum of the absolute values of its eigenvalues:

$$\|\boldsymbol{A}_{\boldsymbol{\delta}}\|_1 = \sum_{i=1}^{n} |\lambda_n|$$

where $\lambda_1, \lambda_1, \cdots, \lambda_n$ are the eigenvalues of $\boldsymbol{A}_{\boldsymbol{\delta}}$. The non-vanishing matrix elements are a function of the interatomic distances. This function is well approximated by a decreasing exponential function

$$\delta_i = f(d_i) = c \cdot e^{-d_i}, \qquad \text{where} \qquad d_i > 0 \tag{2}$$

The graph of this function is given in fig 2.

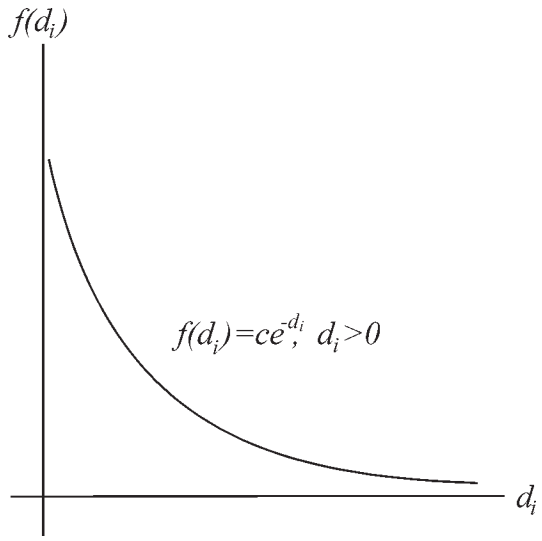

fig 2

E_2 represents the energy of the sigma bonds between neighboring carbon atoms, given by

$$E_2 = \sum_{i=1}^{n} V(d_i) \tag{3}$$

where V is a function of the type $V(d_i) = c \cdot (d_i \Leftrightarrow d_0)^2$. See fig 3.

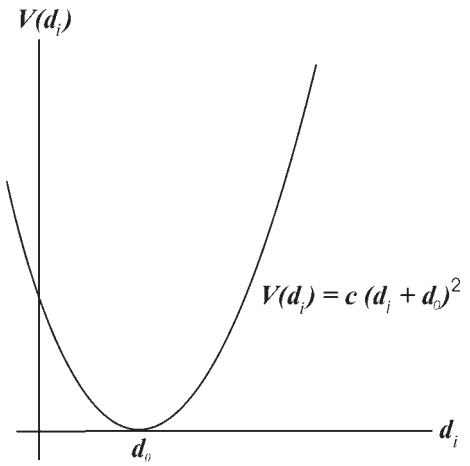

fig 3

The focus of this article is to test the conjecture

$$E(\boldsymbol{d}_\pi) \leq E(\boldsymbol{d}) \tag{4}$$

where $\boldsymbol{d}_\pi$ is a particular permutation of $\boldsymbol{d}$. An intuitive way to think of this permutation is to consider a polygon of n sides. Let the set $S = \{s_1, s_2, \cdots, s_n\}$ where $s_1 = d_1, s_2 = d_2, \cdots, s_n = d_n$. Obtaining this permutation involves the following steps:

Step 1) Remove the smallest element from the set S and assign it to one of the sides of the polygon, then remove the two largest elements from the set and assign it to the two sides of the polygon that are connected to the smallest side.

Step 2) Remove the smallest element from the remaining set and assign it to the side next to the largest available side in the polygon. Repeat this one more time with the new set.

Step 3) Remove the largest element from the remaining set and assign it to the side next to the smallest available side in the polygon. Repeat this one more time with the new set.

Step 4) Repeat steps 2 and 3 until there is just one element left in the set, then remove this element from the set and assign it to the only available side of the polygon.

The order in which the sides appear in this polygon is the permutation $\boldsymbol{d}_{\boldsymbol{\pi}}$ in equation 4. A more convenient way to express this permutation in the context of programming is to first order the set S in increasing order. Thus, $s_1 \leq s_2 \leq \cdots \leq s_n$. Then for each $i = 1, 2, \ldots \frac{n}{4}$, leave unchanged all the positions s_{2i} and s_{n-2i-1}, and interchange s_{2i+1} with s_{n-2i+1}. See the GetMaxPermut algorithm in the appendix. For example, if $n = 8$ and $s_1 \leq s_2 \leq \cdots \leq s_8$, then this permutation is

$$s_1, s_7, s_3, s_5, s_4, s_6, s_2, s_8$$

For convenience, denote this permutation by π_{max}.

Definition 1: $\mathbf{c}$ is a **shift** of $\mathbf{d}$ if and only if there exists $k = 1, 2, \cdots, n$ such that $c_{i-k(mod\ n)} = d_i$ for all $i = 1, 2, \cdots, n$. We say that $\mathbf{c}$ is the k'th shift of $\mathbf{d}$.

Definition 2: $\mathbf{c}$ is the **reflection** of $\mathbf{d}$ if and only if $c_{n-i+1} = d_i$ for all $i = 1, 2, \cdots, n$.

Definition 3: $\mathbf{c}$ and $\mathbf{d}$ are **equivalent**, denoted $\mathbf{c} \approx \mathbf{d}$, if and only if $\mathbf{c}$ can be obtained from $\mathbf{d}$ by applying a shift, reflection, or a combination of the two.

Definition 1 represents taking the molecule and rotating it clockwise or counter clockwise, and Definition 2 represents the mirror image of the molecule.

One can see that any permutation of $\mathbf{d}$ does not affect E_2. Therefore, we only have to test the conjecture for E_1. Moreover, since f defined in equation 2 is a strictly monotone function, a permutation of $(d_1, d_2, \cdots, d_n)$ is equivalent to a permutation of the $(f(d_1), f(d_2), \cdots, f(d_n)) = (\delta_1, \delta_2, \cdots, \delta_n)$. This amounts to testing the following inequality:

$$\|\boldsymbol{A}_{\boldsymbol{\delta}\,\pi_{max}}\|_1 \geq \|\boldsymbol{A}_{\boldsymbol{\delta}\,\pi}\|_1 \tag{5}$$

where $\boldsymbol{\delta}_{\pi_{max}}$ is π_{max} of $\boldsymbol{\delta}$, and $\boldsymbol{\delta}_\pi$ is any permutation of $\boldsymbol{\delta}$. We have performed experiments which show that this inequality holds. These experiments have also shown that the only time the equality holds in (5) is when $\boldsymbol{\delta}_{\pi_{max}} \approx \boldsymbol{\delta}_\pi$.

It is possible to have two different permutations $\boldsymbol{\delta}$ and $\boldsymbol{\delta}_\pi$ such that $\|\boldsymbol{A}_{\boldsymbol{\delta}}\|_1 = \|\boldsymbol{A}_{\boldsymbol{\delta}\,\pi}\|_1$. To see this, let $\boldsymbol{\delta}_\pi = ((\delta_1)_\pi, (\delta_2)_\pi, \cdots, (\delta_n)_\pi)$ be a permutation of $\boldsymbol{\delta} = (\delta_1, \delta_2, \cdots, \delta_n)$. Now take two polygons of size n, and let the sides of the first polygon have the same values and order of $\boldsymbol{\delta}$, and if the same is done to the second polygon with $\boldsymbol{\delta}_\pi$ and the polygons are a duplicate of each other, then $\|\boldsymbol{A}_{\boldsymbol{\delta}}\|_1 = \|\boldsymbol{A}_{\boldsymbol{\delta}\,\pi}\|_1$. This can occur when $\boldsymbol{\delta}_\pi \approx \boldsymbol{\delta}$, see theorem 2.

Theoren 1: If $\mathbf{A}$ and $\mathbf{B}$ are similar matrices, then $\|\mathbf{A}\|_1 = \|\mathbf{B}\|_1$.

Proof: First we need to show that if $\mathbf{A}$ and $\mathbf{B}$ are similar matrices, then they have the same eigenvalues. Following [1], $\mathbf{A}$ and $\mathbf{b}$ are similar implies that there exists a matrix $\mathbf{P}$ such that $\mathbf{A} = \mathbf{P}^{-1} \cdot \mathbf{B} \cdot \mathbf{P}$. Now,

$$\begin{aligned}
|\lambda \cdot \mathbf{I} \Leftrightarrow \mathbf{A}| &= |\lambda \cdot \mathbf{I} \Leftrightarrow \mathbf{P}^{-1} \cdot \mathbf{B} \cdot \mathbf{P}| \\
&= |\mathbf{P}^{-1} \cdot \lambda \cdot \mathbf{P} \Leftrightarrow \mathbf{P}^{-1} \cdot \mathbf{B} \cdot \mathbf{P}| \\
&= |\mathbf{P}^{-1} \cdot (\lambda \cdot \mathbf{I} \Leftrightarrow \mathbf{B}) \cdot \mathbf{P}| \\
&= |\mathbf{P}^{-1}| \cdot |\mathbf{P}| \cdot |\lambda \cdot \mathbf{I} \Leftrightarrow \mathbf{B}| \\
&= |\mathbf{P}^{-1} \cdot \mathbf{P}| \cdot |\lambda \cdot \mathbf{I} \Leftrightarrow \mathbf{B}| \\
&= |\mathbf{I}| \cdot |\lambda \cdot \mathbf{I} \Leftrightarrow \mathbf{B}| \\
&= |\lambda \cdot \mathbf{I} \Leftrightarrow \mathbf{B}|
\end{aligned}$$

Hence, the characteristic polynomial of $\mathbf{A}$ and $\mathbf{B}$ are equivalent, i.e. they have the same eigenvalues. Now, we have

$$\|\mathbf{A}\|_1 = \sum_{i=1}^{n} |\lambda_i| = \|\mathbf{B}\|_1 \qquad \text{q.e.d.}$$

Theorem 2: If $\boldsymbol{\delta}_\pi \approx \boldsymbol{\delta}$, then $\|\boldsymbol{A}_{\boldsymbol{\delta}\,\pi}\|_1 = \|\boldsymbol{A}_{\boldsymbol{\delta}}\|_1$.

Proof: Define $\mathbf{S}_i = [\ e_{i-n+1} \ \cdots \ e_n \ \ e_1 \ \cdots \ e_{i-1}\ ]$ and $\mathbf{E} = [\ e_n \ \ e_{n-1} \ \cdots \ e_1\ ]$ for $e_1 = [\ 1 \ \ 0 \ \cdots \ 0\ ]^t$, $e_2 = [\ 0 \ \ 1 \ \ 0 \ \cdots \ 0\ ]^t$, $\cdots$, $e_n = [\ 0 \ \cdots \ 0 \ \ 1\ ]^t$ where e_i is n by 1. If $\boldsymbol{\delta}_\pi$ is the i'th shift of $\boldsymbol{\delta}$ then $\boldsymbol{A}_{\boldsymbol{\delta}\,\pi} = \mathbf{S}_i^{-1} \cdot \boldsymbol{A}_{\boldsymbol{\delta}} \cdot \mathbf{S}_i$. Furthermore, if $\boldsymbol{\delta}_\pi$ is a reflection of $\boldsymbol{\delta}$, then $\boldsymbol{A}_{\boldsymbol{\delta}\,\pi} = \mathbf{E}^{-1} \cdot \boldsymbol{A}_{\boldsymbol{\delta}} \cdot \mathbf{E}$. Finally, if $\boldsymbol{\delta}_\pi$ is a combination of the two, then $\boldsymbol{A}_{\boldsymbol{\delta}\,\pi} = (\mathbf{S}_i \cdot \mathbf{E})^{-1} \cdot \boldsymbol{A}_{\boldsymbol{\delta}} \cdot (\mathbf{S}_i \cdot \mathbf{E})$. Thus, $\boldsymbol{A}_{\boldsymbol{\delta}\,\pi}$ and $\boldsymbol{A}_{\boldsymbol{\delta}}$ similar matrices and by theorem 1 $\|\boldsymbol{A}_{\boldsymbol{\delta}\,\pi}\|_1 = \|\boldsymbol{A}_{\boldsymbol{\delta}}\|_1$ q.e.d.

Definition 4: The period of $\boldsymbol{d}$ is the smallest integer $p \geq 1$ such that $d_i = d_{i+p(mod\ n)}$ for all $i = 1, 2, \cdots, n$.

The conjecture found in equation 4 implies that no periods other than 1, 2, or n can occur for any $\mathbf{d} \approx \mathbf{d}_{\pi_{max}}$ which is stated in the following theorem.

Theorem 3: If $\mathbf{d} \approx \mathbf{d}_{\pi_{max}}$, then the period of $\mathbf{d}$ is 1,2, or n.

Proof: The two trivial cases occur when all of the values of $\mathbf{d}$ are equivalent or distinct, causing $\mathbf{d}$ to have a period of 1 and n respectively. If there are two distinct values and half of the values are d_1 and the other values are d_2, then any equivalent order of $\mathbf{d}$ must be of the form $(d_1, d_2, \cdots, d_1, d_2)$ which is a period of two. On the other hand, if there are two distinct values and there are more d_1 values than d_2, an equivalent order of $\mathbf{d}$ would be $(d_1, d_2, \cdots, d_1, d_2, d_1, \cdots, d_1)$, which is a period of n. Finally, if there are three or more values with d_1 the smallest value and d_n the largest value, then an order of $\mathbf{d}$ would be $(d_1, d_2, \cdots, d_1, d_2, *, \cdots, *)$ where $(*, \cdots, *) \neq (d_1, d_2, \cdots, d_1, d_2)$ which is a period of n q.e.d.

Method:

Define the L^1_{max} to be the L^1 norm produced by π_{max} or an equivalent permutation. To see that L^1_{max} the is the largest L^1 norm, i.e. confirm equation 4, programs have been written that calculate the L^1 norms and compare the L^1_{max} to the L^1 norms produced by all the other permutations.

To calculate the L^1 norms, the Householder method is applied to the matrix, which converts any n by n symmetric matrix to tridiagonal form with n - 2 orthogonal transformations, then a QL algorithm with implicit shifts is applied to the new matrix to calculate the eigenvalues [2,3].

The program that tests (4) consists of two functions and a main routine: the first function creates the data; the second produces the permutations, and the main routine calculates the L^1 norms and writes to a file every L^1 norm that is close to L^1_{max}. To confirm that this program is performing correctly, smaller

programs were created, which took the results of these two functions and wrote the results to files. Then each one of these files were evaluated.

There are actually three different functions that produce the data called GeneralData, RealData and HalfData. These functions produce each set of data one at a time and return 1 (not done) if there are more data values to get, otherwise the functions return 0 (done). To obtain all the sets of data, just run one of the data functions inside a while loop; in C++ this routine would be in the form:

```
while(DataRoutine)                    // DataRoutine = GeneralData, RealData, or HalfData
{
        // Do something with the data.
}
```

GeneralData generates the sets of data by initially setting all the values to the smallest value, which is the first set of data. To create the rest of the data, this routine adds the increment value to the data equating all the values to the right of the value that was just increased–rearranging each new set of data in the order of π_{max} until all the values are the largest value. The reason for setting all the values equal to the right of the value that was just increased is to avoid creating data that only differ by a permutation, which would just be recalculated. For example: if the smallest value is 1, the largest value is 3, the increment value is 1, and the size is 4, then the results would be:

Not in π_{max} order	In π_{max} order
1 1 1 1	1 1 1 1
1 1 1 2	1 1 1 2
1 1 1 3	1 1 1 3
1 1 2 2	1 2 1 2
1 1 2 3	1 2 1 3
1 1 3 3	1 3 1 3
1 2 2 2	1 2 2 2
⋮	⋮
3 3 3 3	3 3 3 3

The algorithm of this function is found in the appendix as GeneralData.

The second data function, RealData, models what happens in nature by generating data clustered around two values, denote them v_1 and v_2. The basic idea behind this function is that it uses GeneralData on each half of the data. Initially, the function starts out with half of the values a distance Δ less than v_1, call this set of values s_1, and the other half of the values a distance Δ less than v_2, call this set of values s_2. It then applies GeneralData to s_1. Each time a value in s_1 is increased, GeneralData is applied to s_2, then the values in s_2 are reset to their initial values. After each time a new set of data is generated, it is written in the order of π_{max}. This routine terminates when the values in s_1 are all $v_1 + \Delta$ and all the values in s_2 are $v_2 + \Delta$.

Another set of data is created by using GeneralData, called HalfData. To create this set of data,

GeneralData is used to create half of the data and then ones are placed between all the data values created by GeneralData.

There are two different functions to produce the permutations, one called SubPermut and the other MinPermut. Like the data functions, these functions produce one permutation at a time returning 1 (not done) if there are more permutation to get, otherwise the functions return 0 (done). Both of these functions use a modified version of the function NextPermut found in [4] which takes a set of integers and rewrites them in a different order; this algorithm is in the appendix as NextPermut. The permutations produced by SubPermut are the permutations of the numbers 1 through n-1 with the first value of every permutation 0. This was done to avoid producing permutations that only differ by a shift.

The second permutation function, MinPermut, finds $((n/2 \Leftrightarrow 1)!) \cdot ((n/2)!)$ different permutations with the first value of every permutation 0. Let permut1, permut2 be two sets of size $\frac{n}{2}$, where the elements of permut1 are initially $0, 1, \cdots, \frac{n}{2} \Leftrightarrow 1$ and the elements of permut2 are initially $\frac{n}{2}, \frac{n}{2}+1, \cdots, n \Leftrightarrow 1$. The first permutation is the first element of permut1, the first element of permut2, the second element of permut1, $\cdots$. Then to get the permutations thereafter, if the elements of permut2 are not in the order of the last permutation, then it permutes the elements in permut2. If all the elements of permut1 are not in the order of the last permutation, it resets the elements of permut2 to $\frac{n}{2}, \frac{n}{2}+1, \cdots, n \Leftrightarrow 1$ and permutes all but the first elements in permut1, otherwise all the permutation are found. If all the permutations were not found, the next permutation is in the order of the first element of permut1, the first element of permut2, the second element of permut1, $\cdots$.

The program that looks for permutations that are close to or greater than the assumed maximum permutation uses one of the data functions, and the assumed maximum L^1 norm for each set of data. Then this routine uses one of the permutation routines to find all the L^1 norms produced by the necessary permutations of the current set of data. A permutation is written to a file if it is not equivalent to π_{max}, which may occur only when there are two elements, d_1 and d_2, in the current set of data such that $d_1 = d_2$, and if its L^1 norm is close to the L^1_{max} of the current set of data, i.e. if $(L^1_{max}) \Leftrightarrow (L^1) < \epsilon$ for $\epsilon = 10^{-i}, i = 1, 2, \cdots$. A routine was created which checks to see if a certain permutation is π_{max}, and the algorithm is given in the appendix as IsMaxPermut.

If one would choose ϵ to be less than the precision of calculation for the L^1 norms the results would be meaningless. To avoid this, a program was created which estimates the precision of these calculations. This is done by taking all the sets from either of the data functions, denote these sets by $\{s_1, s_2, \cdots, s_n\}$, then for each s_i, $i = 1, 2, \cdots, n$ $\|A_{s_i}\|_1$ is calculated as well as every $\|A_{t_j}\|_1$ where $t_j \in \{t : t \text{ is a shift of } s_i\}$ for $j = 1, 2, \cdots, n$. Then the greatest difference between $\|A_{t_i}\|_1$ and $\|A_{t_j}\|_1$ for every set of data is written to the screen, and ϵ is chosen to be larger than this difference.

Results:

Several data sets have been tested for the routines GeneralData, RealData, and HalfData and some of them are displayed in table 1, and no permutation was found to produce a larger L^1 norm then L^1_{max}.

General

size	smallest	largest	step
4	1	20	0.1
8	1	12	0.1
12	1	8	0.1

Real

size	1^{st} pt.	2^{ed} pt.	Δ	step
4	1	1.01, 1.02, $\cdots$, 1.16	0.08	0.01
8	1	1.01, 1.02, $\cdots$, 1.16	0.05	0.01

Half

size	smallest	largest	step
8	0.8	0.95	0.01

table 1

Due to Theorem 4, one only need to check values where the ratio of the first value and the second value or the first point and the second point differ.

Theorem 4: If $\alpha \geq 0$, and N is the L^1 norm of $\mathbf{A}$, then $\alpha \cdot N$ is the L^1 norm of $\alpha \cdot \mathbf{A}$.

Proof: First we need to show that if λ is an eigenvalue of $\mathbf{A}$, then $\alpha \cdot \lambda$ is an eigenvalue of $\alpha \cdot \mathbf{A}$. Following [5], if λ is an eigenvalue of A, then there is a vector $\mathbf{x}$ such that,

$$\mathbf{A} \cdot \mathbf{x} = \lambda \cdot \mathbf{x}$$
$$\Leftrightarrow \alpha \cdot \mathbf{A} \cdot \mathbf{x} = \alpha \cdot \lambda \cdot \mathbf{x}$$

Hence, $\alpha \cdot \lambda$ is an eigenvalue of $\alpha \cdot \mathbf{A}$. Now, with $N = L^1$ norm of $\mathbf{A}$, let $M = L^1$ norm of $\alpha \cdot \mathbf{A}$. Then,

$$M = \sum_{i=1}^{n} |\alpha \cdot \lambda_i| = \sum_{i=1}^{n} |\alpha| \cdot |\lambda_i| = \alpha \cdot \sum_{i=1}^{n} |\lambda_i| = \alpha \cdot N$$

Thus, $\alpha \cdot N$ is the L^1 norm of $\alpha \cdot \mathbf{A}$ q.e.d.

Conclusion:

The numerical values that we have tested confirm that the maximum L^1 norm is obtained only by permutations which are equivalent to π_{max}. Thus the only permutations that minimize the Hückel model's energy function are the permutations which are equivalent to π_{max}. By Theorem 3, this property implies that the only possible permutations of the interatomic distances in annulene molecules is 1, 2, or the number of carbons.

Acknowledgdments:

I would like to give special thanks to the McNair Scholars Program as well as the Research Experiences for Undergraduates program of the National Science Foundation (grant # DMS-9706599) for the funding of this research project and to my mentor, Professor Bruno Nachtergaele, who was a great help and a pleasure to work with.

Appendix

GetMaxPermut:
Get two array of integers of size n: **a**, **temp**
set temp = 1, $\cdots$, n

for i=1 to n/4
    the 2i element of **a** = the 2i element of **temp**
    the (2i+1) element of **a** = the (size-2i-2) element of **temp**
    the (size-2i-2) element of **a** = the (2i+1) element of **temp**
    the (size-2i-1) element of **a** = the (size-2i-1) element of **temp**
Then **a** is in the order of π_{max}.

GeneralData:
get an integer: position initial equal to n.
get an array of numbers of size n: **v** with all elements initial equal to smallest value
let **vals** be the array that holds the data on return

if it is the first time called
    store the elements of **v** in **vals**
    return 1 (not done)
while position > 0
    If position = n
        if the last element < largest_val
            add the com_dif to the last element
            store the elements of **v** in **vals**
            return 1 (not done)
        else
            decrease position by 1
    else
        if the position element is less than the largest_val
            add the com_dif to the position element
            set all the values to the right of position equal to the value at position
            position = size
            store the elements of **v** in **vals**
            return 1 (not done)
        else
            decrease position by 1
return 0 (done)

NextPermut:
get integers: i, k, tmp, left, right
let **p** be the array that holds the new permutation on return.

k = n
while k $\geq$ 1 and the k^{th} element of **p** > the $k^{th} + 1^{st}$ element of **p**
    decrease k by 1
left = k+1
right = n
while left < right
    tmp = the left element of **p**
    the left element of **p** = the right element of **p**

the right element of **p** = the left element of **p**
increase left by 1
decrease right by 1
if $k < 0$
return 0 (done)
$i = k + 1$
while the i^{th} element of **p** < the k^{th} element of **p**
increase i by 1
tmp = ith element of **p**
ith element of **p** = the kth element of **p**
the kth element of **p** = tmp
return 1 (not done)

IsMaxPermut:
Get one array of integers of size n: **max_permute**
Get one array of numbers of size n: **elms**

Store the values of the permutation in question in **elms**.
Use the get max permute algorithm to store the maximum permit in **max_permute**
Call MaxOrder with **elms**, **max_permute**, n
if MaxOrder is true, return 1 (true)
Use the get max permute algorithm to restore the maximum permute in **max_permut**
reverse the order of **max_permute**
Call MaxOrder with **elms**, **max_permute**, n again
if MaxOrder is true, return 1 (true)
return 0 (false)

MaxOrder:
get an integer: is_max
get an array of integers of size n: **current_order**

for i=1 to n
is_max = 1 (true)
for j=1 to n
k^{th} element of **current_order** = j^{th} element of **elms**, where $k^{th} = j^{th}$ element of π_{max}
for j=2 to n
if $j^{th} \Leftrightarrow$1 element of **current_order** > j^{th} element of **current_order**
is_max = 0 (false)
break
if is_max = 1
return 1 (true)
shift the values in **max_permute** over 1
return 0 (false)

References

[1] William H. Press et al. *Numerical Recipes in C. New York*: Press Syndicate of the University of Cambridge, 1996.

[2] Wilkinson, J.H., and Reinsch, C. *Linear Algebra, vol. 2 of Handbook for Automatic Computation*. New York: Springer-Verlag, 1971.

[3] Goldberg, Jack L. *Matrix Theory with Applications*. McGraw-Hill: New York, 1991.

[4] Larson, R.E., and Edwards, B.H. *Elementary Linear Algebra*. Massachusetts: D. C. Heath and Company, 1991.

[5] Ammeraal, Leendert. *Algorithms and Data Structures in C++*. New York, John Wiley & Sons, Inc., 1996.