Calculo de los centros de N de las caras regulares en los sólidos platónicos

Raúl Santoy Flores, Alfredo Tlahuice Flores

Universidad Autónoma de Nuevo León
Facultad de Ciencias Físico Matemáticas

25 de mayo de 2022

1. Resumen

Las estructuras moleculares que se pueden formar en algún proceso natural o sintético, en dos o tres dimensiones, presentan figuras geométricas características por su simpleza de estudiar, como los polígonos regulares o los sólidos platónicos. En este trabajo se calcula los centros de las caras poligonales de los sólidos platónicos, para después avanzar en el calculo de las posiciones centrales del fullereno C_{60}, y así de forma didáctica sentar presentar precedentes en estos cálculos a través de un resultado ya establecido en la geometría analítica, que es la definición del punto medio de n cantidad de puntos. Se reporta una metodología y todos los detalles de cada caso particular que se presenta y como se pueden generalizar funciones vectoriales en Python para facilitar los cálculos y los algoritmos internos.

2. Objetivos

- Calcular el centro de todo polígono de cualquier estructura molecular
  a) Clasificar los polígonos encontrados en la molécula
  b) Determinar las distintas relaciones que puedan surgir

3. Centro de un polígono

El punto medio es el lugar entre dos puntos cualesquiera en el espacio cartesiano. Es decir, sea el punto $A = (1, 1, 1)$ y el punto $B = (2, 2, 2)$, entonces el punto medio P_m entre A y B es

$$P_m = \frac{A + B}{2} \tag{1}$$

Cuando tenemos un polígono, podemos calcular el centro por el mismo método del punto medio entro dos puntos, siendo en este caso para una cantidad contable de puntos que se puedan unir en una trayectoria cerrada.

En el caso de un triangulo se tienen tres puntos, entonces el punto medio de estos es

$$P_m = \frac{P_1 + P_2 + P_3}{3} \tag{2}$$

Para un cuadrado

$$P_m = \frac{P_1 + P_2 + P_3 + P_4}{4} \tag{3}$$

En el caso general, podemos calcular el centro de un polígono de n puntos de la siguiente forma

$$P_m = \frac{\sum_{\forall i} P_i}{n} \tag{4}$$

Se calculá el centro de un pentágono, donde las posiciones de los vértices circunscriben a un circulo de radio unitario. El código se elaboro en Python y se utilizó la paquetería de POV-Ray

Primero importamos las paqueterías a usar

```
import numpy as np
from vapory import *
```

Cargamos el entorno en POV-Ray, incluyendo las paqueterías colors y textures, configuramos la cámara que es el punto desde donde se va a percibir la escena, asignamos un color al fondo del entorno y definimos las fuentes de luz.

```
include = ['colors.inc', 'textures.inc']
camera = Camera( 'location', [0, 0, 4],
                 'look_at',  [0, 0, 0])
bg = Background('color', [1,1,1])
L1 = LightSource([0,10000,0], 'color','Goldenrod')
L2 = LightSource([10000,10000,10000], 'color', 'Goldenrod')
```

Listing 1: Set up environment

render es una lista de objetos que mediante la función Scene nos devuelve un renderizado de esta lista de objetos, que en primer lugar siempre debe de incluir el

- Background
- LightSource

```
render =  [bg,L1,L2]
```

Se define una función rotV que devuelve un vector rotado un cierto angulo dado un vector inicial.

```
def rotV(th, v):
    rot = np.array([[cos(th), -sin(th)],[sin(th), cos(th)]])
    return np.matmul(rot, v)
```

En este bloque de código se emplea para calcular las posiciones de los vértices de un pentágono circunscrito en un circulo de radio $\sqrt{2}$. En cada iteración se agregan en la lista render las esferas (Sphere) que se localizan en estas posiciones.

```
p0 = np.array([1,1])
pos = []

for i in range(0,5):
    ang = np.deg2rad(360*i/5)
    p = rotV(ang, p0)
    pos.append(p)
    s = Sphere(p, 0.35, Texture( Pigment( 'color', 'Magenta')))
    render.append(s)
```

En este, se calculá el centro del pentágono por la *regla del punto medio entre N puntos*, en este caso 5, de acuerdo a las posiciones conseguidas en el bloque anterior.

```
c = np.array([0,0])
for p in pos:
    c = np.add(c,p)
```

Nuevamente renderizamos una esfera localizada en el centro del pentágono, es decir, en la posición obtenida en las lineas anteriores

```
s = Sphere(c.tolist(), 0.35, Pigment('color','Gold'))
render.append(s)
```

Mediante la función Scene se carga una escena mediante la configuración de los siguientes argumentos

- Cámara - camera
- Objetos - render
- Paqueterías - include

```
scene = Scene( camera, render, included = include )
```

El objeto scene se puede guardar en formato png e incluir algunas características de la imagen, como el ancho y altura, o incluso un suavizado de los pixeles.

```
scene.render('pentagono.png', width=600, height=500, antialiasing=0.001)
```

Por tanto, se calcula el centro de un polígono regular tan solo conociendo las posiciones de los vértices de este.

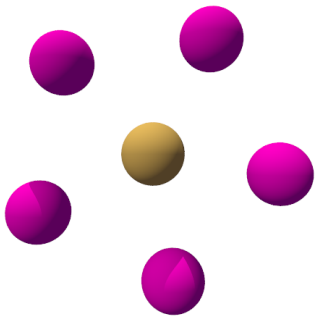

Figura 1: Centro de un pentágono, calculo de punto medio

4. Sólidos Platónicos

Para calcular el centro de cualquier figura geométrica de puntos se pude utilizar este método, pero también se puede utilizar para encontrar el centro de las caras poligonales que se puedan formar en cualquier estructura 3-Dimensional. Este es el caso de los sólidos platónicos, que cuentan con caras triangulares, cuadradas y pentagonales, los cuáles pueden servir como una primera aproximación del método.

La metodología que se sigues es, en primer lugar, calcular la medida de las aristas de las caras mediante el siguiente algoritmo Arista que mide la distancia entre todos los vértices y colecta las distancias más cortas que difieren por una diferencia 1×10^{-2} y calcula un promedio de estas. Después se emplea una combinación o permutación de posiciones según el numero de vértices de las caras, para poder determinar las caras de este y así mediante el calculo del punto medio, determinar el centro de estas.

	Tetraedro	Cubo	Octaedro	Dodecaedro	Icosaedro
Numero de caras	6	12	12	30	30
Numero de vértices	4	8	6	20	12
Caras concurrentes en cada vértice	3	3	4	3	5
Vértices contenidos en cada cara	3	4	3	5	3

Cuadro 1: Características geométricas de los sólidos platónicos

```
def Arista(pos):
    dprom, d = DistProm(pos)
    M = []
    m0 = np.min(d)
    m = m0
    while (abs(m0-m)<1E-2):
        idx = np.where(d == m)
        d = np.delete(d, idx)
        M.append(m)
        m = np.min(d)

    return np.mean(M)
```

Listing 2: Calculo de aristas de un solido platónico

Esta función utiliza la función DistProm que calcula la distancia entre todos los vértices de la estructura y nos devuelve una lista con estas medidas y un flotante del valor promedio de esta lista se define como sigue,

```
def DistProm(pos):
    d = []
    for p1 in pos:
        for p2 in pos:
            d_ = VDist(p1,p2)
            if (d_ != 0):
                d.append(d_)
    return np.mean(d), d
```

Listing 3: Calculo de aristas de un solido platónico

4.1. Tetraedro

El siguiente bloque de código se basa en averiguar que combinación de posiciones es la que forma las caras de tetraedro. En la figura (2) se muestra los resultados.

```
centros = []
lst = list(combinations([0,1,2,3], 3))
for idx in lst:
    c = (0,0,0)
    for i in range(0,np.shape(lst)[1]):
        c = tuple(map(sum, zip(c, P[idx[i]]) ))
    c = tuple(t/np.shape(P)[1] for t in c)
    centros.append(c)
```

Listing 4: Calculo de centro por combinación de posiciones

Este primer código es simple debido a que el tetraedro tiene pocas posiciones y no es necesario medir los ángulos internos de las caras para determinar con certeza que es una cara del tetraedro, aunque se puede calcular. Nótese que el centro de las caras de tetraedro es el mismo tetraedro.

Figura 2: Posiciones centrales en las caras triangulares del tetraedro

4.2. Octaedro

Para este bloque ademas de una combinación de posiciones se miden los ángulos entre estas posiciones y se determina si satisface a las caras triangulares del octaedro.

```
comb = list(combinations(octa, 3))
centros = []
for face in comb:
    dprom, d = DistProm(face)
    if (abs(dprom - Arista(octa)) < 1E-2):
        c = np.round(MidPoint(face), 4)
        centros.append(c)
```

Listing 5: Calculo de centro caras cuadradas del cubo

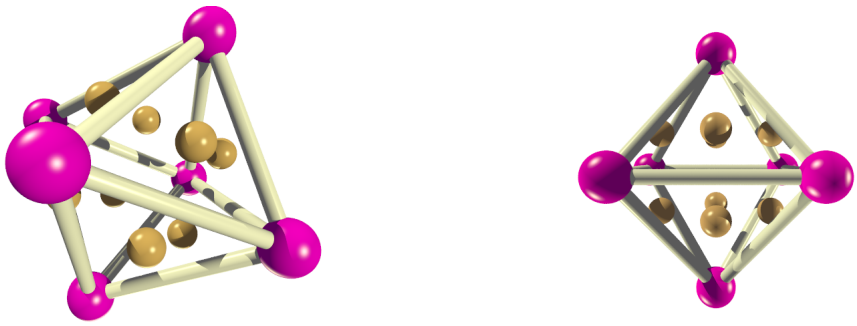

Figura 3: Posiciones centrales en las caras triangulares del octaedro

Se pude observar que el centro de las caras de un octaedro forman un cubo y el centro de las caras de un cubo forma un octaedro y así sucesivamente.

4.3. Cubo

El cubo es un caso particular puesto que requiere un algoritmo distinto para determinar los centros. Es la función DistC que mide las distancias en una trayectoria cerrada de un combinación de posiciones de los vértices del cubo, así mismo se rechaza la combinación de posiciones si no es una trayectoria cerrada.

```
comb = np.array(list(permutations(cubo, 4)))
centros, ang = [], []
for cara in comb:
    dist = DistC(np.array(cara))
    err = abs(np.mean(dist) - Arista(cubo))
```

```
6       if (err < 1E-4 and tuple(MidPoint(cara)) not in centros):
7           centros.append(tuple(MidPoint(cara)))
```

Listing 6: Centro de las caras del cubo

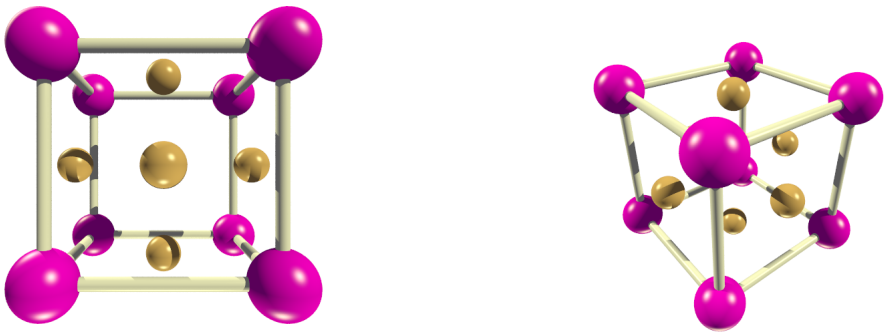

Figura 4: Posiciones centrales en las caras cuadradas del cubo

4.4. Icosaedro

Este código mantiene sencillez debido a que las caras son triangulares.

```
ico_c = []
comb = list(combinations(ico,3))

for face in comb:
    dprom, d = DistProm(face)
    if ( abs(dprom - Arista(ico)) < 1E-2 ):
        c = MidPoint(face)
        ico_c.append(c)
```

Listing 7: Calculo del centro de las cara del Icosaedro

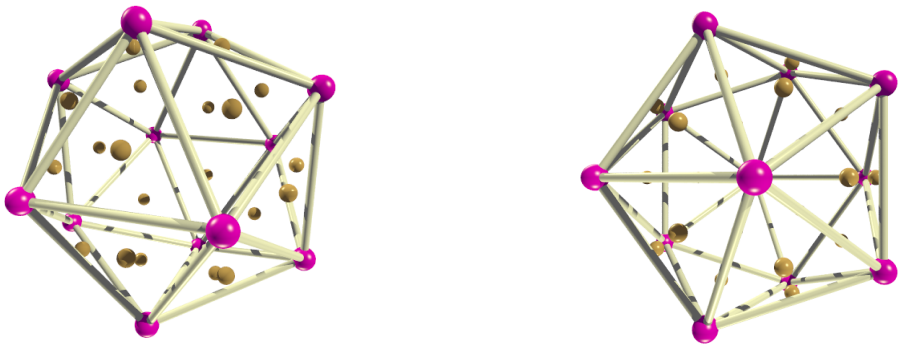

Figura 5: Posiciones centrales en las caras cuadradas del cubo

4.5. Dodecaedro

El dodecaedro presenta caras pentagonales, lo cuál requiere precisar en las distancias entre todos los vértices de la cara, resultando en dos: la distancia mas corta a medida es la medida de la arista y como una segunda más grande es la distancia a_1 a los vértices más lejanos. Dejando inútil el calculo de los ángulos puesto que al querer conseguir una trayectoria cerrada, la medida del ángulo no precisa en ello.

La proporción que hay entre a_1 y a es de (ley de cosenos)

$$\frac{a_1}{a} = \sqrt{2\,(1 - \cos 108°)} = 1{,}618 \tag{5}$$

```
ad = np.round(Arista(dode),decimals=3)
au = np.sqrt(2*(1 - np.cos(np.deg2rad(108))))
cond_d = [ad, ad*au]
np.mean(cond_d)
```

Listing 8: Calculo de variedad de distancias inscritas entre los vértices de un pentágono

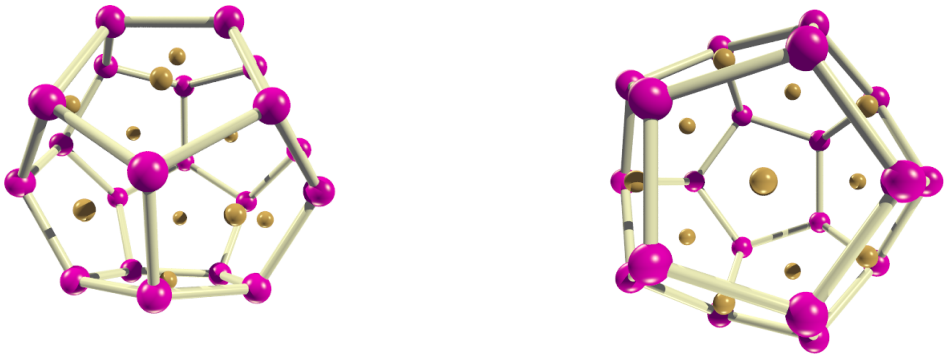

Figura 6: Posiciones centrales en las caras pentagonales del cubo

```
dode_c = []

comb = list(combinations(dode,5))
for face in comb:

    dprom, d = DistProm(face)

    if (abs(dprom - np.mean(cond_d)) < 1E-2):
        c = np.round(MidPoint(face), decimals = 3)
        dode_c.append(tuple(c))
```

Listing 9: Calculo de los centros en los pentágonos del dodecaedro

5. Fullereno C_{60}

Para calcular los centros en una estructura un nivel más complejo como lo es el C_{60}, esté es altamente simétrico y presenta dos tipos de caras poligonales: el pentágono y el hexágono, los cuales comparten aristas (distancia de enlace) según la vecindad, estas pueden ser hexágono-hexágono (6-6) o hexágono-pentágono (6-5) los cuales miden 1,407 $\mathring{A}$ y 1,458 $\mathring{A}$ respectivamente. Es decir, mientras las caras pentagonales tiene solo aristas del tipo (6-5), las caras hexagonales tienen alternancias de aristas (6-6) y (6-5).

5.1. Caras Pentagonales

Las características de cada uno de estos es que el pentágono tiene ángulos internos de 108° y aristas que miden aprox 1.42, mientras que para el hexágono se cuenta con ángulos internos de 120° y dos medidas de aristas que se alternar en una trayectoria cerrada. El algoritmo consiste en cerrar una trayectoria para asegurar que estamos en un pentágono, con la ventaja que tienen el fullereno que cada pentágono esta rodeado de hexágonos, pero no así en viceversa.

```
a = Arista(c60)
poli, centros = [], []
i = 0
while (i < 60):
    j = 0
    poli.append(c60[i])
    while (j < 60):
        pj = c60[j]
        d_ = VDist(poli[-1], pj)
        err = abs(d_ - a[1])
        if (err < 1E-2 and pj not in poli and len(poli) < 5):
            poli.append(pj)
            j = 0
            if (len(poli) == 5):
                c = MidPoint(poli)
                if (tuple(c) not in centros):
                    centros.append(tuple(c))
        else:
            j += 1

    i += 1
    poli = []
```

Listing 10: Calculo de los centros en los pentágonos del Fullereno C_{60}

Obtenemos todas las posiciones centrales de las caras pentagonales mediante el método del punto medio. En el siguiente bloque, se muestra una lista de las posiciones obtenidas.

```
In [1]: centros
Out[1]: [(2.992, 0.0, 1.496),
         (0.925, 2.845, 1.496),
         (0.925, -2.845, 1.496),
         (2.42, 1.758, -1.495),
         (2.42, -1.758, -1.495),
         (0.0, 0.0, 3.345),
         (-2.42, 1.758, 1.495),
         (-0.925, 2.845, -1.496),
         (-2.42, -1.758, 1.495),
         (-2.992, 0.0, -1.496),
         (-0.925, -2.845, -1.496),
         (-0.0, 0.0, -3.345)]
```

Listing 11: Posicione centrales de las caras pentagonales del Fullereno

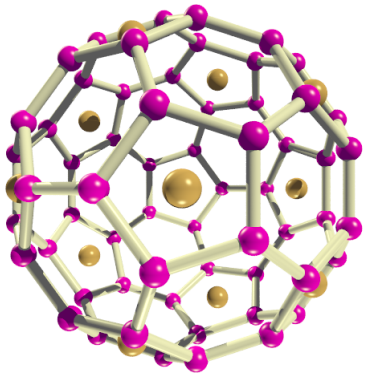

Figura 7: Centros en las caras poligonales del C_{60}

5.2. Caras Hexagonales

Las polígonos hexagonales cuentan con ángulos internos de 120° y tienen dos aristas, una de 1,407 $\mathring{A}$ y 1,458 $\mathring{A}$ alternadas sucesivamente, contando con 3 de cada una. Ilustraremos la funcionalidad de Arista, que se emplea en el siguiente algoritmo. Al aplicar esta función a las posiciones del C_{60} obtenemos lo que sigue

```
In [1]: Arista(c60)
Out[1]: [1.4070825788941252,
         1.457663839034025,
         2.358555277224563,
         2.4810823791372534,
         2.8647359730112787,
         3.5933873352575065,
         3.7107536798011713,
         4.133797237736114,
         4.530589395598866,
         4.635277788352717,
         4.859033177157977,
         5.223259833938456,
         5.422902440659379,
         5.510646570679926,
         5.814222733193961,
         6.092941475799751,
         6.1628754093099944,
         6.53346170141262,
         6.688624375779495,
         6.732853823027872,
         6.98350712094777,
         6.993793476275667]
```

Es decir, nos desvuelve un listado de las distancias más cortas hasta las más largas respecto a cualquier vértice. Así, las dos distancias más pequeñas respectan a las aristas (6-6) y (6-5). El siguiente algoritmo acopla dos ciclo while e incluye dentro condicional if que exige cumplir no solo una alternancia en las aristas, si no la siguiente condición de distancias, con el objetivo de que cierre una trayectoria que aseguré ser una cara hexagonal.

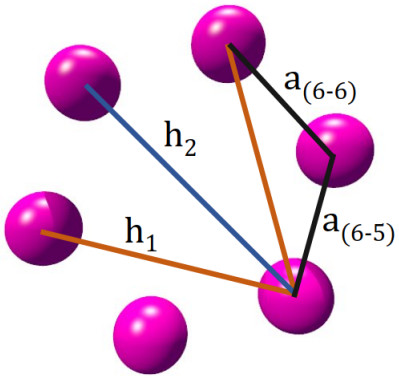

Figura 8: Distancias inscritas en una cara hexagonal del C_{60}

Estas distancias se pueden calcular mediante la ley de coseno, las cuáles quedan definidas de la siguiente manera

$$h_1^2 = a_{(6-6)}^2 + a_{(6-5)}^2 - 2a_{(6-6)}a_{(6-5)}\cos(120°) \tag{6}$$
$$h_2^2 = a_{(6-5)}^2 + h_1^2 - 2a_{(6-5)}h_1\cos(89{,}414°) \tag{7}$$

Siendo $a_{(6-6)} = 1{,}407\,\mathring{A}$ y $a_{(6-5)} = 1{,}458$ los enlaces (6-6) y (6-5) respectivamente, lo que resulta en

$$h_1 = 2{,}481\,\mathring{A} \qquad\qquad h_2 = 2{,}851\,\mathring{A} \tag{8}$$

Esto se escribe como

```
a = Arista(c60)
ang = np.deg2rad(120)
h1 = sqrt(a[0]**2 + a[1]**2 - 2*a[0]*a[1]*cos(ang))
ang1 = np.deg2rad(89.414)
h2 = sqrt(a[1]**2 + h1**2 - 2*a[0]*h1*cos(ang1))
cond = [h1,h2]
```

Listing 12: Calculo de distancias internas h_1 y h_2 dentro de una cara hexagonal en C_{60}

El asumir un angulo de 89,414° para el calculo de h_2 provoca una generalización no adecuada para todas las caras hexagonales, lo cuál requiere perfeccionar este detalle. Así, el algoritmo es el que sigue

```
hexa = []
i = -1
while (i<60):
    i = i + 1
    poli = [c60[i]]
    j = 0
    alt = 1
    while (j<60 and len(poli) < 6):
        pj = c60[j]
        d_ = VDist(poli[-1], pj)

        err = abs(d_ - a[alt])
        if (err < 1E-2 and pj not in poli and len(poli) < 6):
            poli.append(pj)
            j = 0
            if (alt == 1):
                alt = 0
            else:
                alt = 1
        else:
            j += 1
        if (len(poli) == 6):
            d = []
            for idx in [2,3,4]:
                d1 = VDist(poli[idx], poli[0])
                d.append(d1)
            d = list(set(np.round(d, 3)))

            err1 = abs(np.mean(cond) - np.mean(d))
            if (err1 < 1E-2):
                c = tuple(MidPoint(poli))
                print(np.array(poli))

                if (c not in hexa):
                    hexa.append(c)
```

Listing 13: Calculo del posiciones centrales en caras hexagonales C_{60}

Como resultado obtenemos 7 de 20 posiciones centrales de las caras hexagonales, lo que muestra un truncamiento no adecuado de la distancia h_2. Aun así, obtenemos la siguiente imagen.

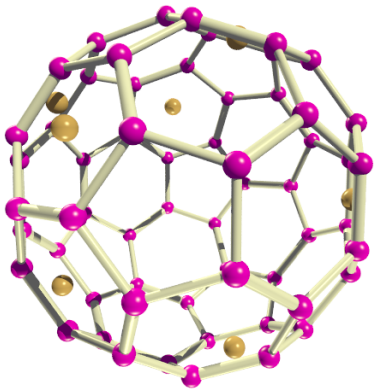

Figura 9: Posiciones centrales en caras hexagonales del C_{60}

6. Conclusión

Obtener los centros de las caras de los sólidos platónicos así como de cualquier estructura de átomos cerrada en $3D$ es de mucha utilidad debido a que estamos localizando distintos sitios de interacción, por ejemplo el C_{60} tiene huecos de caras hexagonales y pentagonales, esta molécula es de interés de estudio en astrobiología (por ejemplo), debido a que se ha encontrado en el espacio interestelar, sin embargo se sabe que interacciona en entornos de abundancia de distintas especies de átomos, como hidrógeno, helio, oxigeno, litio, entre algunos metales pesados en formación como el hierro, se sabe que esta moléculas son precursoras de la vida, entonces estudiar la interacción entre C_{60} y las demás especies en el medio y todas las configuraciones posibles, se sospecha entre investigadores que debido a detecciones de aminoácidos en el medio interestelar, aunque no se ha podido confirmar de manera inequívoca, se han encontrado también aminoácidos en meteoritos, entonces es de interés para la comunidad de astrobiología conocer si también existen procesos de formación de aminoácidos en el espacio. Pero a fines prácticos, también se puede estudiar, con fines de aplicación, un mayor numero de interacciones en un diseño de laboratorio. Es por ello que esta primera aproximación ofrece un área oportuna de cálculos y caracterizar las propiedades electrónicas y químicas. Hoy en día ya se estudia ciertos tipos de interacción pero no se tiene una base metodológica y eficiente para localizar sin mucho esfuerzo estos sitios de interacción.

7. Apéndice

7.1. Funciones Vectoriales

Funciones vectoriales definidas en Python, yendo dese el calcula de distancias, encontrar el punto medio de un conjunto de puntos y el algoritmo para determinar las aristas de una estructura de átomos.

```
def VDist(V1, V2):
    return np.sqrt(np.sum( (np.array(V2)-np.array(V1))**2 ))
```

Listing 14: Calculo de distancias entre dos posiciones

```
def VAngleD(V1,V2,V0):
    v1 = np.subtract(V1,V0)
    v2 = np.subtract(V2,V0)
    Lx = np.dot(v1,v1)
    Ly = np.dot(v2,v2)
```

```
    L = np.dot(v1,v2)/np.sqrt(Lx*Ly)
    return np.rad2deg(np.arccos(L))
```

Listing 15: Calculo de Angulo entre dos vectores no posicionados en el origen

```
def DistC(pos):
    lista = list(permutations(pos, 2))
    d = np.array([])
    for i in range(0,len(pos)):
        d_ = VDist(pos[i-1],pos[i])
        d = np.append(d, d_)

    return np.array(d)
```

Listing 16: Calculo de distancias en una trayectoria cerrada entre vértices del cubo

```
def DistProm(pos):
    d = []
    for p1 in pos:
        for p2 in pos:
            d_ = VDist(p1,p2)
            if (d_ != 0):
                d.append(d_)
    return np.mean(d), d
```

Listing 17: Calculo de la distancia promedio de una lista de posiciones

```
def MidPoint(pos):
    c = (0,0,0)
    for point in pos:
        c = np.add(c,point)
    return np.array(c/len(pos))
```

Listing 18: Calculo del punto medio dado una lista de posiciones inscritas en un triangulo

```
def rotV(th, v):
    rot = np.array([[cos(th), -sin(th), 0],[sin(th), cos(th), 0],[0,0,1]])
    return np.matmul(rot, v)
```

Listing 19: Calculo de la rotación de vectores en un angulo dado

```
def Arista(pos):
    dprom, d = DistProm(c60)
    M, lst = [], []
    m0 = np.min(d)
    m = m0
    while (abs(m0 - m) < 1E-2 and len(d)>0):
        idx = np.where(d == m)
        M.append(m)
        d = np.delete(d, idx)
        if (len(d) > 0):
            m = np.min(d)
            if (abs(m0 - m) > 1E-2):
                lst.append(np.mean(M))
                m0 = np.min(d)
                M = []
    return lst
```

Listing 20: Función arista para C_{60}