

Geometria Computacional em C e Python: Aplicações Práticas

Descubra como aplicar geometria computacional na programação.

Explore conceitos, algoritmos e aplicações essenciais para desenvolvedores.



4 Contributors





Introdução à Geometria Computacional: Conceitos Fundamentais

Definição

Estudo de algoritmos para resolver problemas geométricos com computadores.

Objetos

Inclui pontos, segmentos, polígonos, e outras formas geométricas.

Uso Prático

Essencial para gráficos, CAD, jogos, e análise espacial.

Representação de Objetos Geométricos em C e Python

Em C

- Structs para pontos e polígonos
- Manipulação manual de memória
- Alta performance e controle

Em Python

- Classes para gerenciamento fácil
- Bibliotecas como Shapely e NumPy
- Fácil prototipagem e legibilidade

Algoritmos Básicos: Envoltória Convexa e Interseção de Segmentos

1 Envoltória Convexa

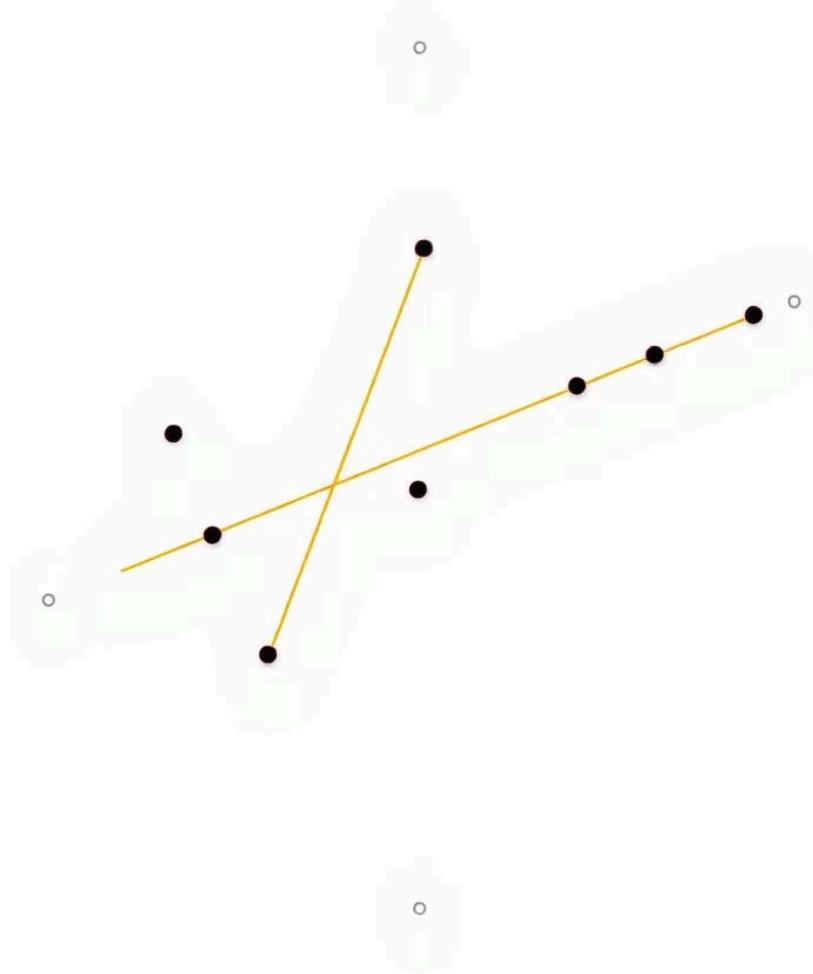
Algoritmo de Graham Scan e Jarvis March para formar seu polígono mínimo.

2 Interseção de Segmentos

Detecta se dois segmentos se cruzam, fundamental para validações geométricas.

3 Importância

Base para aplicações complexas e otimizações posteriores.





Aplicações em Gráficos e Jogos: Detecção de Colisão e Renderização

Detecção de Colisão

Essencial para interações realistas em jogos e simuladores.

Renderização

Usa geometria para desenhar modelos complexos e cenas.

Performance

Algoritmos otimizados garantem fluidez e resposta rápida.

Aplicações em GIS: Análise Espacial e Geoprocessamento

Análise Espacial

Determina relações e padrões entre elementos geográficos.

- Buffer zones
- Sobreposição de polígonos

Geoprocessamento

Processa dados espaciais para planejamento e decisões.

- Transformações geométricas
- Manipulação de mapas digitais



Otimização e Desafios: Desempenho e Precisão Numérica

Desempenho

Reducir complexidade de algoritmos para acelerar o processamento.

Precisão

Lidar com erros numéricos e arredondamentos em cálculos geométricos.

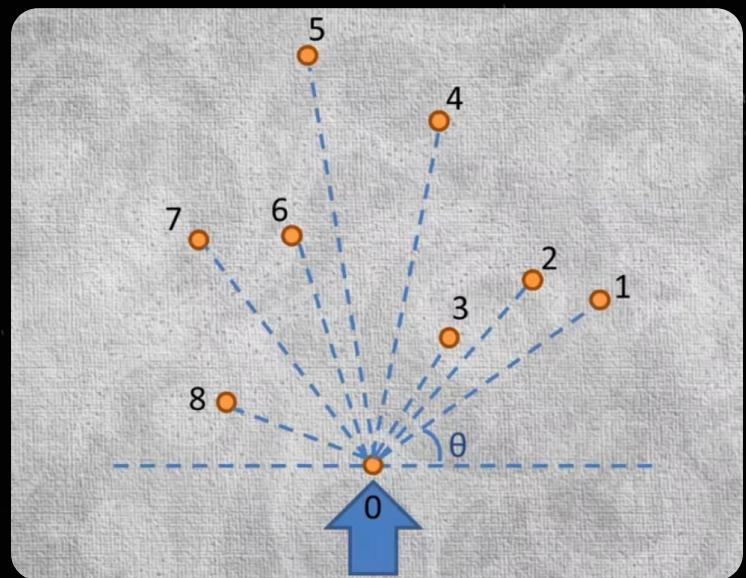
Desafios

Gerenciar estruturas de dados eficientes e garantir robustez.

Graham Scan

- Utilizado para calcular o envoltório convexo de um conjunto de pontos;
- Ordena os pontos em ordem angular (ou da direita para a esquerda) e depois insere os pontos na envoltória nesta ordem
- Complexidade de $O(n \log n)$

Passo a passo - Graham Scan

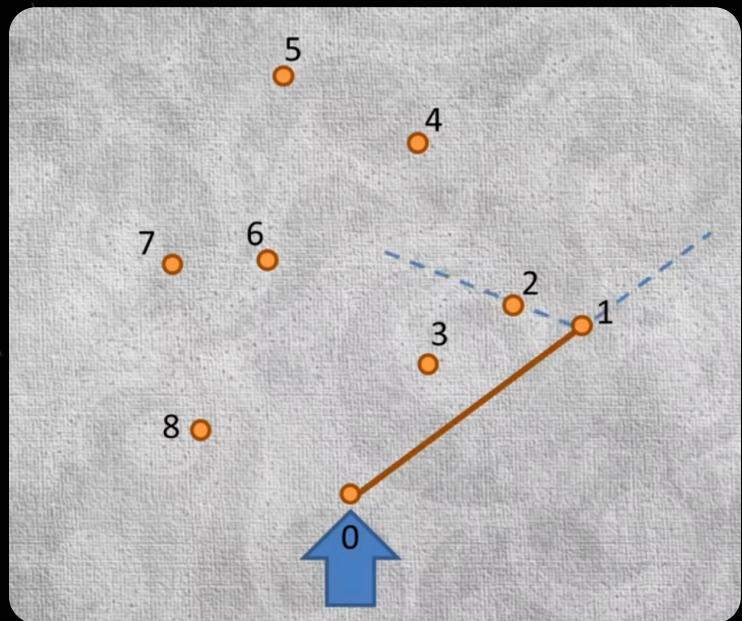


Etapa 1:

- Escolher o ponto com menor angulação e à direita como ponto inicial.
- Ordenar os demais pontos em ordem crescente do ângulo polar em relação ao ponto inicial.
 - Da direita para esquerda.

Pilha: []

Passo a passo - Graham Scan

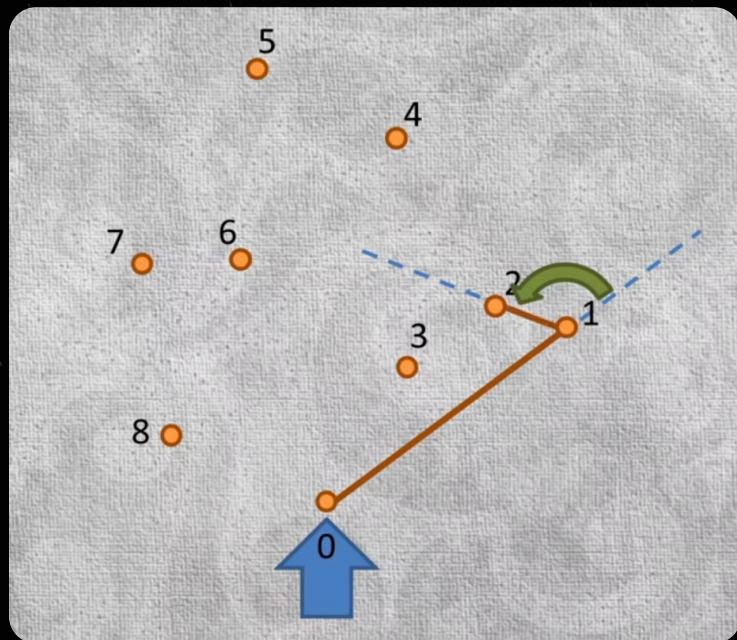


Etapa 2:

- Iniciar a pilha com os dois primeiros pontos ordenados.

Pilha: [0, 1]

Passo a passo - Graham Scan

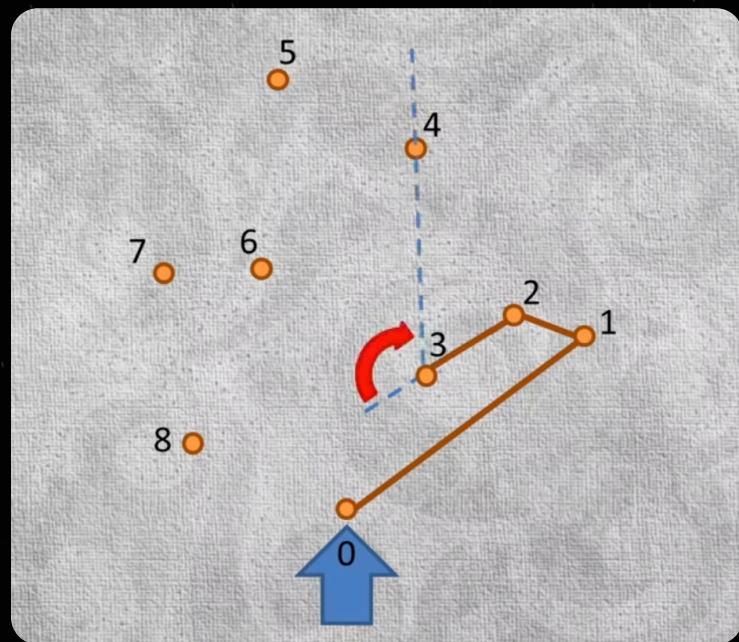


Etapa 3:

- Para cada ponto restante, verificar a orientação formada com os dois pontos do topo da pilha.

Pilha: [0, 1, 2]

Passo a passo - Graham Scan

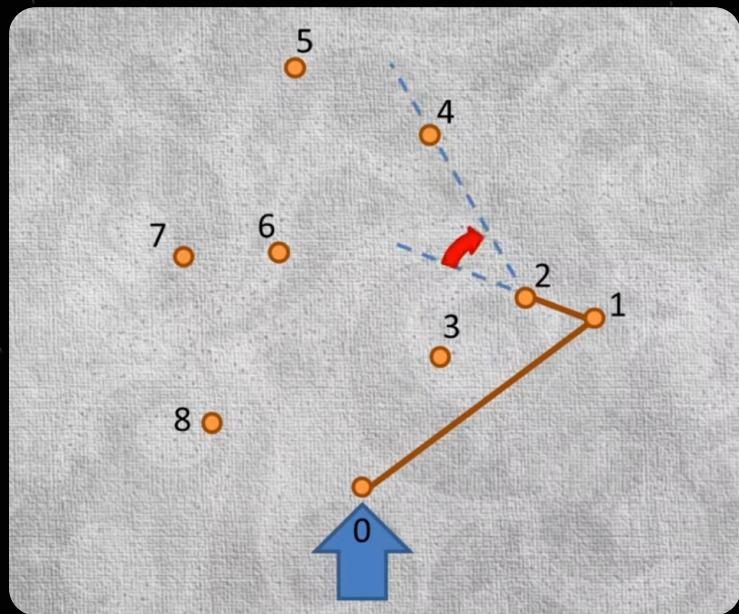


Etapa 3:

- Se formar um giro à esquerda ou colinearidade, remover o ponto do topo da pilha.

Pilha: [0, 1, 2, 3]

Passo a passo - Graham Scan

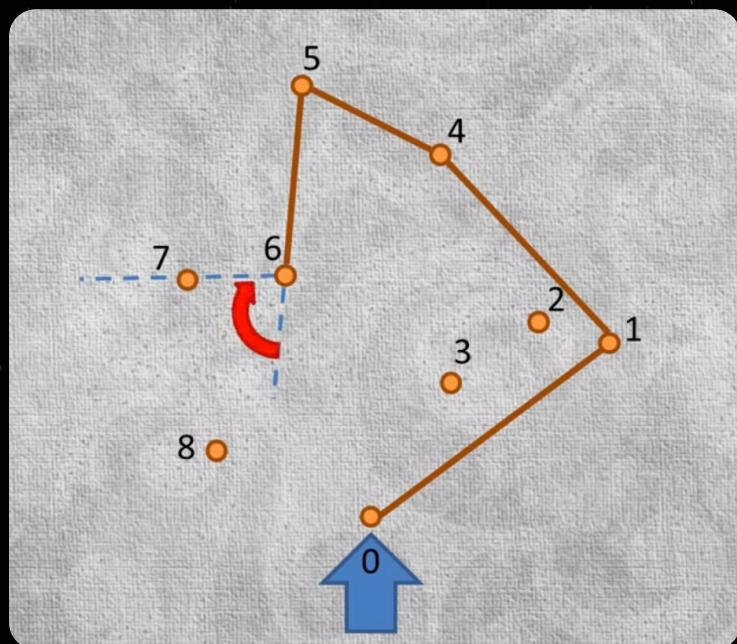


Etapa 2:

- Se formar um giro à esquerda ou colinearidade, remover o ponto do topo da pilha.

Pilha: [0, 1, 2]

Passo a passo - Graham Scan

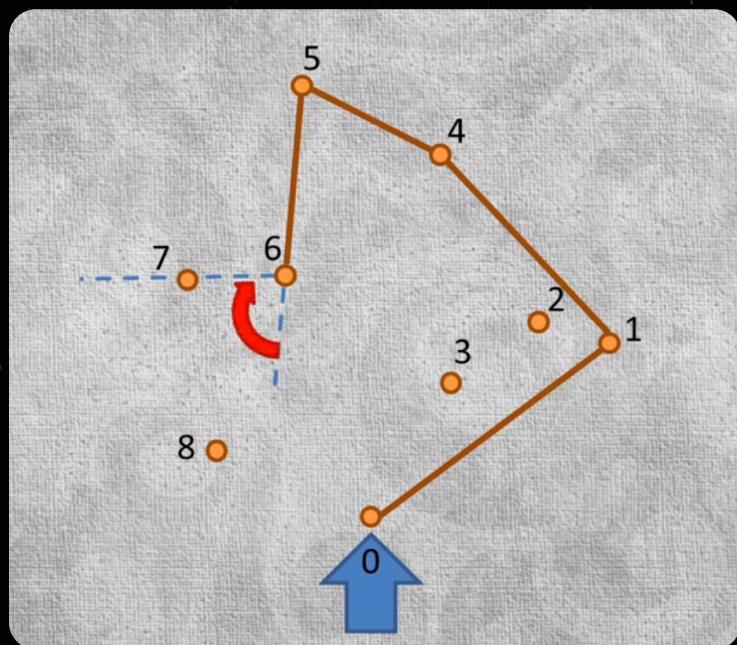


Etapa 2:

- Se formar um giro à esquerda ou colinearidade remover o ponto do topo da pilha.

Pilha: [0, 1, 4, 5, 6]

Passo a passo - Graham Scan

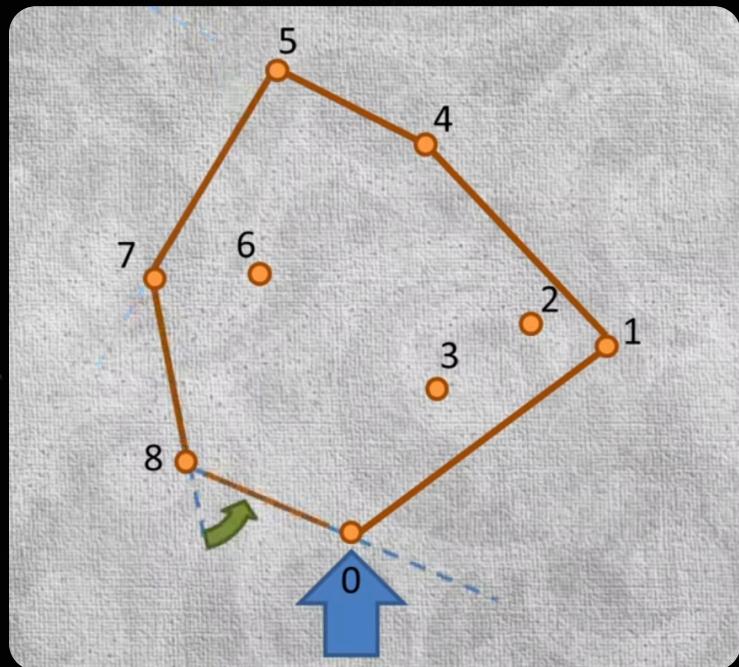


Etapa 2:

- Se formar um giro à esquerda ou colinearidade remover o ponto do topo da pilha.

Pilha: [0, 1, 4, 5, 6]

Passo a passo - Graham Scan



Etapa 2:

- Repetir até percorrer todos os pontos, resultando na envoltória convexa.

Pilha: [0, 1, 4, 5, 6, 7, 8]

Implementação

Funções complementares

```
# Uma classe usada para armazenar as coordenadas x e y dos pontos
class Point:
    def __init__(self, x = None, y = None):
        self.x = x
        self.y = y

# Um ponto global necessário para ordenar os pontos em relação ao primeiro ponto
p0 = Point(0, 0)

# Função utilitária para obter o penúltimo elemento da pilha
def nextToTop(S):
    return S[-2]

# Função utilitária para retornar o quadrado da distância
# entre p1 e p2
def distSq(p1, p2):
    return ((p1.x - p2.x) * (p1.x - p2.x) +
           (p1.y - p2.y) * (p1.y - p2.y))
```

Implementação

Funções complementares

```
25
26     # Para encontrar a orientação do triplo ordenado (p, q, r).
27     # A função retorna os seguintes valores:
28     # 0 --> p, q e r são colineares
29     # 1 --> Sentido horário
30     # 2 --> Sentido anti-horário
31     def orientation(p, q, r):
32         val = ((q.y - p.y) * (r.x - q.x) -
33                 (q.x - p.x) * (r.y - q.y))
34         if val == 0:
35             return 0 # colineares (alinhados)
36         elif val > 0:
37             return 1 # sentido horário
38         else:
39             return 2 # sentido anti-horário
40
```

Implementação

```
# Função principal do fecho convexo
def convexHull(points, quantity_points):
    # Encontrar o ponto mais abaixo
    ymin = points[0].y
    min_index = 0
    for i in range(1, quantity_points):
        y = points[i].y
        if y < ymin or (y == ymin and points[i].x < points[min_index].x):
            ymin = points[i].y
            min_index = i

    # Coloca o ponto mais abaixo na primeira posição
    first_point = points[0]

    points[0] = points[min_index]
    points[min_index] = first_point

    p0 = points[0]

    # Ordena todos os pontos, exceto o primeiro, com base no ângulo e distância ao p0
    def polar_angle(p):
        return math.atan2(p.y - p0.y, p.x - p0.x)

    sorted_points = sorted(points[1:], key=lambda p: (polar_angle(p), distSq(p0, p)))
    points = [points[0]] + sorted_points

    # Remove pontos colineares próximos
    filtered_count = 1
    for i in range(1, quantity_points):
        while i < quantity_points - 1 and orientation(p0, points[i], points[i + 1]) == 0:
            i += 1
        points[filtered_count] = points[i]
        filtered_count += 1

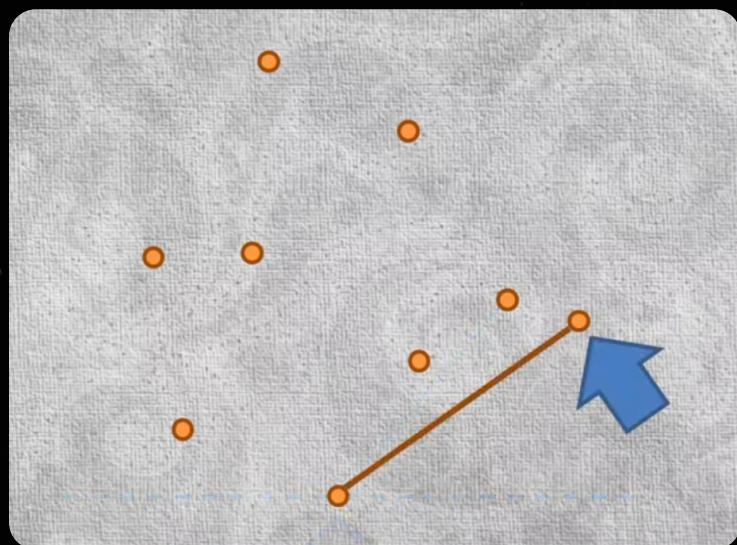
    # Construir o fecho convexo (Repete o processo para os demais pontos)
    stack = [points[0], points[1], points[2]]
    for i in range(3, filtered_count):
        while len(stack) > 1 and orientation(nextToTop(stack), stack[-1], points[i]) != 2:
            stack.pop()
        stack.append(points[i])

    # Imprimir resultado
    while stack:
        p = stack.pop()
        print(f"({p.x}, {p.y})")
```

Algoritmo Jarvis March

- Realiza os cálculos por força bruta;
- Complexidade de $O(nh)$
 - n = número total de pontos.
 - h = número de vértices (pontos que pertencem ao fecho convexo).

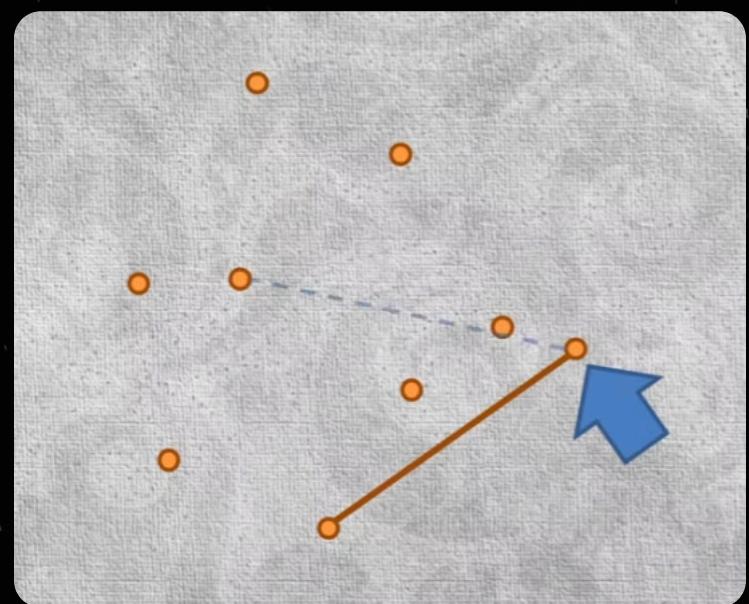
Passo a passo - Jarvis March



Etapa 1:

- Encontre o ponto com a coordenada x mais baixa; este será o ponto inicial da envoltória convexa
 - Não há ordenação prévia
- Defina o ponto atual como esse ponto inicial.

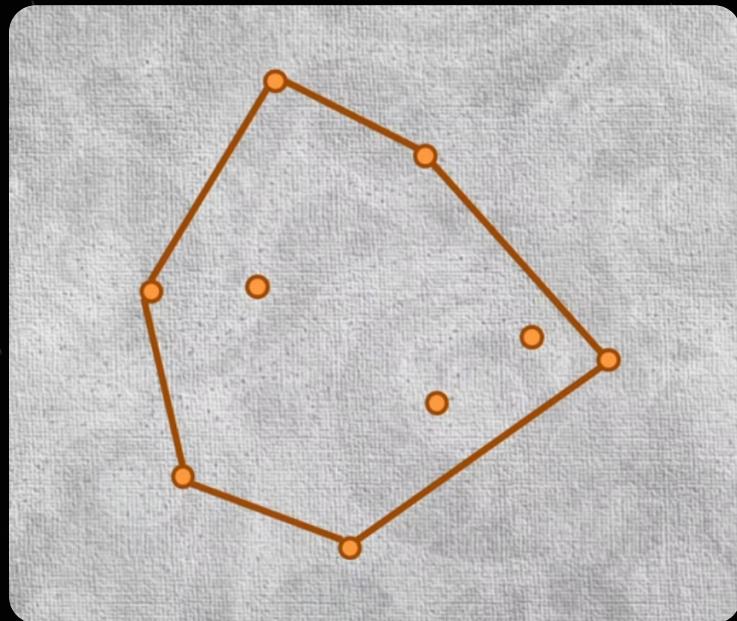
Passo a passo - Jarvis March



Etapa 2:

- Faz uma verificação com todos os pontos para escolher o próximo mais a esquerda (força-bruta).

Passo a passo - Jarvis March



Etapa 3:

- Repete o processo até chegar ao ponto inicial.

Implementação

Complementares

```
def Left_index(points):
    """
    Encontra o ponto mais à esquerda
    """

    minn = 0
    for i in range(1, len(points)):
        if points[i].x < points[minn].x:
            minn = i
        elif points[i].x == points[minn].x:
            if points[i].y > points[minn].y:
                minn = i
    return minn

def orientation(p, q, r):
    """
    Determina a orientação do triplo ordenado (p, q, r).
    A função retorna os seguintes valores:
    0 --> p, q e r são colineares
    1 --> Sentido horário
    2 --> Sentido anti-horário
    """

    val = (q.y - p.y) * (r.x - q.x) - \
          (q.x - p.x) * (r.y - q.y)

    if val == 0:
        return 0
    elif val > 0:
        return 1
    else:
        return 2
```

Implementação

Inicialização

```
def convexHull(points, quantity_points):

    # Devem existir pelo menos 3 pontos
    if quantity_points < 3:
        return

    # Encontra o ponto mais à esquerda
    left_most_point = Left_index(points)

    hull = []
```

Implementação

Continuação

```
"""

Começa a partir do ponto mais à esquerda e continua
movimentando-se no sentido anti-horário até retornar
ao ponto inicial.

"""

current_point = left_most_point
q = 0
while(True):

    # Adiciona o ponto atual ao resultado
    hull.append(current_point)

    """

    Busca por um ponto 'q' tal que a orientation(p, q, x)
    seja anti-horária para todos os pontos 'x'.
    A ideia é manter controle do ponto mais anti-horário
    visitado por último em q. Se algum ponto 'i' for mais
    anti-horário que q, então atualiza q.

    """

    q = (current_point + 1) % quantity_points

    for i in range(quantity_points):
        # Se i for mais anti-horário que o q atual, atualiza q
        if orientation(points[current_point], points[i], points[q]) == 2:
            q = i

    """

    Agora q é o ponto mais anti-horário em relação a p.
    Define p como q para a próxima iteração, de modo que q
    seja adicionado ao resultado 'hull'.

    """

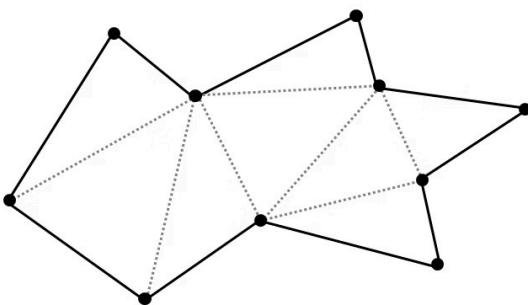
    current_point = q

    # Quando retornar ao primeiro ponto, encerra o laço
    if current_point == left_most_point:
        break

    # Imprime o resultado
    for each in hull:
        print(points[each].x, points[each].y)
```

Triangulação

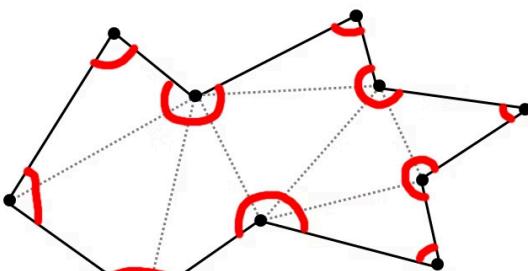
Divisão de um polígono em triângulos sem sobreposição cobrindo toda sua área



- Facilita o cálculo de **área, renderização gráfica e malhas em engenharia**.
- É base para algoritmos de **computação gráfica, simulação física e modelagem 3D**

Ear clipping - passo a passo

Dado um polígono P simples com n vértices, representado por $V(v_0, v_1, \dots, v_{n-1})$, orientado no sentido anti-horário:



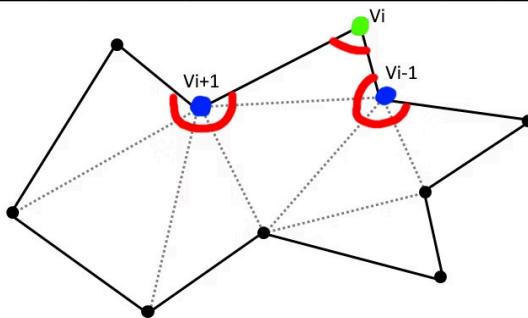
Etapa 1:

Calcular os ângulos internos de cada vértice de P.

- Se o ângulo for **menor que 180°** , o vértice é **convexo**.
- Caso contrário, é um vértice **côncavo**.

Ear clipping - passo a passo

Dado um polígono P simples com n vértices, representado por $V(v_0, v_1, \dots, v_{n-1})$, orientado no sentido anti-horário:



Etapa 2:

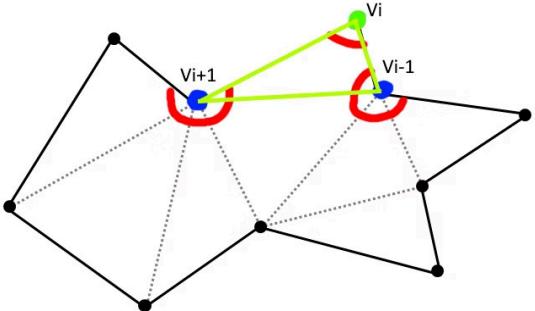
Identificar todos os vértices que são possíveis "pontas de orelhas".

Três vértices consecutivos (v_{i-1}, v_i, v_{i+1}) formam uma orelha se:

1. v_i é um vértice **convexo**;
2. O **fechamento** $C(v_{i-1}, v_i, v_{i+1})$ do triângulo $\triangle(v_{i-1}, v_i, v_{i+1})$ **não contém** **nenhum vértice côncavo** de P (exceto possivelmente v_{i-1} e v_{i+1}).

Ear clipping - passo a passo

Dado um polígono P simples com n vértices, representado por $V(v_0, v_1, \dots, v_{n-1})$, orientado no sentido anti-horário:



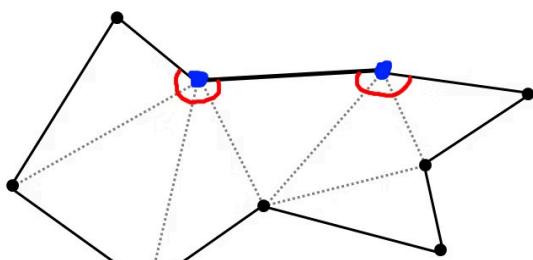
Etapa 3:

Selecionar a ponta de orelha v_i para formar o triângulo $\triangle(v_{i-1}, v_i, v_{i+1})$, **remover o vértice v_i do polígono e atualizar**:

- A conexão entre os vértices vizinhos;
- Seus ângulos internos;
- O status de orelha de v_{i-1} e v_{i+1} .

Ear clipping - passo a passo

Dado um polígono P simples com n vértices, representado por $V(v_0, v_1, \dots, v_{n-1})$, orientado no sentido anti-horário:



Etapa 4:

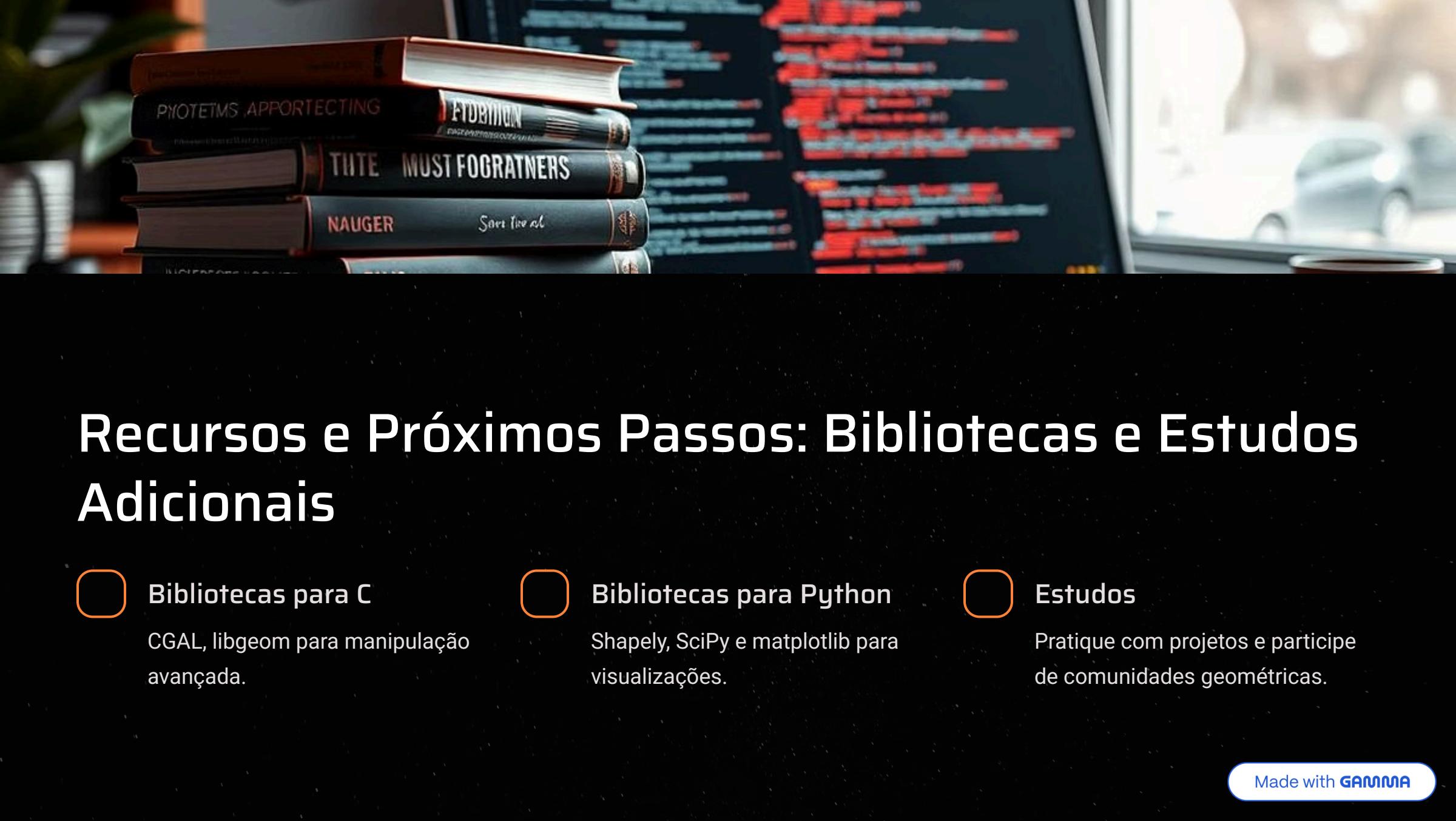
Repetir a Etapa 3 até que **(n-2) triângulos** tenham sido construídos.

As Etapas 1 e 2 são usadas para **inicializar** o polígono.

As Etapas 3 e 4 consistem na **remoção repetida de orelhas e atualização do polígono**.

Conclusão

- A Geometria Computacional é essencial em diversas áreas práticas.
- Permite resolver problemas complexos com algoritmos eficientes.
- Estruturas como **envoltórias convexas, triangulações e interseções** são fundamentais para organizar e processar dados espaciais.
- Compreender e implementar esses algoritmos contribui diretamente para aplicações modernas como gráficos 3D, navegação, robótica e mais.



Recursos e Próximos Passos: Bibliotecas e Estudos Adicionais



Bibliotecas para C

CGAL, libgeom para manipulação avançada.



Bibliotecas para Python

Shapely, SciPy e matplotlib para visualizações.



Estudos

Pratique com projetos e participe de comunidades geométricas.