UNIVERSIDAD NACIONAL DE INGENIERÍA

Aplicación de algoritmos de CONVEX-HULL en 2D

(**Application of CONVEX-HULL algorithms in 2D**)

AGUILAR ROMERO, Jhon Hamilton[1]; LARREATEGUI CASTRO, Angel Vidal [2]; REYMUNDO PARIONA, Juan Pablo[3]; VENTURA EUGENIO, James Jeanpierre[4]

Facultad de Ciencias 1*, Universidad Nacional de Ingeniería* 1*, e-mail: jhon.aguilar.r@uni.pe*
Facultad de Ciencias 2*, Universidad Nacional de Ingeniería* 2*, e-mail: alarreateguic@uni.pe*
Facultad de Ciencias 3*, Universidad Nacional de Ingeniería* 3*, e-mail: jreymundop@uni.pe*
Facultad de Ciencias 4*, Universidad Nacional de Ingeniería* 4*,e-mail: jventurae@uni.pe*

Resumen

El presente trabajo describe los algoritmos de cápsula convexa y se concentra en algunos de ellos, con el fin de mostrar aplicaciones con respecto al algoritmo de Graham, Jarvis March y Quickhull en dos dimensiones usando el lenguaje de programación Python.

Palabras Claves: **Algoritmos de la cápsula convexa, Algoritmo de Graham, Algoritmo de Jarvis March, Algoritmo Quickhull, Aplicaciones de la cápsula convexa en 2 dimensiones**

Abstract

This research work describes the convex hull algorithms and focuses in some of them to show applications dealing with Graham algorithm, Jarvis March algorithm and Quickhull algorithm about two dimensions in Python programming language.

Keywords: **Convex Hull algorithms, Graham algorithm, Jarvis March algorithm, Quickhull algorithm, 2-dimensional convex hull applications**

1. INTRODUCCIÓN

La Geometría computacional es una rama de la ciencia de la computación que estudia algoritmos para resolver problemas geométricos. Se trata de encontrar algoritmos eficientes o procedimientos computacionales para resolver un amplio espectro de problemas geométricos.

El uso de objetos geométricos como puntos, rectas y polígonos son la base de una amplia variedad de importantes aplicaciones en optimización . Problemas tan triviales como determinar si un punto dado se encuentra dentro de un polígono o si dos segmentos se intersecan, requieren programas no triviales.

En análisis convexo se plantea la siguiente pregunta **¿cómo determinar la cáscara convexa (convex hull) de un conjunto de puntos?**. Abordaremos tres algoritmos : El algoritmo de Graham, Jarvis March y Quickhull) y compararemos sus eficiencias.

Los **objetivos parciales** del presente estudio son el análisis de todos los algoritmos de la convex hull, su comprensión y posterior aplicación en dos dimensiones usando el lenguaje de programación Python.
El **objetivo general** es conocer algunas aplicaciones de los algoritmos de la cápsula convexa. Se ha elegido el tema de **Convex-hull en 2D pues se justifica** en el uso de conceptos matemáticos y algoritmos computacionales. El interés del trabajo es la aplicación de la convex-Hull en el reconocimiento facial.

2. CONCEPTOS PREVIOS

A continuación mencionaremos algunos conceptos básicos que tienen que ser dominados de antemano, para comprender los temas a tratar.

2.1. Convexidad

Definición 2.1. *Un conjunto $S \subseteq \mathbb{R}^n$ es **convexo** si, para todo $x, y \in S$, $\lambda x + (1-\lambda)y \in S$, para todo $\lambda \in [0,1]$ [15].*

(a) Convexo. (b) No-convexo.

Figura 1: Ejemplo de convexo y no convexo en $\mathbb{R}^2$

2.2. Combinación convexa

Definición 2.2. *Sean $v_1, v_2, \cdots, v_k \in \mathbb{R}^n$. Se dice que v es **combinación convexa** de $\{v_1, v_2, \cdots, v_k\}$ si $v = \sum_{i=1}^{k} \lambda_i v_i$ para algunos λ_i de tal forma que $\lambda_i \geq 0$ y $\sum_{i=1}^{k} \lambda_i = 1$ [15].*

2.3. Cápsula convexa

Definición 2.3. *Sea $K \subset \mathbb{R}^n$ un conjunto cualquiera. Definimos la **cápsula convexa** de K, que representaremos por $\mathbf{co(K)}$, como la intersección de todos los subconjuntos convexos de $\mathbb{R}^n$ que contienen a K. Notación:*

$$co(K) := \bigcap \Big\{ A : A \ \ es\ convexo\ y\ \ K \subset A \Big\}$$

Teorema 2.1. *Caracterización de cápsula convexa. Para todo $S \subset \mathbb{R}^n$ su cápsula convexa admite la representación:*

$$co(S) = \left\{ \sum_{i=1}^{p} \lambda_i x_i : \sum_{i=1}^{p} \lambda_i = 1, \quad \lambda_i \geq 0, x_i \in S, \quad p \in \mathbb{N} \right\}$$

Demostración .1. *Ver [15].*

Definición 2.4. *Sea Q la cápsula convexa de un número finito de vectores $\{v_1, v_2, \cdots, v_k\}$, entonces Q es un politopo.*

Lema 2.1. *Q es convexo.*

Prueba 2.1. *Sean $x, y \in Q$, entonces podemos escribir a:*

$$x = \sum_{i=1}^{k} \alpha_i v_i, \quad con \ \ \alpha_i \geq 0 \ \ y \ \ \sum_{i=1}^{k} \alpha_i = 1.$$

$$y = \sum_{i=1}^{k} \beta_i v_i, \quad con \ \ \beta_i \geq 0 \ \ y \ \ \sum_{i=1}^{k} \beta_i = 1.$$

Para $\lambda \in [0,1]$, entonces

$$\lambda x + (1-\lambda)y = \lambda \sum_{i=1}^{k} \alpha_i v_i + (1-\lambda) \sum_{i=1}^{k} \beta_i v_i$$

$$= \sum_{i=1}^{k} [\lambda\alpha_i + (1-\lambda)\beta_i]\, v_i$$

Es claro que $\lambda\alpha_i + (1-\lambda)\beta_i \geq 0$ *para todo* $i = 1, 2, \cdots, k$; *y*

$$\sum_{i=1}^{k} (\lambda\alpha_i + (1-\lambda)\beta_i) = \lambda \sum_{i=1}^{k} \alpha_i + (1-\lambda) \sum_{i=1}^{k} \beta_i = 1$$

Así

$$\lambda x + (1-\lambda)y = \sum_{i=1}^{k} \delta_i v_i$$

Donde $\delta_i = \lambda\alpha_i + (1-\lambda)\beta_i$ *con* $\delta_i \geq 0$, *para todo* $i = 1, 2, \cdots, k$ *y* $\sum_{i=1}^{k} \delta_i = 1$. *Así* $\lambda x + (1-\lambda)y \in Q$.

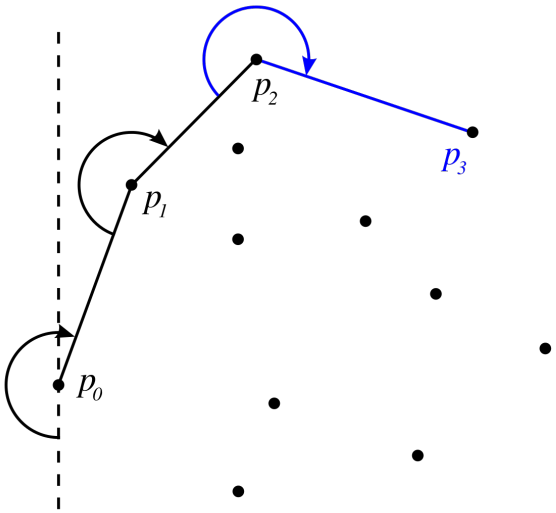

Figura 2: Algoritmo Jarvis March

3. Algoritmos de la Cápsula Convexa

3.1. Algoritmo de Gift Wrapping (Jarvis March)

Se basa en que dado un conjunto de puntos, encontrar el punto que se encuentre más a la izquierda del plano (al cual llamaremos punto P), se traza una línea imaginaria de manera vertical tal que pase por el punto P. Para encontrar el siguiente P_1 punto lo que haremos será comparar todos los puntos del conjunto (Q) tal que el ángulo formado por la recta que pasa por P y el segmento $\bar{PQ}$ sea el mínimo posible, al punto Q que cumpla esa condición lo llamaremos P_1. Para hallar el siguiente punto P_{i+1} se logrará de manera similar, buscaremos el punto Q de conjunto tal que el ángulo formado en sentido horario por los puntos P_{i-1}; P_i; Q sea el mínimo, al punto Q que cumpla dicha condición se le llamará P_{i+1}. Este proceso se realizará hasta que el último punto encontrado sea el punto P.

3.2. Algoritmo de Graham

Lo que hace este algoritmo es en primer lugar calcular el punto mínimo, es decir al que tenga menor valor en el eje Y, en caso exista más de uno se buscará al que tenga menor valor en el eje X (por comodidad lo llamaremos punto P). Luego se ordena los puntos tomado como referencia al punto P, se puede usar cualquier algoritmo de ordenamiento, el más común es el que ordena por el ángulo que $\bar{PQ}$ con la recta horizontal que pasa por P (Q es cualquier punto del conjunto). Nótese que los ángulos varían de 0 a 2π. Después de lo anterior, empezamos a formar la envolvente convexa. Buscamos un el siguiente punto en el orden al que llamaremos P_1, luego vemos si el siguiente punto se encuentra a un giro horario o anti-horario. Si el giro es horario, entonces pasamos al siguiente punto. Si el giro es anti-horario, entonces retrocedemos un punto y omitimos dicho punto pasando al siguiente. El algoritmo termina cuando el último punto coincide con el punto P. Nota: Este algoritmo trabaja con pilas.

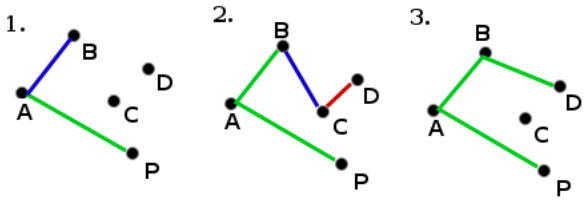

Figura 3: Algoritmo de Graham

3.3. Algoritmo de Quickhull

La versión en $2D$ del algoritmo Quickhull puede dividirse en los siguientes pasos: Buscar un par de puntos con menor y mayor coordenada en el eje x, ya que estos siempre forman parte de la envolvente convexa. Usar la línea entre ambos puntos para dividir el conjunto en dos subconjuntos que serán procesados de forma recursiva. Determinar el punto situado a mayor distancia de la línea anterior. Junto a los dos puntos anteriores, formará un triángulo. Todos los puntos situados en el interior del triángulo pueden ser descartados, ya que no formarán parte del cierre convexo. Repetir los dos pasos anteriores en los dos lados del triángulo (no en el lado inicial). Repetir hasta que no queden puntos sin clasificar. Los puntos seleccionados forman el cierre convexo.

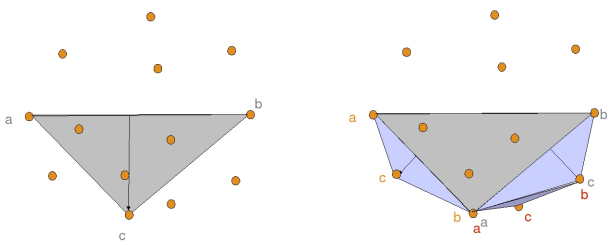

Figura 4: Algoritmo Quickhull

3.4. Algoritmo de la Cadena Monótona

El algoritmo de la cadena monótona de la cápsula convexa fue publicado por A.M Andrew en 1979. Es capaz de determinar la región más exterior de un conjunto de puntos. Un conjunto de puntos obtenidos está formado dentro de una cápsula convexa. Este algoritmo construye una capa convexa de un conjunto de puntos en dos dimensiones con tiempo $\mathcal{O}(n \log n)$. La forma de trabajo es primero se realiza el algoritmo de ordenamiento lexicográfico (desde la coordenada x enlazando con su coordenada y). Entonces construye la estructura superior e inferior de los puntos en el tiempo $\mathcal{O}(n)$. La estructura de la parte superior es parte de la cápsula convexa que puede ser vista desde arriba. Se ejecuta desde los puntos de la derecha hacia los de la izquierda en el orden opuesto a las manecillas del reloj. La estructura inferior es la parte restante de la cápsula convexa [13].

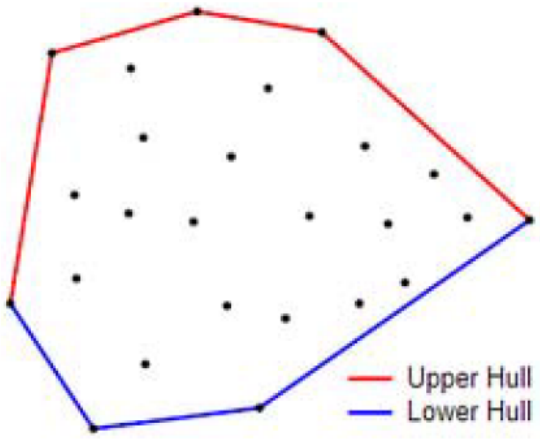

Figura 5: Mecanismo de la cadena monótona de la cápsula convexa donde se define Upper Hull como Cápsula superior y Lower Hull, Cápsula inferior

3.5. Algoritmo Incremental

Propuesto por Michael Kallay en 1984. Agrega puntos uno por uno, es seguramente el algoritmo eficiente más simple para el problema de encontrar un conjunto de puntos de la cápsula convexa, la idea básica del algoritmo incremental de la cápsula convexa es para agregar los puntos uno por uno mientras se mantiene la cápsula convexa [3].

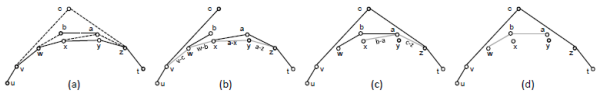

Figura 6: Ejemplo del algoritmo incremental en 2 dimensiones

3.6. Algoritmo Kirkpatrick-Seidel

Es un algoritmo que lleva el nombre de sus inventores David G. Kirkpatrick y Raimund Seidel [8], es de salida sensible que determina la cápsula convexa de un conjunto n de los reales en dos dimensiones de tiempo $\mathcal{O}(n \log h)$ [14]. Es asintóticamente óptimo pero no es muy práctico para problemas de tamaño moderado [10]. Finalmente, el algoritmo está basado en una variación del paradigma divide y vencerás [12].

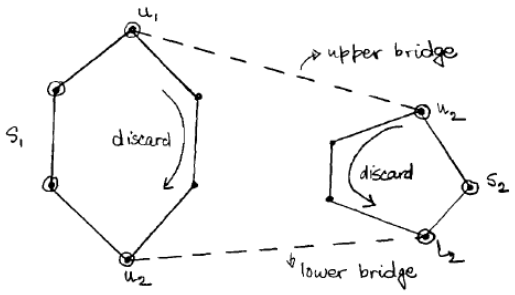

Figura 7: Mezclando dos cápsulas convexas para encontrar el puente superior e inferior

3.7. Algoritmo de Chan

Propuesto por Thimothy M. Chan en 1996. Combina Gift Wrapping con Graham en pequeños subconjuntos de la entrada. Es un algoritmo óptimo de salida sensible óptimo para construir la cápsula convexa en dos y tres dimensiones[1]. Es más simple que el algoritmo de Kirkpatrick-Seidel [4].

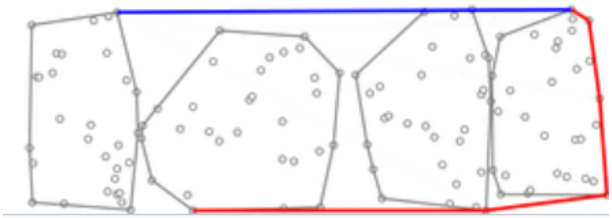

Figura 8: Simulación del algoritmo de Chan en 2 dimensiones

3.8. Comparación de algoritmos de la cápsula convexa

En la siguiente figura se muestra la comparación de cada uno de los algoritmos de la cápsula convexa revisados con la finalidad de mostrar al descubridor, año de descubrimiento y la velocidad de cada uno de ellos (complejidad temporal).

Algoritmo	Autor	Año	Velocidad
Fuerza Bruta	-	-	$O(n^4)$
Envoltorio de regalo	Chan y Kapur	1970	$O(nh)$
Escaneo de Graham	Graham	1972	$O(n \log n)$
Marcha de Jarvis	Jarvis	1973	$O(nh)$
Quick Hull	Eddy, Bykat	1977, 1978	$O(nh)$
Divide y conquistaras	Preparata y Hong	1977	$O(n \log n)$
Cadena monótona	Andrew	1979	$O(n \log n)$
Incremental	Kallay	1984	$O(n \log n)$
Matrimonio antes de la conquista	Kirkpatrick y Seidel	1986	$O(n \log n)$

Figura 9: Comparación de algoritmos de la cápsula convexa (n: cantidad de puntos del conjunto; h: cantidad puntos de la cápsula convexa) [13]

4. ANÁLISIS

4.1. Pre-procesamiento

Se extrae el fondo de la imagen y para ello se emplea un contexto geométrico con coherencia (puede ser centro geométrico de la foto, un rayo hacia la izquierda o distancia entre ojos).
Se tiene en cuenta también otros parámetros como RGB (red, green, blue) o HSV (Hue, saturation o Value). Se contempla otros detalles como: Tonalidad de gris, filtrado, rostro respecto al ROI (Region of Interest) como por ejemplo la segmentación ROI empleando preprocesamiento [5] , etc.
Terminado el proceso obtenemos un conjunto de puntos al cual se le puede aplicar los algoritmos ya mencionados para hallar su respectiva cápsula convexa (Convex Hull).

4.2. Reconocimiento facial

Una de las aplicaciones de la cápsula convexa es el reconocimiento facial. En la actualidad es común ver que varios equipos electrónicos usan como sistema de seguridad, el reconocimiento facial; sin embargo, cómo podremos aplicar los conocimientos de la cápsula convexa en el reconocimiento facial. Veamos la siguiente imagen para darnos cuenta la aplicación que tiene el análisis convexo, en específico la cápsula convexa, en el reconocimiento facial.

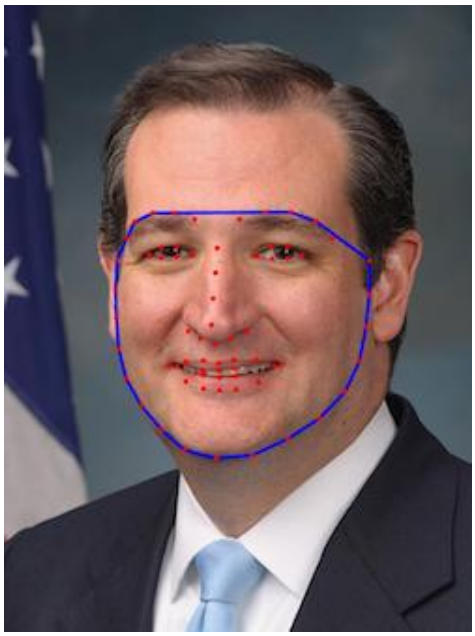

Figura 10: Aplicación de la cápsula convexa en el rostro de una persona

En la figura podemos ver como actúa el análisis convexo en el rostro de una persona. Podemos observar que cada parte del rostro actúa como un punto suponiendo que el rostro de la persona es el plano XY. Entonces, lo que debemos hallar es la cápsula convexa de los puntos en rojo en la figura 10, esta es representada por la línea azul.
Ahora, nuestro objetivo es llevar estos conocimientos a la programación. Para lograr esto se usaría uno de los algoritmos para hallar la envolvente convexa en Python y OpenCV.

5. MODELAMIENTO Y ALGORITMO

Para ver el modelamiento de nuestra aplicación vamos a pasar a ver el código del Algoritmo de Graham Scan en Python [17].

- Entonces, primero vamos a importar estas tres librerías las cuales vamos a usar para la realización de nuestro código. La librería ***matplotlib.pyplot*** la cual nos servirá para poder generar gráficos, también usaremos la librería ***numpy*** para poder crear matrices multidimensionales, también usaremos la librería ***math*** para realizar operaciones matemáticas y finalmente usaremos la librería ***random*** para poder generar números aleatorios más adelante.

```
import matplotlib.pyplot as plt
import math
import numpy as np
import random
```

- Definimos la funcion ***obtener_punto_inferior*** la cual tiene como parámetro ***puntos***. Esta función se va a encargar de devolver el punto cuya abscisa sea la más pequeña.

```
def obtener_punto_inferior(puntos):
    min_indice = 0
    n = len(puntos)
    for i in range(0, n):
        if puntos[i][1] < puntos[min_indice][1] or (
                puntos[i][1] == puntos[min_indice][1] and
      puntos[i][0] < puntos[min_indice][0]):
            min_indice = i
    return min_indice

```

- Ahora vamos a pasar a definir la función ***ordenar_angulo_polar_cos*** a la que vamos a pasar los parámetros ***puntos*** y ***punto_central***. Esta función se va a encargar de ordenar según el ángulo polar.

```
    def ordenar_angulo_polar_cos(puntos, punto_central):
    n = len(puntos)
    cos_valor = []
    rank = []
    norm_lista = []
    for i in range(0, n):
        punto_ = puntos[i]
        punto = [punto_[0] - punto_central[0], punto_[1] -
      punto_central[1]]
        rank.append(i)
        norm_valor = math.sqrt(punto[0] * punto[0] + punto[1] *
       punto[1])
        norm_lista.append(norm_valor)
        if norm_valor == 0:
            cos_valor.append(1)
        else:
            cos_valor.append(punto[0] / norm_valor)

    for i in range(0, n - 1):
        indice = i + 1
        while indice > 0:
            if cos_valor[indice] > cos_valor[indice - 1] or (
                    cos_valor[indice] == cos_valor[indice - 1]
                    and norm_lista[indice] > norm_lista[indice
      - 1]):
                temp = cos_valor[indice]
                temp_rank = rank[indice]
                temp_norm = norm_lista[indice]
                cos_valor[indice] = cos_valor[indice - 1]
                rank[indice] = rank[indice - 1]
                norm_lista[indice] = norm_lista[indice - 1]
                cos_valor[indice - 1] = temp
                rank[indice - 1] = temp_rank
                norm_lista[indice - 1] = temp_norm
                indice = indice - 1
            else:
                break
    puntos_ordenados = []
    for i in rank:
        puntos_ordenados.append(puntos[i])

    return puntos_ordenados

```

- Pasamos a definir también la función ***vector_angulo*** la cual tendrá como parámetro ***vector***. Esta función se va a encargar de devolver el ángulo entre ese ***vector*** que evaluaremos en la función y el vector (1 ; 0). Y este ángulo nos servirá para saber cuántos grados debe girar en sentido antihorario desde (1 ; 0) para poder alcanzar a ese ***vector***.

```
    def vector_angulo(vector):
    norm_ = math.sqrt(vector[0] * vector[0] + vector[1] *
      vector[1])
    if norm_ == 0:
        return 0

    angle = math.acos(vector[0] / norm_)
    if vector[1] >= 0:
        return angle
    else:
        return 2 * math.pi - angle

```

- También vale agregar que vamos a definir la función ***coss_multi*** la cual tendrá como parámetro a dos vectores, los cuales serán ***v1*** y ***v2***. Esta función se va a encargar de hallar el producto cruz de esos dos vectores.

```
def coss_multi(v1, v2):
    return v1[0] * v2[1] - v1[1] * v2[0]

```

- Una vez habiendo definido las funciones anteriores ahora podemos empezar a hacer uso del Algoritmo de Graham Scan. La parte teórica del Algoritmo de Graham y el cómo funciona se encuentra en la parte teórica de los Algoritmos. Adicionalmente, a este código también se le agrega un condicional para que si en caso el usuario solo quiere ingresar la cantidad de dos puntos entonces no se procederá a realizar el Algoritmo, pues la cantidad mínima de puntos que se necesitan es la de 3.

```
def graham_scan(puntos):
    bottom_indice = obtener_punto_inferior(puntos)
    punto_inferior = puntos.pop(bottom_indice)
    puntos_ordenados = ordenar_angulo_polar_cos(puntos,
      punto_inferior)

    m = len(puntos_ordenados)
    if m < 2:
        print("El n mero de puntos es demasiado peque o para
      formar un Convex Hull")
        return

    stack = []
    stack.append(punto_inferior)
    stack.append(puntos_ordenados[0])
    stack.append(puntos_ordenados[1])

    for i in range(2, m):
        longitud = len(stack)
        top = stack[longitud - 1]
        next_top = stack[longitud - 2]
        v1 = [puntos_ordenados[i][0] - next_top[0],
      puntos_ordenados[i][1] - next_top[1]]
        v2 = [top[0] - next_top[0], top[1] - next_top[1]]

        while coss_multi(v1, v2) >= 0:
            if longitud < 3:     # Despu s de agregar estas
      dos l neas de c digo , no se informar  ning n error
      cuando la cantidad de datos sea grande
                break          # Despu s de agregar estas dos
      l neas de c digo , no se informar  ning n error cuando
       la cantidad de datos sea grande
            stack.pop()
            longitud = len(stack)
            top = stack[longitud - 1]
            next_top = stack[longitud - 2]
            v1 = [puntos_ordenados[i][0] - next_top[0],
      puntos_ordenados[i][1] - next_top[1]]
            v2 = [top[0] - next_top[0], top[1] - next_top[1]]
        stack.append(puntos_ordenados[i])

    return stack

```

- En esta parte del código se pasará a definir la función ***Algoritmo_Graham_Scan***. Esta función tendrá como finalidad realizar la ejecución del Algoritmo ya mencionado; primero pidiendo al usuario que ingrese la cantidad de puntos que se van a generar de manera aleatoria para que así pueda proseguir con la ejecución del algoritmo para que finalmente llegue a mostrar la gráfica de la Convex Hull de los puntos que el usuario ingresó.

```
def Algoritmo_Graham_Scan():
    num_puntos = int(input(" Cuantos  puntos desea ingresar?: "
      ))
    print("\n")
    for n in range(num_puntos,num_puntos+1):
        puntos = []
        for i in range(n):
            punto_x = random.randint(0,num_puntos)
            punto_y = random.randint(0,num_puntos)
            print("\n")
            temp = np.hstack((punto_x, punto_y))
            punto = temp.tolist()
            puntos.append(punto)

        resultado = graham_scan(puntos)

        for punto in puntos:
            plt.scatter(punto[0], punto[1], marker='o', c='y',
      s=8)
        longitud = len(resultado)
        for i in range(0, longitud - 1):
            plt.plot([resultado[i][0], resultado[i + 1][0]], [
      resultado[i][1], resultado[i + 1][1]], c='r')
        plt.plot([resultado[0][0], resultado[longitud - 1][0]],
       [resultado[0][1], resultado[longitud - 1][1]], c='r')
        plt.xlabel("EJE X",size = 18)
        plt.ylabel("EJE Y", size = 18)
        plt.title("CONVEX HULL",fontdict={'family':'monospace',
      'color':'darkblue','weight':'bold','size':22})
        plt.show()

```

- Finalmente se implementa un menú en donde habrán dos opciones. Al seleccionar la opción 1 se procederá a ejecutar el Algoritmo de Graham Scan en donde el programa le pedirá al usuario que ingrese la cantidad de puntos que él desea que se genere de manera aleatoria, mientras que si se escoge la opción 2 entonces el programa pasará a finalizarse. También, se implementó una condicional adicional en donde si el usuario elige una opción distinta a las que se muestran en el menú le salte un mensaje el cual dice que ***No has seleccionado una opción correcta.*** y el programa volverá a pedir

al usuario que ingrese una opción válida.

```python
def menu():
    print("\t CONVEX HULL EN PYTHON!\n\t-----------------")
    print("1) Ejecutar el Algoritmo de Graham Scan")
    print("2) Salir")
    opcion = int(input("Elija una opcion: "))
    return opcion

while True:
    opc = menu()
    if opc==1:
        Algoritmo_Graham_Scan()
    elif opc==2:
        break
    else:
        print("No has seleccionado una opcion correcta.")

```

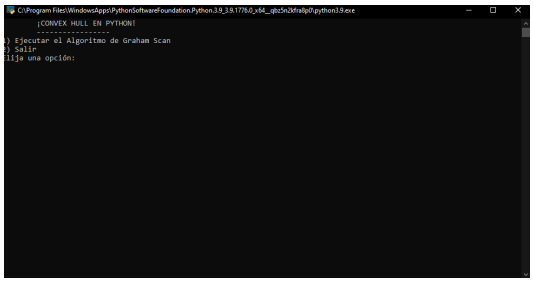

Figura 11: Menú que se muestra al ejecutar el programa.

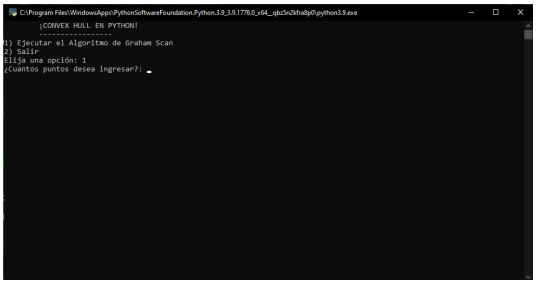

Figura 12: Al seleccionar la opción 1, el programa pedirá que el usuario ingrese la cantidad de puntos que desea que se generen aleatoriamente, mientras que si el usuario decide la opción 2 el programa pasará a finalizarse.

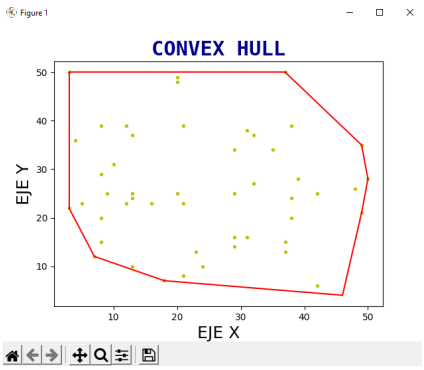

Figura 13: En la imagen se presenta la Convex-Hull de 50 puntos generados de manera aleatoria.

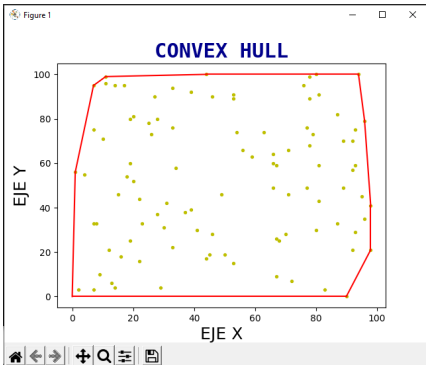

Figura 14: En la imagen se presenta la Convex-Hull de 100 puntos generados de manera aleatoria.

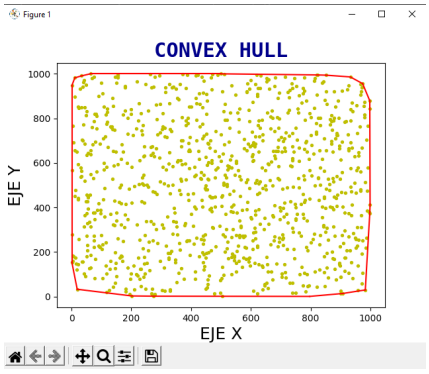

Figura 15: En la imagen se presenta la Convex-Hull de 1000 puntos generados de manera aleatoria.

Puntos	Tiempo de ejecución
50	0.001012300000000188
100	0.0042775999999946634
150	0.009082900000009886
200	0.01469520000000557
250	0.023710600000015347
300	0.026336899999989782
350	0.039792499999997592
400	0.04358150000001615
450	0.05903210000002446
500	0.06148229999996602
550	0.08005760000003193
600	0.07497710000006919
650	0.08026910000000953
700	0.100669900000007081
750	0.11773909999999432
800	0.151745099999971
850	0.14707420000002003
900	0.1594731000000138
950	0.2895121999999901
1000	0.20129229999997733

Figura 16: Finalmente hemos realizado una tabla comparativa en el cual se aprecia como incrementa el tiempo ejecución del código mientras mayor es la cantidad de puntos.

6. CONCLUSIONES

1. En la actualidad se están investigando otros métodos para la aplicación del Reconocimiento facial como:
   - Método de Cascada de Haar [7].
   - PCA o Análisis de componentes principales [11].
   - Momentos invariantes de Hu [6].
   - Contornos activos y snakes (The greedy snake algorithm)[9].
2. El algoritmo de la cápsula convexa no solo es utilizado en el Reconocimiento facial, otras de sus aplicaciones son:
   - Recopilación de datos de sensores disjuntos [2].
   - Coordinación de robot en un sistema de enjambre[16].
3. Los algoritmos de Jarvis y Graham son los más usados en los proyectos por su facil implementación (Dr. Luis Antonio Rivera Escriba [6]).

7. BIBLIOGRAFÍA

[1] Alzubaidi, A. M. N. (2014). Minimum bounding circle of 2d convex hull. *International journal of Science and research*, 3:364–367.

[2] Arvin1nghvi, K. W. J. H. B. L. F. (2021). Sdp-based robust formation-containment coordination of swarm robotic systems with input saturation. `https://link.springer.com/content/pdf/10.1007/s10846-021-01368-4.pdf`. [Online; accessed 29-May-2021].

[3] Blelloch, G. E., Gu, Y., Shun, J., and Sun, Y. (2020). Randomized incremental convex hull is highly parallel. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 103–115.

[4] Chan, T. M. (1996). Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete & Computational Geometry*, 16(4):361–368.

[5] Cheng, J., Foo, S. W., and Krishnan, S. M. (2006). Watershed-presegmented snake for boundary detection and tracking of left ventricle in echocardiographic images. *IEEE Transactions on Information technology in Biomedicine*, 10(2):414–416.

[6] ESCRIBA, L. A. (2014). ClassificaÇÃo de imagens usando mapas auto-organizÁveis. `https://www.researchgate.net/publication/271370798_CLASSIFICACAO_DE_IMAGENS_USANDO_MAPAS_AUTO-ORGANIZAVEIS`. [Online; accessed 18 de junio-2021].

[7] Guevara, M., E. J. . U. W. (2008). Detección de rostros en imágenes digitales usando clasificadores en cascada.

https://revistas.utp.edu.co/index.php/revistaciencia/article/view/3679. [Online; accessed 18 de junio-2021].

[8] Kirkpatrick, D. G. and Seidel, R. (1986). The ultimate planar convex hull algorithm? *SIAM journal on computing*, 15(1):287–299.

[9] López, R. F. N. (2006). Multiple object image segmentation based on genetic active contours. http://cicese.repositorioinstitucional.mx/jspui/handle/1007/2220. [Online, consultado el 20 de junio -2021].

[10] McQUEEN, M. M. and Toussaint, G. T. (1985). On the ultimate convex hull algorithm in practice. *Pattern Recognition Letters*, 3(1):29–34.

[11] Pavón, S. D. (2017). Reconocimiento facial mediante el análisis de componentes principales (pca). http://hdl.handle.net/11441/66514. [Online, consultado el 20 de junio -2021].

[12] Ramaswami, S. (1993). Convex hulls: Complexity and applications (a survey). *Technical Reports (CIS)*, page 264.

[13] Sabilarrusyda, N. A., Basuki, A., and Fathoni, K. (2017). Game mobile application history of uthman ibn affan based on monotone chain convex hull algorithm. In *2017 International Electronics Symposium on Knowledge Creation and Intelligent Computing (IES-KCIC)*, pages 200–205. IEEE.

[14] Snyder, T. L. and Steele, J. M. (1993). Convex hulls of random walks. *Proceedings of the American Mathematical Society*, 117(4):1165–1173.

[15] Williamson, D. P. and Singhvi, D. (2014). ORIE 6300 Mathematical Programming I. https://people.orie.cornell.edu/dpw/orie6300/Lectures/lec03.pdf. [Online; accessed 29-May-2021].

[16] X. Liu, T. Wang, W. J. A. L. y. K. C. (2008). Planificación de encuentro rápida convexa basada en casco para recopilación de datos móviles con demora severa en redes de sensores disjuntos. https://ieeexplore.ieee.org/document/8844855/keywords#keywords. [Online; accessed 18 de junio-2021].

[17] (2018). Implementación del código python del algoritmo graham-scan para calcular el casco convexo. https://blog.csdn.net/john_bian/article/details/85221039. [Online; accessed 29-May-2021].