Taller 4: Geometría Computacional: Cascos Convexos

Miguel Ángel Gutiérrez Ibagué - Juan Sebastián Bastos Cruz

October 18, 2018

Abstract

El siguiente documento contiene el desarrollo de los algoritmos para calcular cascos convexos de manera diferente. En primer lugar encontramos el algoritmo incremental, en segundo lugar el algoritmo basado en la técnica de dividir y vencer, y, finalmente el algoritmo conocido como Jarvis - March, o de Wrapping.

Manual del usuario

Para la gráficación se hizo uso del módulo matlibplot.pyplot perteneciente a Python. Su instalación se realiza de la siguiente manera, en los tres casos se realiza la instalación mediante la consola y en modo administrador:

1. Para sistema operativo Windows: pip install matplotlib
2. Para sistema operativo Mac OS: En especial necesitamos instalar primero el homebrew con el siguiente comando brew install pkg-config, posteriormente se instala con el comando pip install matplotlib
3. Para sistema operativo Debian / Ubuntu: sudo apt-get install python3-matplotlib
4. Para sistema operativo Fedora: sudo dnf install python3-matplotlib
5. Para sistema operativo Red Hat: sudo yum install python3-matplotlib
6. Para sistema operativo Arch: sudo pacman -S python-matplotlib

A continuación se mostrarán los pasos para compilar los códigos de cascos convexos: Inicialmente debemos hacer claridad en que los párametros usados por los 3 algoritmos son los siguientes:
- n: Cantidad de puntos a generar
- t: Tipo de distribución, 'e' para una distribución elíptica, y 'r' para una distribución rectangular.
- a: En un rectángulo representa el tamaño, y en una elipse representa el diámetro.
- b: Representa la altura,
- r: Representa el ángulo de inclinación.

1. Para el algoritmo de metodología incremental se ejecutará de la siguiente manera: python INCREMENTAL_HULL.PY N T A B R
2. Para el algoritmo de metodología dividir y vencer se ejecutará de la siguiente manera: python DC_HULL.PY N T A B R
3. Para el algoritmo de Jarvis - March se ejecutará de la siguiente manera: python JARVIS_HULL.PY N T A B R

1 Formalización del problema

Dado un conjunto β de n puntos diferentes repartidos en una distribución elíptica o rectangular. Se espera retornar el casco convexo más pequeño que pueda contener todos los puntos contenidos en el conjunto β.

2 Formalización del algoritmo:

2.1 Entradas:

- Un conjunto β de n puntos distintos repartidos en una distribución elíptica o rectangular definida de la siguiente manera: $\beta = \langle (s_i, s_j) : i \in \{1..n\} \wedge j \in \{1..n\} \subset \mathbb{R} \rangle$

- Un cáracter t que indica la distribución elíptica o rectangular que se utilizará.

- Un entero n que contiene la cantidad de puntos que se encontrarán dentro la distribución.

- Un entero a que indica el tamaño del rectángulo, o, el diámetro de la elipse.

- Un entero b que indica la altura de la distribución.

- Un entero r que indica el grado de inclinación de la figura.

2.2 Salidas:

El conjunto γ que representa el Casco Convexo más pequeño de m puntos diferentes que contenga el conjunto β de puntos enunciado en las entradas. $\gamma = \langle (s_i, s_j) : i \in \{1..n\} \wedge j \in \{1..n\} \subset \mathbb{R} \rangle | \beta \subseteq \gamma$

3 Observaciones

- Cuando $n < 3$ implica que no existe ningún casco convexo, esto debido a que no hay puntos suficientes que permitan generar un polígono que cumpla con las condiciones de un caso convexo.
- Cuando $n \geq 3$ existe un casco convexo y puede hallarse utilizando cualquiera de los algoritmos mencionados previamente.

4 Escritura del algoritmo

4.1 Algoritmo Incremental

4.1.1 Pseudocódigo

4.1.2 Invariante

La invariante es que se toman puntos mínimos por su coordenada en izquierda o derecha que se encuentra dentro del casco.

4.1.3 Análisis de Complejidad

La complejidad es $O(n)$ puesto que se realiza el recorrido del conjunto de puntos en los dos sentidos, sin embargo estos ciclos se realizan de manera independiente, siendo en el peor de los casos de $O(n)$ cada uno.

4.2 Algoritmo Divide y Conquista

4.2.1 Pseudocódigo

4.2.2 Invariante

Se garantiza que todos los puntos analizados se encuentran dentro del casco convexo. Es decir, no se analizará ningún punto que no se encuentre dentro del casco convexo generado.

4.2.3 Análisis de Complejidad

La complejidad del algoritmo de dividir y vencer es de $O(n \log n)$ debido a que se requiere un tiempo $O(n)$ para realizar el merge de los cascos hallados por izquierda y derecha. Además, se debe tener en cuenta lo que se requiere para realizar la división del conjunto de puntos inicial.

4.3 Algoritmo Jarvis - March

4.3.1 Pseudocódigo

4.3.2 Invariante

Para el ciclo que permite encontrar el casco convexo más pequeño, se garantiza que se temrinará en el momento en el que se vuelva al punto de inicio. Además, al recorrer todo el conjunto de puntos, se garantiza que ninguno quedará fuera del análisis para el casco.

Algorithm 1 Incremental Hull

```
procedure pointsRightTurn(p1, p2, p3)
    x1 ← p1.x − p2.x
    x2 ← p1.x − p3.x
    y1 ← p1.y − p2.y
    y2 ← p1.y − p3.y
    total ← y2 ∗ x1 − y1 ∗ x2
    return total
end procedure
procedure incrementalHull(Points)
    Points ← sortByXAxis(Points)
    Lupper = []
    Lupper.append(Points[1])
    Lupper.append(Points[2])
    for i ← 3 (to) |Points| do
        Lupper.append(Points[i])
        while |Lupper|>2 and pointsRightTurn(Lupper[|Lupper|-2],Lupper[|Lupper|-1],Lupper[|Lupper|]) do
            Lupper.pop(|Lupper|-1)
        end while
    end for
    Llower = []
    Llower.append(|Points|)
    Llower.append(|Points|-1)
    for i ← |Points| downto 1 do
        Llower.append(Points[i])
        while |Llower|>2 and pointsRightTurn(Llower[|Llower|-2],Llower[|Llower|-1],Llower[|Llower|]) do
            Llower.pop(|Llower|-1)
        end while
    end for
    Llower.pop(0)
    Llower.pop(|Llower|)
    hull ← Llower + Lupper
    return hull
end procedure
```

4.3.3 Análisis de Complejidad

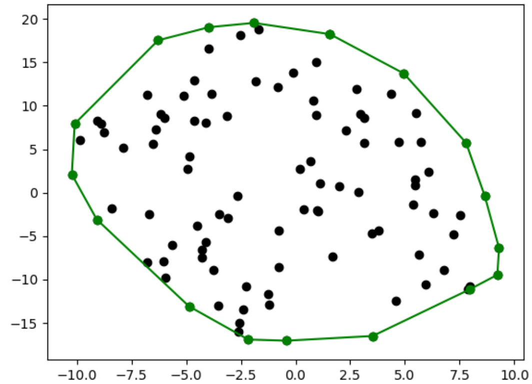
La complejidad de este algoritmo es $O(m*n)$ siendo m la cantidad de puntos del casco convexo, y, n la cantidad de puntos incial de la distribución. Por ende se necesita $O(m)$ para el recorrido del casco convexo, y, $O(n)$ para recorrer el conjunto de puntos. Por ende, en el peor de los casos se tendría una complejidad de $O(n^2)$

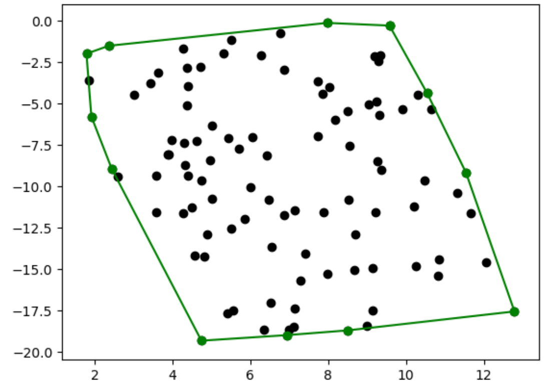
5 Pruebas

Se realizaron tres pruebas con cada algoritmo, a continuación se presentarán las gráficas generadas por cada uno de ellos.

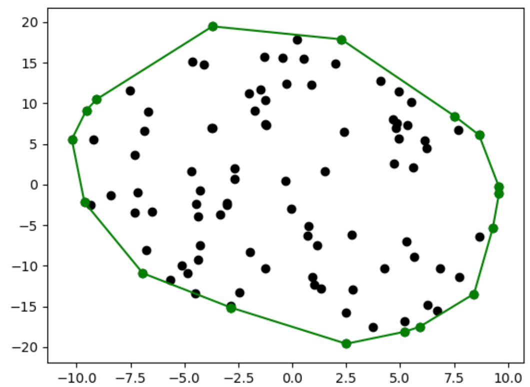
5.1 Prueba 1. n = 100, a = 20, b = 10, r = 30

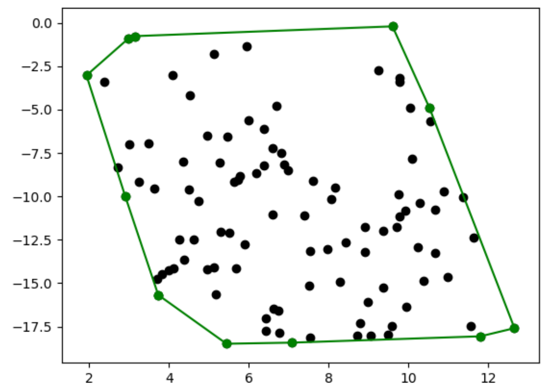
5.1.1 Algoritmo Incremental



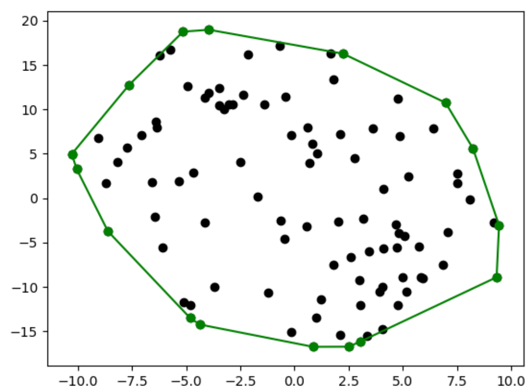


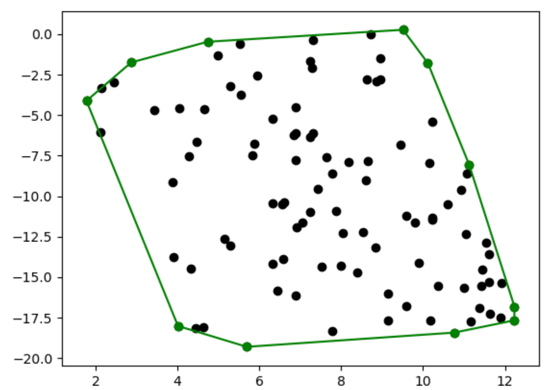
5.1.2 Algoritmo Divide y Conquista





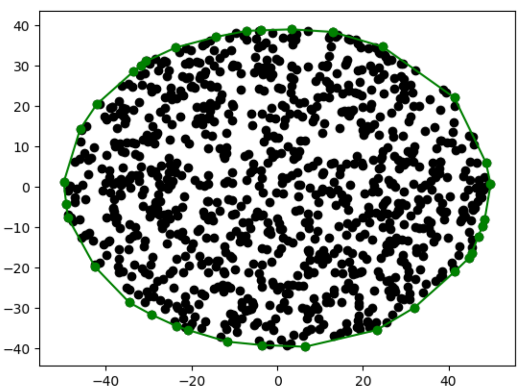
5.1.3 Algoritmo Jarvis - March

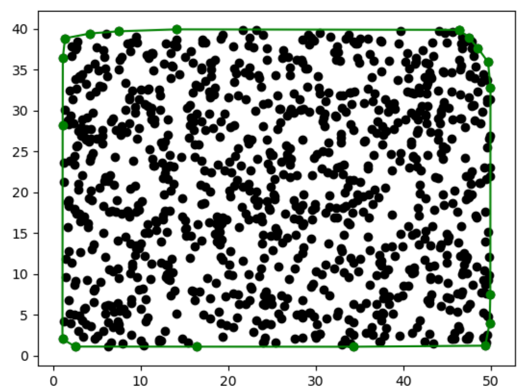




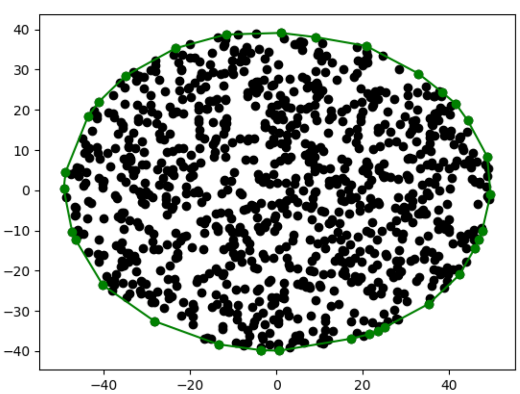
5.2 Prueba 1. n = 1000, a = 50, b = 40, r = 0

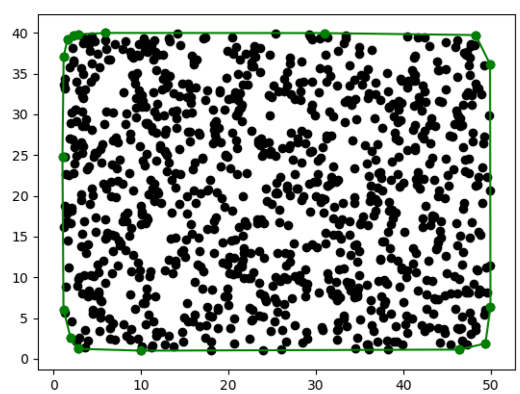
5.2.1 Algoritmo Incremental



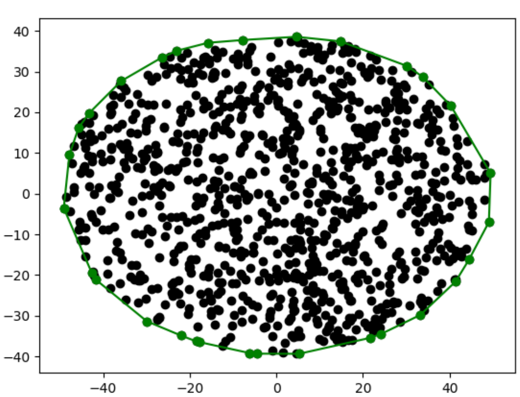


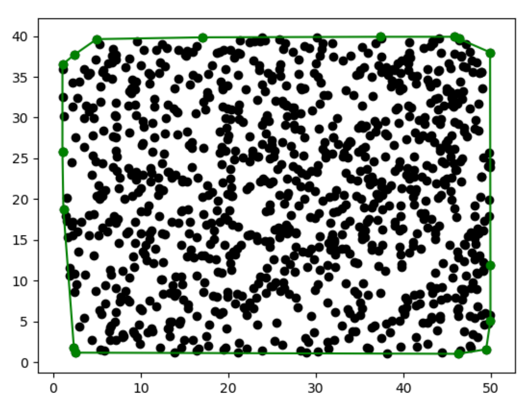
5.2.2 Algoritmo Divide y Conquista





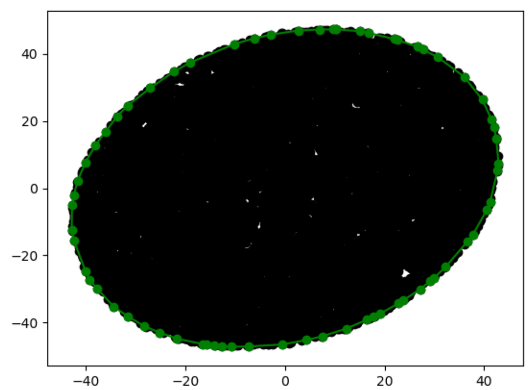
5.2.3 Algoritmo Jarvis - March

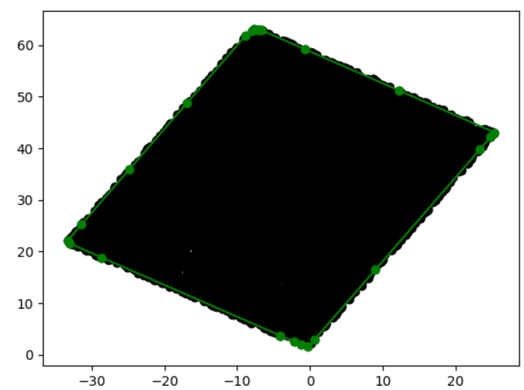




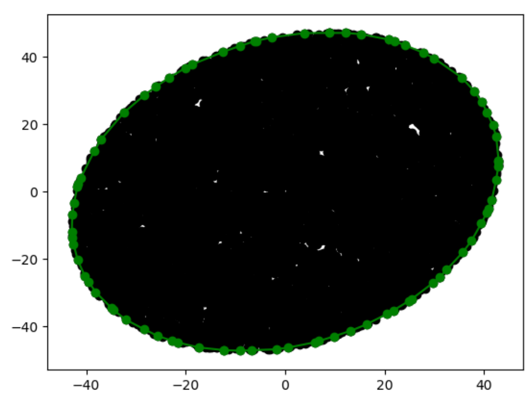
5.3 Prueba 1. n = 10000, a = 50, b = 40, r = 45

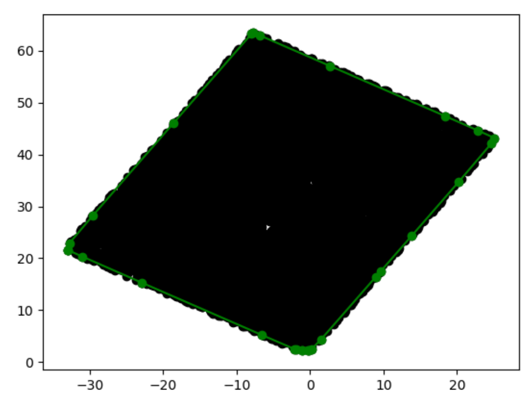
5.3.1 Algoritmo Incremental



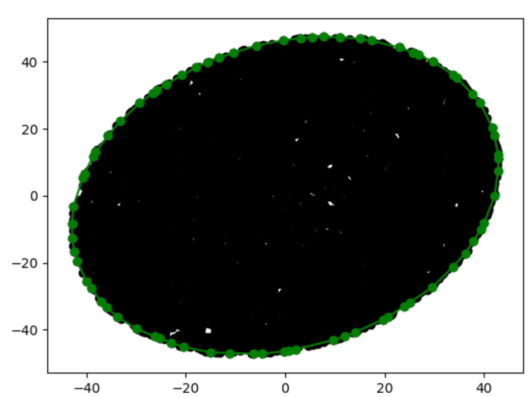


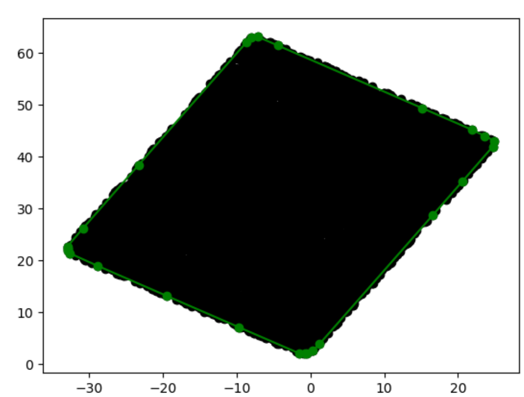
5.3.2 Algoritmo Divide y Conquista



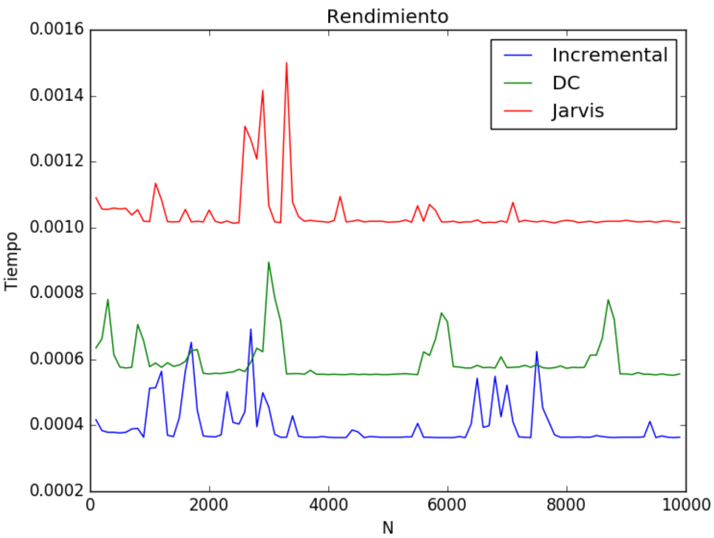


5.3.3 Algoritmo Jarvis - March





6 Análisis de resultados



- Haciendo un análisis de la gŕafica obtenida, podemos evidenciar que el algoritmo más eficiente es el algoritmo incremental, puesto que, pese a que la gráfica presentada demuestra picos en los rendimientos de cada uno de los algoritmos desarrollados.
- A partir de la gráfica, podemos comprobar que, si se realiza un análisis de las gráficas obtenidas, el algoritmo de menor complejidad es el algoritmo de metodología incremental. Verificando que, si observamos el peor de los casos, el algoritmo más eficiente, según su complejidad, es el algoritmo incremental, con una complejidad de $O(n)$.

Algorithm 2 DC Hull

```
procedure DCHull(Points)
    (minimum, maximum) ← extremePoints(Points)
    hull ← subHull(minimum, maximum, points)
    hull ← hull + subHull(maximum, minimum, points)
    return hull
end procedure
procedure subHull(a, b, S)
    S ← leftPoints(a, b, S)
    c ← farthestPoint(a, b, S)
    if c == None then
        return [b]
    end if
    hull ← subHull(a, c, S)
    hull ← hull + subHull(c, b, S)
    return hull
end procedure
procedure extremePoints(points)
    minimum ← ∞
    maximum ← −∞
    posMax ← 0
    posMin ← 0
    for i ← 1 (to) |Points| do
        if points[i].y > maximum then
            maximum ← points[i].y
            posMax ← i
        end if
        if points[i].y < minimum then
            minimum ← points[i].y
            posMin ← i
        end if
    end for
    return (points[posMin], points[posMax])
end procedure
procedure leftPoints(s, e, points)
    left ← []
    for i ← 1 (to) |Points| do
        if pointOp(s, e, points[i]) == 2 then
            left.append(points[i])
        end if
    end for
    return left
end procedure
procedure pointOp(p1, p2, p3)
    val ← (p2.y − p1.y) ∗ (p3.x − p2.x) − (p2.x − p1.x) ∗ (p3.y − p2.y)
    if val == 0 then
        return 0
    else if val > 0 then
        return 1
    else
        return 2
    end if
end procedure
```

Algorithm 3 Algoritmos de ayuda

```
procedure FARTHESTPOINT(s, e, points)
    maximum ← −∞
    for i ← 1 (to) |Points| do
        if points[i] ! = s and points[i] ! = e then
            dist ← distance(s, e, points[i])
            if dist > maximum then
                maximum ← dist
                maxP ← points[i]
            end if
        end if
    end for
    if maximum ! = −∞ then
        return maxP
    else
        return None
    end if
end procedure
procedure DISTANCE(start, end, point)
    nom ← abs((end.y − start.y) ∗ point.x − (end.x − start.x) ∗ point.y + endx ∗ start.y − end.y ∗ start.x)
    denom ← ((end.y − start.y) ∗∗ 2 + (end.x − start.x) ∗∗ 2) ∗∗ 0.5
    result ← nom / denom
    return result
end procedure
```

Algorithm 4 Jarvis Hull

```
procedure jarvisHull(Points)
    if |Points| < 3 then
        return
    end if
    hull ← []
    left ← 0
    for i ← 2 (to) |Points| do
        if points[i].x < points[left].x then
            left ← i
        end if
    end for
    start ← left
    while true do
        hull.append(points[start])
        q ← (start + 1) % |Points|
        for i ← 1 (to) |Points| do
            if pointOp(points[start], points[i], points[q]) == 2 then
                q ← i
            end if
        end for
        start ← q
        if start == left then
            break
        end if
    end while
    return hull
end procedure
procedure pointOp(p1, p2, p3)
    val ← (p2.y − p1.y) ∗ (p3.x − p2.x) − (p2.x − p1.x) ∗ (p3.y − p2.y)
    if val == 0 then
        return 0
    else if val > 0 then
        return 1
    else
        return 2
    end if
end procedure
```