FACULTADE DE MATEMÁTICAS

**Traballo Fin de Grao**

# Xeometría Computacional

Pedro Chans Fanego

2020/2021

UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

GRAO DE MATEMÁTICAS

**Traballo Fin de Grao**

# Xeometría Computacional

Pedro Chans Fanego

2020/2021

UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

# Traballo proposto

| **Área de Coñecemento: Xeometría e Topoloxía** |
| --- |
| **Título: Xeometría Computacional** |
| **Breve descrición do contido** |
| O traballo consistirá en estudar algúns algoritmos clásicos de xeometría computacional, como triangulación de polígonos, envolventes convexas de puntos, e algunhas das súas consecuencias. |
| **Recomendacións** |
| |
| **Outras observacións** |
| |

# Índice xeral

**Resumo** VIII

**Introdución** XI

**1. Triangulación dun polígono** 1

1.1. Existencia dunha triangulación . . . . . 3

1.2. Área dun polígono . . . . . 5

1.2.1. Área dun triángulo . . . . . 5

1.2.2. Área dun polígono convexo . . . . . 6

1.2.3. Área dun polígono non convexo . . . . . 7

1.3. Algoritmo de triangulación de polígonos . . . . . 10

1.3.1. Cálculo de áreas . . . . . 11

1.3.2. Intersección de segmentos . . . . . 13

1.3.3. Diagonais . . . . . 15

1.3.4. Triangulación . . . . . 17

1.3.5. Outros algoritmos de triangulación . . . . . 19

**2. Envolvente convexa** 21

2.1. Definicións e propiedades básicas . . . . . 21

2.2. Implementación dalgúns algoritmos . . . . . 25

2.2.1. Puntos non extremos . . . . . 26

2.2.2. Segmentos Extremos . . . . . 28

2.2.3. Embalaxe de agasallo . . . . . 29

2.2.4. QuickHull . . . . . 32

2.3. Algoritmo de Graham . . . . . 35

2.3.1. Inicialización . . . . . 36

2.3.2. Colinearidades . . . . . 36

2.3.3. Implementación . . . . . 37

2.3.4. Cota inferior da complexidade . . . . . 39

**3. Diagramas de Voronoi** **41**

3.1. Definicións e propiedades básicas . . . . . 41

3.1.1. Semiplanos . . . . . 42

3.1.2. Tamaño do diagrama . . . . . 44

3.2. Triangulación de Delaunay . . . . . 45

3.3. Algoritmos . . . . . 49

3.3.1. Intersección de semiplanos . . . . . 49

3.3.2. Construción incremental . . . . . 49

3.3.3. Divide e vencerás . . . . . 50

3.3.4. Algoritmo de Fortune . . . . . 50

**4. Anexo** **53**

4.1. Estruturas de datos . . . . . 53

4.1.1. Lista . . . . . 53

4.1.2. Lista dobremente enlazada . . . . . 55

4.1.3. Lista circular dobremente enlazada . . . . . 56

4.2. Programas principais . . . . . 58

4.2.1. Point . . . . . 58

4.2.2. Segment . . . . . 61

4.2.3. Triangle . . . . . 63

4.2.4. Polygon . . . . . 64

4.2.5. Convex Hull . . . . . 66

**Bibliografía** **71**

## Resumo

O obxectivo deste traballo será tratar tres temas da xeometría, tales como a triangulación de polígonos, a envolvente convexa e os diagramas de Voronoi. En cada un deles aportaremos os conceptos máis básicos, partiremos dunha serie de resultados teóricos, e postularemos un problema ao que daremos resposta mediante a xeometría computacional. En todos eles tentaremos detallar a explicación aos algoritmos que abordan dito problema, e nalgúns elaboraremos e engadiremos o código de ditos algoritmos implementado na linguaxe de programación Python.



## Abstract

The goal of this work will be to deal with three topics of geometry, such as triangulation of polygons, convex hull and Voronoi diagrams. In each of them we will present the most basic concepts, we will start with some theoretical results, and we will postulate a problem which will be answered by means of computational geometry. In all of them we will try to detail the explanation of the algorithms that solve the problem, and in some of them we will develop and add the code of these algorithms implemented in the Python programming language.

# Introdución

A xeometría computacional é unha rama das ciencias da computación dedicada ao estudio de algoritmos que poden ser expresados en termos da xeometría. Poderiamos dicir tamén que é o arte de resolver problemas conceptualmente sinxelos empregando a menor cantidade de recursos e tempo posibles. Entre as áreas nas que se aplican os resultados desta disciplina atópanse a enxeñaría, sistemas de posicionamento global, robótica, modelado xeométrico e recoñecemento de patróns. A xeometría computacional é independente da tecnoloxía das máquinas de computación, pero a complexidade computacional xoga un papel vital no seu estudo. Se ben podemos estar a traballar con decenas ou centos de millóns de puntos, a diferencia entre $O(n^2)$ e $O(n \log n)$ pode significar a diferencia entre días ou segundos de computación.

Neste traballo, centrarémonos nunha rama ampla da xeometría computacional, que será a xeometría computacional combinatoria, na que os obxectos xeométricos serán tratados como entidades discretas. O obxectivo principal do traballo en cada capítulo será partir dun problema esencialmente teórico da xeometría e desenvolver unha serie de resultados que nos permitirán, cando menos, dar unha explicación dos algoritmos que resolven o problema na práctica seguindo os pasos de O'Rourke [1] e Preparata & Shamos [2]. Implementaremos o código dos algoritmos que nos interese en Python, diferindo de gran parte da bibliografía na que nos apoiamos e aportando así exclusividade ao traballo. Esta elección vén motivada polo nivel da linguaxe, tempo de compilación, a súa fácil interpretación, simplicidade e lexibilidade de código. É unha das linguaxes máis empregadas na actualidade e é por isto que ten maior soporte, ademais de aportar modernidade.

Gran parte da xeometría computacional realiza os seus cálculos en obxectos xeométricos coñecidos como polígonos, estes son unha representación conveniente para moitos obxectos do mundo real. Ás veces os polígonos por si mesmos son obxectos complicados e resulta máis sinxelos entendelos como unha composición de pezas máis simples, o que nos leva ao tópico do primeiro capítulo: a partición de polígonos. Polo tanto, a principal motivación da triangulación de polígonos é unha cuestión tanto práctica como teórica. A triangulación de polígonos permítenos coñecelos máis a fondo e deducir información deles. Por exemplo,

simplifica o cálculo de áreas e serve para resolver o problema da galería de arte, cuxo concepto clave é a visibilidade entre puntos. Neste traballo falaremos da existencia de triangulación, que é a primeira pregunta vital ante este concepto. Desenvolveremos fórmulas do cálculo de áreas de polígonos que ademais terán como obxectivo a creación de funcións booleanas que darán resposta á posición relativa de puntos no espazo. Estas funcións serán empregadas ao remate do capítulo co obxectivo de implementar a triangulación á computación, escollendo as listas enlazadas como a estrutura de datos sobre as que aplicar ditas funcións, xa que serán elas as que representen aos nosos polígonos.

No segundo capítulo falaremos da envolvente convexa, un dos temas con maior antigüidade dentro da historia da xeometría computacional. Entre as súas aplicacións podemos atopar a evitación de obstáculos para robots, resolver o problema da menor área rectangular contendo a un polígono, ou clasificación de figuras. Aquí non abarcaremos tanto o campo matemático, no que daremos os conceptos básicos e necesarios, como o computacional. Comezaremos con algoritmos intuitivos e pouco eficientes, cun obxectivo didáctico. Posteriormente pasaremos a algoritmos máis realistas como a embalaxe de agasallo, ou QuickHull. Finalizaremos detallando o algoritmo máis importante nesta sección, o algoritmo de Graham. O interese neste algoritmo vén motivado pola súa complexidade computacional, da que tamén daremos unha cota inferior e superior xustificada.

Para o remate falaremos dos diagramas de Voronoi, unha estrutura xeométrica que nos ofrece toda a información que nos gustaría ter sobre a proximidade nun conxunto de puntos. Entre as súas aplicacións están a detección e a expansión de lume nun bosque, localización de comercios e planificación de rutas para robots. Este é un capítulo esencialmente teórico, onde trataremos a súa definición e desenvolveremos resultados importantes, en relación tamén ao dual dun diagrama de Voronoi, e falaremos da triangulación de Delaunay. Para finalizar, esquematizaremos os algoritmos sen entrar en detalle, e falaremos da súa complexidade computacional.

Ao remate do traballo podemos atopar un anexo con todo o código detallado que se emprega, incluíndo os obxectos empregados e as súas funcións e propiedades, engadindo tamén o código das estruturas de datos empregadas á hora de almacenar información.

# Capítulo 1

# Triangulación dun polígono

Neste capítulo abordaremos a triangulación de polígonos, mostraremos e probaremos a súa existencia. Implementaremos o cálculo dunha triangulación á computación en Python seguindo os pasos de O'Rourke [1]. Para iso será necesario tamén facer un inciso previo sobre o cálculo de áreas en polígonos. Para comezar a falar da triangulación dun polígono, cuxo interese reside en dividir o polígono en elementos máis simples, falaremos primeiro do que é un polígono e cales son os elementos que o forman. Daremos unhas definicións sinxelas que servirán como preliminares:

**Definición 1.1.** Un *segmento ab* é un subconxunto pechado dunha liña que conten todos os puntos da liña entre $a$ e $b$.

$$ab = \{(1-t)a + tb \colon t \in [0,1]\}$$

*Observación* 1.2. Unha curva é:

- Simple se non ten autointerseccións.
- Pechada se é homeomorfa a unha circunferencia.

**Definición 1.3.** Un *polígono* é a rexión do plano limitada por unha colección finita de segmentos $e_0, e_1, e_2, \ldots, e_{n-1}$ (chamados *lados do polígono*) que forman unha curva simple pechada.

**Definición 1.4.** A intersección de cada par de segmentos adxacentes recibe o nome de *vértice* e denótase como $v_{i+1} = e_i \cap e_{i+1}$.

Tendo isto, pode existir unha idea intuitiva de a que nos referimos cando falamos do interior e o exterior do polígono, formalizarémolo:

**Teorema 1.5 (Teorema da curva de Jordan).** *Toda curva simple pechada divide o plano en dúas compoñentes conexas.*

**Definición 1.6.** Chamaremos *exterior do polígono* á compoñente conexa do plano non limitada dada polo teorema anterior, e *interior do polígono* á compoñente limitada.

*Observación* 1.7. Os segmentos non adxacentes non se intersecan.

Faremos agora unha clasificación entre os diferentes vértices que poden existir nun polígono.

**Definición 1.8.** Diremos que un vértice $v_{i+1}$ dun polígono é:

1. *Convexo* se o ángulo interior ao polígono que forman os lados adxacentes $e_i, e_{i+1}$ verifica $\sphericalangle(e_i, e_{i+1}) \leq \pi$.

2. *Estritamente convexo* se o ángulo interior ao polígono que forman os lados adxacentes $e_i, e_{i+1}$ verifica $\sphericalangle(e_i, e_{i+1}) < \pi$.

3. *Cóncavo* se o ángulo interior ao polígono que forman os lados adxacentes $e_i, e_{i+1}$ verifica $\sphericalangle(e_i, e_{i+1}) \geq \pi$.

4. *Estritamente cóncavo* se o ángulo interior ao polígono que forman os lados adxacentes $e_i, e_{i+1}$ verifica $\sphericalangle(e_i, e_{i+1}) > \pi$.

Como últimas nocións básicas, falaremos do concepto de diagonal e de orella. Son elementos chave na división de polígonos en subpolígonos e serán de gran interese na triangulación.

**Definición 1.9.** Unha *diagonal* dun polígono $P$ é un segmento entre dous vértices $a$ e $b$ que verifica:

1. $ab \subseteq P$.

2. $ab \cap \partial P \subseteq \{a, b\}$.

**Definición 1.10.** Tres vértices consecutivos $a, b, c$ dun polígono forman unha *orella* do polígono se $ac$ é unha diagonal, neste caso dicimos que $b$ é a *punta da orella*. Dicimos que dúas orellas *non se solapan* se os triángulos interiores son disxuntos.

## 1.1. Existencia dunha triangulación

A veces os polígonos serán obxectos complicados, e será máis sinxelo entender o polígono como unha división de obxectos máis simples. Neste capítulo veremos que eses obxectos máis simples poden ser triángulos, e esta división é o que chamaremos triangulación. Daremos unha definición máis precisa:

**Definición 1.11.** Unha triangulación dun polígono é unha división nun conxunto de triángulos que verifican as seguintes condicións:

1. A unión de todos os triángulos é igual ao polígono orixinal.
2. Os vértices dos triángulos son os vértices do polígono orixinal.
3. Calquera parella de triángulos é disxunta ou comparte unicamente un vértice ou un lado.

A pregunta máis natural ante esta definición é: ¿todos os polígonos admiten unha triangulación? Neste capítulo probaremos que a resposta a esta pregunta é afirmativa, e para eso primeiro probaremos a existencia dunha diagonal:

**Lema 1.12.** *Todo polígono ten polo menos un vértice estritamente convexo.*

*Demostración.* Consideramos os vértices do polígono $v_0, v_1, \ldots, v_{n-1}$ orientados en sentido antihorario, de modo que se trazamos a liña entre $v_i$ e $v_{i+1}$, o interior do polígono quede á esquerda desa liña considerando o sentido $v_i v_{i+1}$. Seleccionamos $v$ o vértice coa menor coordenada $y$ respecto ao sistema de coordenadas $XY$; se hai varios, seleccionamos o que teña a maior coordenada $x$, e seleccionamos tamén $e$, o lado previo a ese vértice en sentido antihorario, de modo que o polígono quede sobre $e$ (ou á esquerda supoñendo que avanzamos cara $v$). Temos agora con todas estas condicións que $v$ ten que ser un xiro á esquerda. E polo tanto $v$ é estritamente convexo. □

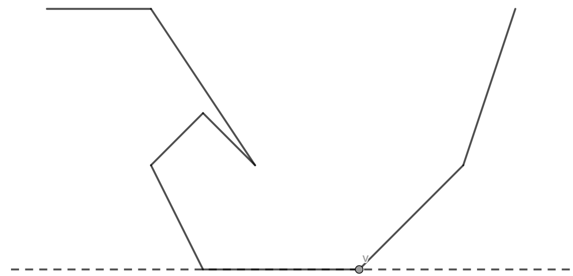

Figura 1.1: O vértice $v$ é estritamente convexo

**Lema 1.13** (Meisters [4])**.** *Todo polígono de $n \geq 4$ vértices ten unha diagonal.*

*Demostración.* Sexa $v$ un vértice estritamente convexo, cuxa existencia vén garantida polo Lema 1.12. Sexan $a$ e $b$ vértices adxacentes a $v$, se $ab$ é unha diagonal; temos rematado. Supoñamos que $ab$ non é diagonal; entón ou ben é exterior a $P$ ou ben interseca a $\partial P$.

En calquera caso, como $n > 3$, o triángulo pechado $\triangle avb$ contén polo menos un vértice de $P$ distinto de $a, v, b$. Chamaremos $x$ ao que cumpla esta condición e sexa o máis próximo a $v$ coa distancia medida ortogonalmente a $ab$. Polo tanto $x$ é o primeiro vértice en $\triangle avb$ intersecado por un paralelo a $ab$ movéndose dende $v$ a $ab$. Agora temos que $vx$ é unha diagonal de $P$, xa que o polígono que resulta da intersección de dito paralelo co polígono inicial e os lados que inclúen a $v$ como vértice non ten no seu interior ningún punto de $\partial P$. □

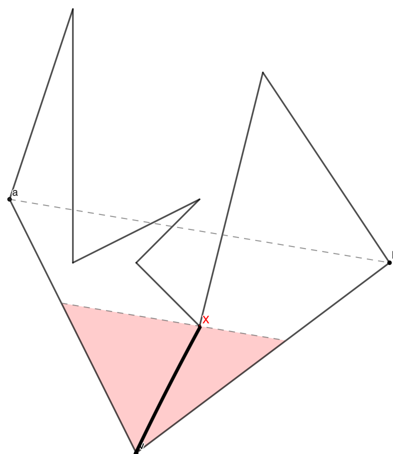


Figura 1.2: Proceso da demostración nun polígono

Vistos os lemas previos, estamos listos para proporcionar a proba da existencia dunha triangulación:

**Teorema 1.14** (Triangulación [1])**.** *Todo polígono $P$ de $n$ vértices admite unha partición en triángulos mediante o engadido de cero ou máis diagonais. Ademais, cada triangulación dun polígono $P$ de $n$ vértices usa $n - 3$ diagonais e consiste en $n - 2$ triángulos.*

*Demostración.* Farémolo por inducción:

Se $n = 3$, o polígono é un triángulo, e o teorema dáse trivialmente.

Sexa $n \geq 4$, $d = ab$ diagonal de $P$ como garante o lema anterior. Como $d$ por definición só interseca a $\partial P$ nos seus puntos inicial e final, particiona $P$ en dous polígonos que empregan a $d$ como lado e que teñen menos de $n$ lados. De feito, se estes polígonos teñen

$t$ e $s$ vértices, verifícase $t + s = n + 2$ (xa que os subpolígonos comparten 2 vértices). Aplicando a hipótese de indución temos que os subpolígonos admiten triangulación e polo tanto tamén o noso polígono $P$. Tendo en conta a diagonal engadida, o número de diagonais é:

$$s - 3 + t - 3 + 1 = s + t - 5 = n - 3.$$

E o número de triángulos:

$$s - 2 + t - 2 = n - 2. \square$$

**Corolario 1.15.** *A suma dos ángulos internos dun polígono de $n$ vértices é $(n-2)\pi$.*

## 1.2. Área dun polígono

Nesta sección veremos unha forma sinxela de implementar o cálculo da área dun polígono. Isto seranos útil para calcular se un punto está contido nun semiplano, calcular interseccións entre segmentos, e finalmente para triangular un polígono.

### 1.2.1. Área dun triángulo

A área dun triángulo é un medio da base pola altura, pero este xeito de calcular a área non nos será útil computacionalmente, xa que non sempre será sinxelo atopar a altura.

Sabemos que o módulo do produto vectorial entre dous vectores é a área do paralelogramo que ten como lados ditos vectores. Agora ben, se temos as coordenadas dos vértices do triángulo $a, b, c \in \mathbb{R}^3$ (onde podemos tomar $a, b, c \in \mathbb{R}^2$, xa que o noso triángulo é plano) podemos definir dous vectores que serán dous lados do noso triángulo: $A = b - a$ e $B = c - a$, bastará enton con calcular $\frac{1}{2}|A \times B|$:

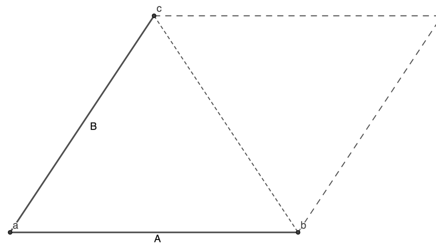


Figura 1.3: Produto vectorial e área dun paralelogramo.

$$\begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ A_0 & A_1 & 0 \\ B_0 & B_1 & 0 \end{vmatrix} = (A_0B_1 - A_1B_0)\vec{k}.$$

Polo tanto a área do triángulo será:

$$A(T) = \frac{1}{2}|A_0B_1 - A_1B_0|.$$

Empregando agora que $A = b - a$ e $B = c - a$:

$$A(T) = \frac{1}{2}|(b_0 - a_0)(c_1 - a_1) - (c_0 - a_0)(b_1 - a_1)|$$

Agora si, temos unha expresión da área do triángulo en función das coordenadas dos seus vértices, o que será sinxelo de engadir á computación.

*Observación* 1.16. En xeral imos considerar os vértices de calquera polígono ordenados en sentido antihorario. Polo tanto, podemos eliminar o valor absoluto da fórmula anterior mentres isto se verifique. Observemos tamén que se o facemos do xeito contrario, ordenando os vértices en sentido horario, obteremos o valor negativo da área de dito triángulo. Empregaremos isto en resultados posteriores. Polo tanto, interesaranos eliminar o valor absoluto de dita fórmula.

### 1.2.2. Área dun polígono convexo

Todo polígono convexo admite unha triangulación trivial, escollendo un vértice $v_0$ e trazando diagonais que teñan como punto inicial $v_0$ e como punto final cada un dos demais vértices non adxacentes.

É sinxelo verificar entón a definición de diagonal. Por ser polígono convexo todos os segmentos trazados están contidos no polígono e só intersecan á fronteira do polígono nos puntos inicial e final. Vemos tamén que o número de diagonais obtidas por este método son $n - 3$, o máximo que admite un polígono de $n$ vértices. Logo o que temos é unha triangulación do polígono.

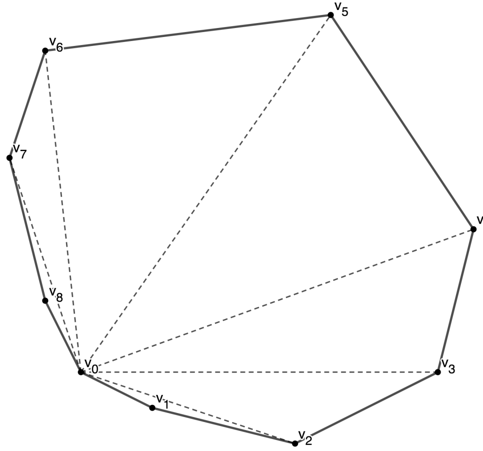


Figura 1.4: Triangulación dun polígono convexo polo método descrito.

Tendo isto en conta e empregando a nosa fórmula para calcular a área dun triángulo, temos que, para $P$ o noso polígono convexo, e $A(v_i, v_j, v_k)$ a área dun triángulo con vértices $v_i, v_j, v_k$:

$$A(P) = A(v_0, v_1, v_2) + A(v_0, v_2, v_3) + \cdots + A(v_0, v_{n-2}, v_{n-1}) = \sum_{i=1}^{n-2} A(v_0, v_i, v_{i+1}).$$

**Exemplo 1.17.** Sexa $Q = (a, b, c, d)$ un cuadrilátero convexo:

$$\begin{aligned} A(Q) = A(a, b, c) + A(a, c, d) = A(d, a, b) + A(d, b, c) = \\ = a_0b_1 - a_1b_0 + a_1c_0 - a_0c_1 + b_0c_1 - c_0b_1 + \\ + a_0c_1 - a_1c_0 + a_1d_0 - a_0d_1 + c_0d_1 - d_0c_1 \end{aligned}$$

Como podemos ver na expresión anterior, $a_1c_0 - a_0c_1$ aparece en ambos sumandos con signo contrario, e polo tanto cancélanse. Este sumando corresponde á diagonal $ac$ que se cancela.

Sexan $x_i, y_i$ as coordenadas de cada vértice $v_i$, podemos reescribir a nosa fórmula como:

$$2A(P) = \sum_{i=0}^{n-1} (x_i y_{i+1} - y_i x_{i+1})$$

A continuación veremos que esta fórmula tamén é válida para polígonos non convexos.

### 1.2.3. Área dun polígono non convexo

**Exemplo 1.18.** Sexa o noso cuadrilátero non convexo $Q = (a, b, c, d)$ como o da Figura 1.5.

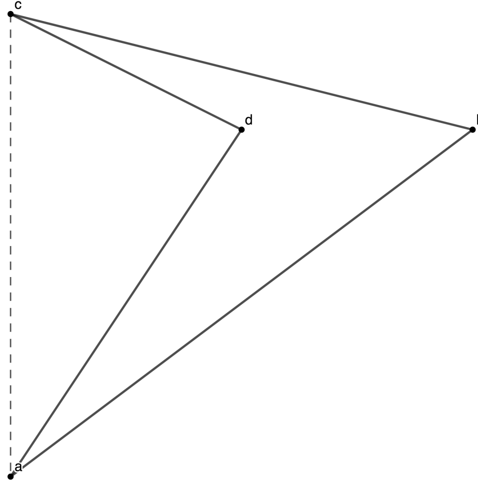


Figura 1.5: Cuadrilátero non convexo Q.

Aplicamos a fórmula anterior:

$$A(Q) = A(a, b, c) + A(a, c, d).$$

O que temos agora é que, empregando a nosa fórmula da área dun triángulo (sen valor absoluto), $A(a, b, c)$ ten un signo positivo por ser vértices ordenados en sentido antihorario, $A(a, c, d)$ ten signo negativo por ser vértices ordenados en sentido horario, Polo tanto, estamos restándolle a $A(a, b, c)$ a área do triángulo que forman os vértices $a, c, d$; resultando así a área do noso cuadrilátero.

Formalizaremos agora o que vimos no exemplo anterior, e xeneralizaremos o método para calcular a área dun polígono non convexo empregando un punto externo arbitrario $p$.

**Lema 1.19.** *Se $T = \triangle abc$ é un triángulo con vértices orientados en sentido antihorario, e $p$ é un punto arbitrario do plano, entón:*

$$A(T) = A(p, a, b) + A(p, b, c) + A(p, c, a).$$

Omitimos esta demostración xa que a idea é seguir un razoamento similar ao do exemplo anterior. Basta con considerar o triángulo de cada sumando e observar o seu signo para interpretar xeometricamente cal o resultado da área final calculada ao sumar os tres termos.

**Teorema 1.20** (Área dun polígono [1]). *Sexa $P$ un polígono (non necesariamente convexo) con vértices $v_0, v_1, ..., v_{n-1}$ ordenados en sentido antihorario, e sexa $p$ un punto calquera do plano, entón:*

$$A(P) = A(p, v_0, v_1) + A(p, v_1, v_2) + ... + A(p, v_{n-2}, v_{n-1}) + A(p, v_{n-1}, v_0). \tag{1.1}$$

*Ademais, se $v_i = (x_i, y_i)$, a seguinte expresión é equivalente a:*

$$2A(P) = \sum_{i=0}^{n-1}(x_i y_{i+1} - y_i x_{i+1}) = \sum_{i=0}^{n-1}(x_i + x_{i+1})(y_{i+1} - y_i). \tag{1.2}$$

*Demostración.* Probarémolo por indución no número de vértices $n$ de $P$. O caso $n = 3$ queda garantido polo Lema 1.19. Supoñamos entón que a ecuación (1.1) se verifica para todos os polígonos de $n-1$ vértices, e sexa $P$ un polígono de $n$ vértices Como sabemos que $n \geq 4$, polo Lema 1.13 teremos que o polígono ten, cando menos, unha diagonal. Se triangulamos o polígono obteremos que polo menos un dos triángulos é unha orella. Agora podemos renomear os vértices de $P$ de xeito que a nosa orella sexa $E = (v_{n-2}, v_{n-1}, v_0)$.

Sexa $P_{n-1}$ o polígono obtido ao eliminar $E$. Por hipótese de indución:

$$A(P_{n-1}) = A(p, v_0, v_1) + ... + A(p, v_{n-3}, v_{n-2}) + A(p, v_{n-2}, v_0),$$

e polo Lema 1.19:

$$A(E) = A(p, v_{n-2}, v_{n-1}) + A(p, v_{n-1}, v_0) + A(p, v_0, v_{n-2}).$$

E como $A(P) = A(P_{n-1}) + A(E)$, temos que:

$$\begin{aligned} A(P) = A(p, v_0, v_1) + ... + A(p, v_{n-3}, v_{n-2}) + A(p, v_{n-2}, v_0) + \\ + A(p, v_{n-2}, v_{n-1}) + A(p, v_{n-1}, v_0) + A(p, v_0, v_{n-2}). \end{aligned}$$

Decatémonos de que $A(p, v_0, v_{n-2}) = -A(p, v_{n-2}, v_0)$; polo tanto eses termos cancélanse e chegamos a ecuación que procurabamos.

A expresión (1.2) obtense expandindo a fórmula da área dun triángulo e cancelando termos. Podemos reagrupalos e volver a cancelar termos para chegar á última igualdade de dita expresión. □

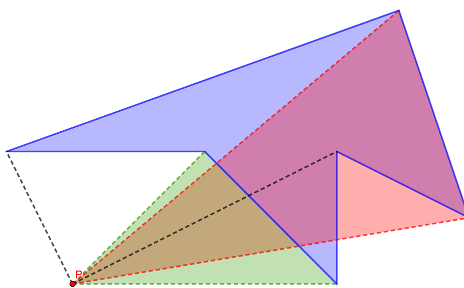

Figura 1.6: Ilustración da demostración nun exemplo, onde o triángulo vermello é un sumando positivo da fórmula e o verde é un sumando negativo.

*Observación* 1.21. Na ecuación (1.2) atoparemos máis interesante empregar a última expresión xa que na maioría de implementacións á computación é máis eficiente ao tratarse de dúas sumas e un produto en lugar de dous produtos e unha suma.

Visto este último teorema, temos todos os resultados necesarios para implementar á computación a triangulación dun polígono, cuxo funcionamento explicaremos na seguinte sección.

## 1.3. Algoritmo de triangulación de polígonos

Agora que temos todos os conceptos necesarios falaremos da implementación da triangulación dun polígono á programación en Python.

Para comezar, definiremos as clases `Pair`, `Point` e `Segment`, onde os pares e os puntos terán como atributos as súas coordenadas, e os segmentos terán como atributos os seus extremos. Definiremos tamén a clase `Vertex`, cuxa única diferencia respecto á clase `Point` será o atributo `is_ear`, que nos dará información sobre se ese vértice é a punta dunha orella dun polígono. Os polígonos `Polygon` terán como atributos iniciais un array dos vectores do mesmo, onde cada vector será definido mediante a clase `Vertex`. Definiremos tamén unha clase aparte para os triángulos `Triangle`. Será de gran utilidade que a estrutura de datos na que almacenemos os vértices do polígono sexa unha lista enlazada.

Todas as clases e estruturas de datos mencionadas nos parágrafos anteriores están detalladas ao completo no anexo do traballo.

```python
class _Pair:
    __slots__ = ('__x', '__y')

    def __init__(self, x, y):
        self.__x = x
        self.__y = y

    def __repr__(self):
        return f'{type(self).__qualname__}({self.x!r}, {self.y!r})'

    def __eq__(self, other):
        if isinstance(other, type(self)):
            return (self.x, self.y) == (other.x, other.y)
        else:
            return NotImplemented

class Point(_Pair):
    __slots__ = ()
```

```
class Segment():
    def __init__(self, p, q):
        self.start = Point(*p)
        self.end = Point(*q)

    def __repr__(self):
        return f'{type(self).__qualname__}({self.start!r},␣{self.end!r})'

    def __eq__(self, other):
        return (
            isinstance(other, type(self)) and

            self.start == other.start and self.end == other.end
        )

    def opposite(self):
        return type(self)(self.end, self.start)
```

```
class Vertex(Point):
    def __init__(self, x, y):
        super().__init__(x, y)
        self.is_ear = None

class Polygon():
    def __init__(self, vertices):
        self.vertices = LinkedList(Vertex(*vertex) for vertex in vertices)

    def __repr__(self):
        return f'{type(self).__qualname__}({list(self.vertices)!r})'
```

```
class Triangle:
    def __init__(self, a, b, c):
        self.a = Point(*a)
        self.b = Point(*b)
        self.c = Point(*c)
```

Tendo todo isto como premisa, comezaremos a falar de todas as funcións que necesitaremos para triangular o polígono.

### 1.3.1. Cálculo de áreas

Empregaremos a función de cálculo de área dun triángulo vista na sección anterior:

$$2A(T) = (b_0 - a_0)(c_1 - a_1) - (c_0 - a_0)(b_1 - a_1)$$

Esta será unha función da clase `Triangle`.

```
def area_2(self):
    a, b, c = self.a, self.b, self.c
    return (b.x - a.x)*(c.y - a.y) - (c.x - a.x)*(b.y - a.y)

def is_degenerate(self):
    return self.area_2() == 0
```

E chamarémola en `Polygon` para calcular a área dun polígono mediante a fórmula do Teorema 1.2:

$$A(P) = A(p, v_0, v_1) + A(p, v_1, v_2) + \cdots + A(p, v_{n-2}, v_{n-1}) + A(p, v_{n-1}, v_0).$$

```
def area_2(self, point=None):
    if point is None:

        point = self.vertices[0]

    pairs = zip(self.vertices, self.vertices.rotated())

    return sum(

        Triangle(point, vertex, next_).area_2()

        for vertex, next_ in pairs

    )


def area_1_14(self):
    pairs = zip(self.vertices, self.vertices.rotated())

    return sum(

        vertex.x*next.y - vertex.y*next.x

        for vertex, next in pairs

    )


def area(self):
    return self.area_2() / 2
```

Este programa e os vindeiros empregarán funcións das listas enlazadas e das listas enlazadas circulares. No anexo tamén podemos atopar detalles destas estruturas de datos.

### 1.3.2. Intersección de segmentos

Unha observación importante é que a función do cálculo de área dun triángulo ofrécenos información sobre onde se atopa o punto $c$ no espazo respecto de $a$ e $b$, en particular, se trazamos a recta pasando por $a$ e $b$:

1. Se $A(T) = 0$ entón $a, b, c$ son colineares.
2. Se $A(T) > 0$ entón $c$ atópase á esquerda da recta se a recorremos no sentido de $\vec{ab}$.
3. Se $A(T) < 0$ entón $c$ atópase á dereita da recta se a recorremos no sentido de $\vec{ab}$.

Engadiremos á clase `Segment` as funcións `is_left`, `is_left_on`. Tamén engadiremos a función `are_collinear` ó módulo `Triangle`. Estas funcións comprobarán todas estas posibilidades.

```
def is_left(self, p):
    return area_triangle_2(self.start, self.end, p) > 0

def is_left_on(self, p):
    return area_triangle_2(self.start, self.end, p) >= 0
```

```
def are_collinear(a, b, c):
    return area_triangle_2(a, b, c) == 0
```

O noso seguinte obxectivo será crear un método de `Segment` que comprobe se dous segmentos $ab$ e $cd$ se intersecan. Para isto, en primeiro lugar crearemos unha función que comprobe as interseccións propias `intersec_properly`, é dicir, comprobará que os segmentos intersecan a non ser que o punto de intersección coincida cun dos extremos $a, b, c$ ou $d$. É abondo con empregar un razoamento similar ao previo coa fórmula da área, basta con ver que:

1. Non existe colinearidade entre ningún dos puntos.
2. Os puntos $c$ e $d$ se atopan en distintos semiplanos limitados pola recta que pasa por $a$ e $b$, e o mesmo pasa para os puntos $a$ e $b$ e a recta que pasa por $c$ e $d$.

O segundo punto é equivalente a comprobar simultaneamente as seguintes desigualdades:

$$A(a, b, c) \cdot A(a, b, d) < 0,$$
$$A(c, d, a) \cdot A(c, d, b) < 0.$$

```
def intersect_properly(segment1, segment2):
    a, b = segment1
    c, d = segment2
    return (
        not (
            are_collinear(a, b, c) or
            are_collinear(a, b, d) or
            are_collinear(c, d, a) or
            are_collinear(c, d, b)
        ) and (
            xor(segment1.is_left(c), segment1.is_left(d)) and
            xor(segment2.is_left(a), segment2.is_left(b))
        )
    )
```

Só necesitamos unha función máis antes de definir `intersect`, trátase dunha función que, sendo $a, b, c$ colineares, comprobe se $c$ se atopa entre $a$ e $b$. Referímonos a verificar a seguinte sentencia:
Se $a_0 \neq b_0$ :

$$(a_0 \leq c_0 \quad e \quad c_0 \leq b_0) \quad ou \quad (a_0 \geq c_0 \quad e \quad c_0 \geq b_0),$$

Se $a_0 = b_0$ :

$$(a_1 \leq c_1 \quad e \quad c_1 \leq b_1) \quad ou \quad (a_1 \geq c_1 \quad e \quad c_1 \geq b_1).$$

```
def is_between(self, p):
    a, b = tuple(self)
    if not are_collinear(a, b, p):
        return False
    elif a.x != b.x:
        return (a.x >= p.x and p.x >= b.x) or (a.x <= p.x and p.x <= b.x)
    else:
        return (a.y >= p.y and p.y >= b.y) or (a.y <= p.y and p.y <= b.y)

def __contains__(self, p):
    return self.is_between(p)
```

Para comprobar a intersección en xeral basta con que se verifique a intersección propia ou que algún punto esté contido no segmento que non está definido por el. Esta función operará sobre dous segmentos, logo será definida fóra de calquera clase.

```
def intersect(segment1, segment2):
    return (
```

```
        intersect_properly(segment1, segment2) or
        segment2.start in segment1 or
        segment2.end in segment1 or
        segment1.start in segment2 or
        segment1.end in segment2
    )
```

### 1.3.3. Diagonais

Para este apartado empregaremos unha caracterización das diagonais.

**Lema 1.22.** *O segmento $s = v_i v_j$ é unha diagonal do polígono $P$ se:*

1. *Para todo lado $l$ do polígono $P$ non incidente a $v_i$ ou $v_j$, $s$ e $l$ non intersecan.*
2. *$s$ está contido en $P$ nunha veciñanza de $v_i$ e $v_j$.*

O que faremos será crear unha función `_is_possible_diagonal` dun segmento que deberá ter como extremos vértices do polígono. Para verificar o punto 1 do Lema 1.22 recurrirá ás funcións de intersección anteriormente explicadas, comprobando que calquera lado non incidente a estes puntos non interseca a $ab$.

```
def _is_possible_diagonal(self, segment):
    pairs = zip(self.vertices, self.vertices.rotated())
    end_points = tuple(segment)
    for vertex, next_ in pairs:
        if (
            not Point(*vertex) in end_points and
            not Point(*next) in end_points and
            intersect(segment, Segment(vertex, next))
        ):
            return False
        else:
            return True
```

Para verificar o punto 2 do Lema 1.22 empregaremos unha nova función `_in_cone` que detectará se un segmento está no interior dun cono ou non (cando menos, localmente).

O que fará exactamente será comprobar se un vector $B$ se atopa estritamente no cono percorrido en sentido antihorario entre dous vectores $C$ e $A$. Lembremos que os lados do noso polígono están ordenados en sentido antihorario. Se $A$ é un lado, $C$ é o seguinte, e $a$ é a orixe dos vectores $A$ e $C$, temos que se conseguimos comprobar que $B = \vec{ab}$ se atopa no

interior do cono descrito e repetiremos o proceso para o vértice $b$ e os seus lados adxacentes, estaremos verificando a condición 2 do Lema 1.22 para a diagonal $ab$.

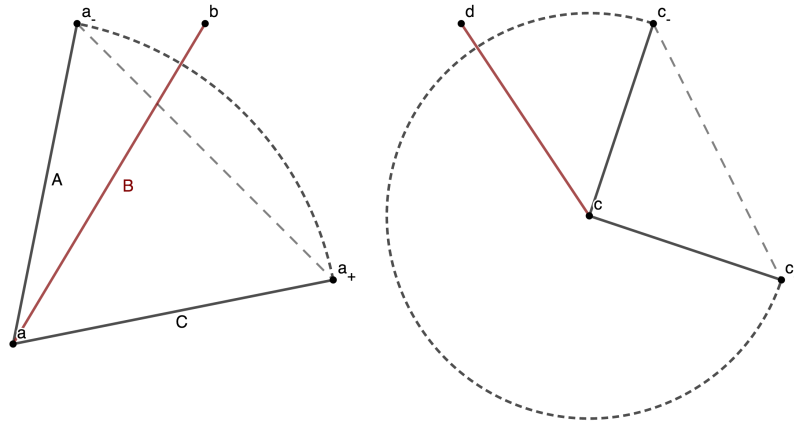


Figura 1.7: Conos percorridos en sentido antihorario entre dous vectores.

Lembremos que: $\sphericalangle(A, C) \leq \pi \Leftrightarrow [a_-$ se atopa á esquerda ou é colinear a $aa_+]$. Isto podemos comprobalo chamando a funcións descritas con anterioridade. Sabendo isto, esquematizaremos o funcionamento do noso programa:

1. $\sphericalangle(A, C) \leq \pi$ , logo $ab$ está no interior do cono se se verifican as seguintes condicións:

   *a*) $a_-$ está á esquerda de $ab$,

   *b*) $a_+$ está á dereita de $ab$.

2. $\sphericalangle(A, C) > \pi$ , logo $ab$ está no interior do cono se non se verifica unha das seguintes condicións:

   *a*) $a_+$ está á esquerda ou é colineal a $ab$,

   *b*) $a_-$ está á dereita ou é colineal a $ab$.

```
def _in_cone(self, start, end):
    segment = Segment(start.value, end.value)
    previous = start.previous.value
    next_ = start.next.value
    if Segment(start.value, next).is_left_on(previous):
        return (
            segment.is_left(previous) and
```

```
                segment.opposite().is_left(next_)
            )
        else:
            return not (
                segment.is_left_on(next_) and
                segment.opposite().is_left_on(previous)
            )
```

Seguindo os pasos anteriores podemos comprobar os puntos 1 e 2 do Lema 1.22 nunha única función `_is_diagonal` e así verificar se $ab$ é unha diagonal do polígono, sendo $a$ e $b$ dous vértices non consecutivos.

```
def _is_diagonal(self, start, end):
    return (
        self._in_cone(start, end) and
        self._in_cone(end, start) and
        self._is_possible_diagonal(Segment(start.value, end.value))
    )
```

### 1.3.4. Triangulación

Xa estamos preparados para desenvolver o código da procura dunha triangulación do polígono. Antes de comezar, cabe destacar que existen diversos métodos, poderiamos imitar o proceso que se segue na proba do Teorema 1.14: atopar unha diagonal, dividir o polígono en dous subpolígonos e recurrir de novo á función ata que o número de lados de todos os subpolígonos sexa 3. Non obstante, este método é enormemente ineficiente, en concreto é un algoritmo de orde $O(n^4)$.

Nós empregaremos o método de eliminación de orellas, que ten un custo computacional de orde $O(n^2)$, a idea clave é que eliminar unha orella dun polígono non cambia se os vértices do polígono son potenciais orellas, salvo para os dous vértices adxacentes ao vértice da orella eliminada.

Lembremos que a nosa clase `Vertex` ten o atributo `is_ear`, que por defecto estará baleiro. Asignarémoslle `True` se o vértice é unha potencial orella do polígono, é dicir, se o segmento comprendido entre os vértices previo e posterior é unha diagonal, ou `False` no caso contrario. Con todo isto, esquematizamos o algoritmo de triangulación de polígonos:

1. Actualización do atributo orella para todos os vértices do polígono.

2. Mentres o número de lados do noso polígono sexa maior que 3, faremos o seguinte:

    *a*) Atopar unha potencial orella $v_i$.

b) Enviar por saída o segmento $v_{i-1}v_{i+1}$.

c) Eliminar $v_i$ do noso polígono e restar unha unidade ao número de lados.

d) Actualizar o estado de potencial orella de $v_{i-1}, v_{i+1}$.

```
def _ear_init(self):
    trios = zip(
        self.vertices.nodes(),
        self.vertices.rotated_nodes(1),
        self.vertices.rotated_nodes(2),
    )
    for v0, v1, v2 in trios:
        v1.value.is_ear = self._is_diagonal(v0, v2)

def triangulate(self):
    new = type(self)(self.vertices)
    yield from new._triangulate()

def _triangulate(self):
    self._ear_init()
    while len(self.vertices) > 3:
        five_consecutive = zip(*(
            self.vertices.rotated_nodes(offset)
            for offset in (-2, -1, 0, 1, 2)
        ))
        for v0, v1, v2, v3, v4 in five_consecutive:
            if v2.value.is_ear:
                yield Segment(v1.value, v3.value)
                v1.value.is_ear = self._is_diagonal(v0, v3)
                v3.value.is_ear = self._is_diagonal(v1, v4)
                v2.remove()
                break
```

*Observación* 1.23. `remove()` é unha función de `CircularList` que, resumidamente, chama a unha función da lista enlazada para eliminar ese punto da lista que o contén. No anexo podemos atopar máis información sobre as estruturas de datos empregadas.

Por último, analizaremos o custo computacional do algoritmo. A inicialización do atributo orella para todos os vértices ten un custo de $O(n^2)$, xa que a nosa función de procura de diagonais ten un custo de $O(n)$ e será chamada un total de $n$ veces.

O noso bucle iterará tantas veces como diagonais existan, e temos garantido polo Teorema 1.14 que existen $n-3$ diagonais, polo tanto o custo é de $O(n)$. No peor dos casos, dentro deste bucle teremos que recorrer constantemente todo o polígono e chegar ao último

vértice para atopar unha orella, polo que teremos unha hipotética complexidade computacional de $O(n^3)$ para o noso algoritmo completo.
Unha análise máis detallada e experimental levaríanos a concluír que na gran maioría dos casos $O(n^2)$ é a complexidade computacional adecuada para o algoritmo.

### 1.3.5. Outros algoritmos de triangulación

O algoritmo que temos visto na sección anterior é o algoritmo de *Lennes* (1911) e ten unha orde de $O(n^2)$. En 1978, o algoritmo de *Garey M.R.* conseguiría acadar unha eficiencia de $O(n\log(n))$, poñendo comezo a unha etapa de 13 anos de procura dun algoritmo que poida mellorar a barreira de $n\log(n)$. O fin desta etapa veu marcado polo algoritmo de *Chazelle* (1990) que acada $O(n)$ no peor dos casos. Este último algoritmo non é sinxelo, por iso quedou aberta a búsqueda dun algoritmo máis simple, práctico e computacionalmente barato, que se acadaría en 1991 co algoritmo aleatorizado de *Seidel*, $O(n\log^* n)$.

*Notación* 1.24. $\log^* n$ é o número de veces que o logaritmo debe ser aplicado a $n$ para obter un resultado igual ou menor a 1. Por exemplo, para $n = 2^{2^{16}}$, $log^*(n) = 5$ xa que $log(log(log(log(log(n))))) = 1$.

| | | |
|---|---|---|
| 1911 | $O(n^2)$ | Lennes |
| 1978 | $O(n\log n)$ | Garey et al. |
| 1983 | $O(n\log r)$, r vértices convexos | Hartel & Mehlhorn |
| 1984 | $O(n\log s)$, s sinuosidad | Chazelle & Incerpi |
| 1988 | $O(n+nt_0)$, $t_0$ triángulos interiores | Toussaint |
| 1986 | $O(n\log\log n)$ | Tarjan & Van Wyk |
| 1989 | $O(n\log^*(n))$, aleatorizado | Clarkson, Tarjan & Van Wyk |
| 1990 | $O(n\log^*(n))$ | Kirkpatrick, Klawe & Tarjan |
| 1990 | $O(n)$ | Chazelle |
| 1991 | $O(n\log^*(n))$, aleatorizado | Seidel |

Brevemente e sen entrar en detalle, comentaremos que as chaves do algoritmo de *Garey* e de *Seidel* son as mesmas. O concepto máis importante é a monotonía dun polígono. Un polígono monótono pode triangularse en tempo lineal, e calquera polígono pódese subdividir en polígonos monótonos. Para atopar esta división en subpolígonos, imos trapezoidalizar o polígono (os algoritmos difiren na procura da trapezoidalización, o de *Garey* simplemente traza liñas horizontais en todos os vértices), e posteriormente trazaranse diagonais dentro dos trapecios nos vértices que cumplan unhas determinadas condicións. Estas diagonais serán as que nos levarán finalmente a ter o polígono dividido en subpolígonos monótonos.

# Capítulo 2

# Envolvente convexa

Unha das estruturas máis polivalentes da xeometría computacional é a envolvente convexa. É útil por si mesma e tamén para a construción doutras estruturas. Entre as súas aplicacións atópanse a solución ao problema de atopar a menor área rectangular contedora dun polígono arbitrario, a evitación de colisións para robots, e a análise e clasificación de figuras mediante "árbores de deficiencia convexa".

## 2.1. Definicións e propiedades básicas

Neste capítulo, o noso obxectivo será a elaboración e implementación de algoritmos que atopen a envolvente convexa a partir dun conxunto de puntos. Daremos as definicións dos conceptos necesarios para este capítulo.

**Definición 2.1.** Un conxunto $S$ é *convexo* se $x \in S$ e $y \in S$ implica que o segmento $xy \subseteq S$.

**Definición 2.2.** Unha *combinación convexa* de puntos $x_1, ..., x_k$ é unha suma da forma $\alpha_1 x_1 + ... + \alpha_k x_k$ con $\alpha_i \geq 0$ para todo $i = 1, ..., k$ e tal que $\alpha_1 + ... + \alpha_k = 1$.

**Definición 2.3.** A *envolvente convexa* dun conxunto de puntos $S$ é o conxunto de todas as combinacións convexas de puntos de $S$. Denotarémolo como $\text{conv}(S)$.

**Teorema 2.4** (Teorema de Caratheodory (en $\mathbb{R}^2$) [5])**.** *Sexa $S$ un conxunto de puntos de $\mathbb{R}^2$ e $x$ un punto de* $\text{conv}(S)$*. Entón $x$ pode ser escrito como combinación convexa de 3 ou menos puntos de $S$.*

*Demostración.* Sexa $x$ un punto de $\text{conv}(S)$. Podemos escribir $x$ como combinación convexa

de puntos de $S$.

$$x = \sum_{j=1}^{k} \alpha_j x_j, \quad \sum_{j=1}^{k} \lambda_j = 1, \quad x_j \in P, \quad \lambda_j \geq 0, \quad \forall j = 1, \dots, k.$$

Supoñamos $k > 3$; logo os vectores $x_2 - x_1, \dots, x_k - x_1$ son linealmente dependentes. Polo tanto, existen escalares reais $\mu_2, \dots, \mu_k$ non todos cero, tales que

$$\sum_{j=2}^{k} \mu_j (x_j - x_1) = 0.$$

Agora definimos $\mu_1$ do seguinte xeito:

$$\mu_1 = -\sum_{j=2}^{k} \mu_j.$$

Logo

$$\sum_{j=1}^{k} \mu_j x_j = -x_1 \sum_{j=2}^{k} \mu_j + \sum_{j=2}^{k} \mu_j x_j = \sum_{j=2}^{k} \mu_j (x_j - x_1) = 0,$$

$$\sum_{j=1}^{k} \mu_j = 0.$$

Ademais, non todos os $\mu_j$ son iguais a cero. Polo tanto, cando menos un $\mu_j$ satisfai $\mu_j > 0$. Logo

$$x = \sum_{j=1}^{k} \lambda_j x_j - \alpha \sum_{j=1}^{k} \mu_j x_j = \sum_{j=1}^{k} (\lambda_j - \alpha \mu_j) x_j, \quad \forall \alpha \in \mathbb{R}$$

Polo tanto podemos escoller $\alpha$ do seguinte xeito:

$$\alpha = \min_{1 \leq j \leq k} \left\{ \frac{\lambda_j}{\mu_j} : \mu_j > 0 \right\} = \frac{\lambda_i}{\mu_i}.$$

Decatémonos de que $\alpha > 0$, e para todo $j = 1, \dots, k$ se verifica

$$\lambda_j - \alpha \mu_j \geq 0.$$

En particular, $\lambda_i - \alpha \mu_i = 0$ por definición de $\alpha$. Polo tanto,

$$x = \sum_{j=1}^{k} (\lambda_j - \alpha \mu_j) x_j,$$

onde cada $\lambda_j - \alpha \mu_j$ é non negativo, a súa suma é 1, e ademais $\lambda_i - \alpha \mu_i = 0$. Noutras palabras, temos $x$ expresado como unha combinación convexa de como moito $k - 1$ puntos de $S$. Este proceso pode ser repetido ata ter $x$ expresado como combinación convexa de como moito 3 puntos de $S$. □

**Proposición 2.5.** *Caracterizacións da envolvente convexa:*

1. *A envolvente convexa dun conxunto finito de puntos $S$ é o menor polígono $P$ convexo que contén a S, é dicir, non existe ningún polígono convexo $P'$ tal que $S \subseteq P' \subset P$.*

2. *A envolvente convexa dun conxunto de puntos $S$ é a unión dos triángulos determinados por todos os posibles tríos de puntos de $S$.*

*Demostración de 1.* "$\subseteq$" Todo polígono convexo que conteña a $S$ debe conter todas as combinacións convexas de puntos de $S$; polo tanto, o conxunto de todas as combinacións está contido na intersección de todos os polígonos convexos que conteñan a $S$.
"$\supseteq$" O conxunto de todas as combinacións convexas de puntos de $S$ é un conxunto convexo que contén a $S$. Polo tanto tamén contén á intersección de todos os conxuntos convexos que conteñen a $S$, en particular, á intersección de todos os polígonos convexos que conteñen a $S$. □

*Demostración de 2.* "$\subseteq$" Sexa $x \in \text{conv}(S)$, temos que por definición $x$ pode ser expresado como combinación convexa de puntos de $S$. Polo Teorema 2.4, $x$ pode ser expresado como combinación convexa de 3 ou menos puntos de $x$. O conxunto de combinacións convexas de 3 puntos é un triángulo, logo $x$ atópase nun triángulo con vértices en $S$. Finalmente, $x$ está na unión de todos os posibles triángulos con vértices en $S$.
" $\supseteq''$ $x$ está na unión de todos os posibles triángulos con vértices en $S$; logo existe un triángulo con vértices en $S$ tal que $x$ pertence a dito triángulo. Finalmente, por estar nese triángulo, podemos escribir $x$ como combinación convexa dos seus vértices, que son puntos de $S$. □

**Definición 2.6.** Dado un conxunto de puntos $S$, diremos que un punto $x$ é *extremo* se pertence á envolvente convexa de $S$ e verifica:

$$x = \frac{a+b}{2} \quad a, b \in \text{conv}(S) \quad \Rightarrow \quad a = x = b.$$

**Proposición 2.7.** *Dado un conxunto de puntos $S$, son equivalentes:*

- *$x$ é punto extremo da envolvente convexa de $S$.*
- *$x$ pertence á envolvente convexa de $S$ e verifica:*

$$x = (1-t)a + tb, \quad a, b \in \text{conv}(S), \quad t \in (0,1) \quad \Rightarrow \quad a = x = b$$

*Demostración.* "$\Rightarrow$" Sexa

$$x = (1-t)a + tb, \quad a, b \in \text{conv}(S), \quad t \in (0,1).$$

Escollemos $x_0$ na expresión anterior con $t_0 \in (0,1)$ fixado. Se $a$ é o punto máis cerca de $x_0$ (no caso de que fose $b$ procederiamos do mesmo xeito) definimos:

$$c = a + 2\overrightarrow{ax_0} = 2x_0 - a.$$

Agora ben, con esta definición temos que $c \in ab \subset \text{conv}(S)$ (xa que a escolla de $c$ consiste en partir nun extremo do segmento e avanzar na dirección do outro extremo do segmento unha distancia menor ou igual á metade da lonxitude de $ab$). Polo tanto, $c \in \text{conv}(S)$, logo o que temos é que

$$x_0 = \frac{a+c}{2} \quad a, c \in \text{conv}(S).$$

Finalmente, por hipótese $a = x_0 = b$.
"$\Leftarrow$" Basta con tomar $t = \frac{1}{2}$. $\square$

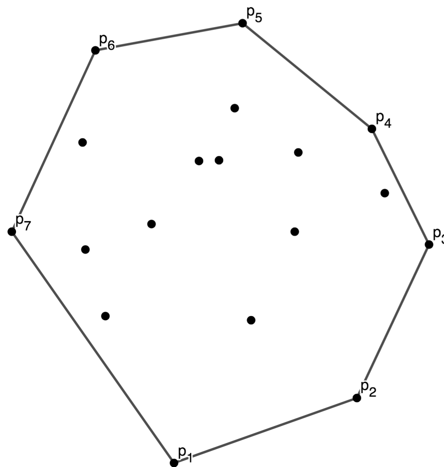


Figura 2.1: $\{p_1, \dots, p_n\}$ son puntos extremos.

*Observación* 2.8. Esta definición de punto extremo fará que os puntos contidos en segmentos da envolvente convexa non sexan considerados puntos extremos.

*Observación* 2.9. O punto coa maior altura de $S$ é extremo se é único. O mesmo sucede co punto coa menor altura, o que está máis á dereita e máis á esquerda.

**Definición 2.10.** Diremos que un segmento é un *segmento extremo* se cada punto de $S$ se atopa sobre el ou sobre un só semiplano que determina a liña sobre ese segmento.

*Observación* 2.11. Na definición anterior seranos útil considerar que percorremos o segmento nun sentido, e o semiplano do que falamos é un lado (escolleremos esquerda).

Non precisamos de moitos máis conceptos para comezar a procura de algoritmos, pero si establecer un obxectivo concreto dos mesmos. Hai esencialmente catro formas de expresar a información que procuramos:

I Todos os puntos da envolvente convexa nunha orde arbitraria.

II Os puntos extremos nunha orde arbitraria.

III Todos os puntos da envolvente convexa en orde de percorrido do borde.

IV Os puntos extremos en orde de percorrido do borde.

Algunhas aplicacións da envolvente convexa necesitarán información expresada dun xeito concreto destes catro. É concebible que en xeral é máis sinxelo ofrecer, por exemplo, os puntos nunha orde arbitraria que en orden transversal de fronteira. Veremos máis adiante que existen algoritmos que fan que o custo computacional de ofrecer esta saída ordenada de puntos non sexa maior. Nun inicio, centrarémonos en obter a saída de información II.

## 2.2. Implementación dalgúns algoritmos

Antes de comezar a presentar algoritmos, cabe destacar que botaremos man das clases `Point`, `Segment` e `Vector`. As dúa primeiras descritas no tema anterior, e todas elas se detallan no anexo. Mostramos a continuación a clase `Vector`.

```
class _Pair:
    __slots__ = ('__x', '__y')

    def __init__(self, x, y):
        self.__x = x
        self.__y = y

    def __repr__(self):
        return f'{type(self).__qualname__}({self.x!r}, {self.y!r})'

    def __eq__(self, other):
        if isinstance(other, type(self)):
            return (self.x, self.y) == (other.x, other.y)
        else:
            return NotImplemented

class Vector(_Pair):
    def __add__(self, vector):
        cls = type(self)
```

```
        if isinstance(vector, cls):
            return cls(self.x + vector.x, self.y + vector.y)
        else:
            return NotImplemented

    def __mul__(self, scalar):
        if isinstance(scalar, Real):
            return type(self)(scalar * self.x, scalar * self.y)
        else:
            return NotImplemented

    def __truediv__(self, scalar):
        return type(self)(self.x / scalar, self.y / scalar)

    def __rmul__(self, scalar):
        return self * scalar

    def __neg__(self):
        return (-1)*self

    def __sub__(self, vector):
        return self + (-1)*vector

    def norm(self):
        return sqrt(scalar_product(self, self))

    def unit_orthogonal(self):
        v = Vector(-self.y, self.x)
        return v / v.norm()
```

### 2.2.1. Puntos non extremos

Neste algoritmo empregaremos que os puntos extremos son sempre puntos de $S$, e a estratexia a seguir será atopar todos os puntos de $S$ que non sexan extremos, obtendo así a envolvente convexa formada polos puntos restantes.

**Lema 2.12.** *Dado un conxunto de puntos $S$, se $p$ é un punto extremo, entón $p \in S$.*

*Demostración.* Supoñamos que existe $p \notin S$, $p$ punto extremo. Empregando a caracterización da envolvente convexa da Proposición 2.5.2 temos que:

$$p \in \bigcup_{x,y,z \in S} \triangle(x, y, z) \quad \Rightarrow \quad \exists\, x_0, y_0, z_0 \in S : \; p \in \triangle(x_0, y_0, z_0).$$

Como $p \notin S$, sabemos que $p \neq x_0, y_0, z_0$. Entón $p$ está no interior do triángulo ou sobre un dos seus lados. Os triángulos son convexos e $\triangle(x_0, y_0, z_0) \subseteq \text{conv}(S)$, logo podemos obter $p$ como unha combinación convexa de puntos de $\text{conv}(S)$ do xeito:

$$p = (1-t)a + tb \quad a, b \in \triangle(x_0, y_0, z_0) \subseteq \text{conv}(S) \quad , \quad t \in (0,1), \quad a \neq p \neq b$$

Polo tanto, entramos nunha contradición coa definición de punto extremo para $p$. □

**Corolario 2.13.** *Un punto $x$ dun conxunto de puntos $S$ é non extremo se está contido nun triángulo cuxos vértices son puntos de $S - \{x\}$.*

Temos visto no capítulo anterior que podemos comprobar se un punto se atopa nun triángulo mediante varias chamadas á nosa función `is_left_on`. Non precisamos de nada máis para elaborar un primeiro algoritmo `non_extreme_points` na procura de puntos non extremos seguindo os pasos de O'Rourke.

```
def non_extreme_points(points):
    for p1, p2, p3, q in permutations(points, 4):
        if (not are_collinear(p1,p2,p3)) and Segment(p1,p2).is_left_on(q)
            and Segment(p2,p3).is_left_on(q) and Segment(p3,p1).is_left_on(
            q) and q!=p1 and q!=p2 and q!=p3:
             yield q
```

*Observación* 2.14. Para comprobar que un punto está dentro do triángulo empregamos tres chamadas á función `is_left_on`. Lembremos que isto só serve se os vértices do triángulo están ordenados en sentido antihorario, e este algoritmo non fai esa distinción. Como só son 3 vértices, só existen dúas posibilidades, ou os vértices están ordenados en sentido horario ou en sentido antihorario. Observamos entón que no caso do sentido horario, nunca se verificarán os 3 `is_left_on` xa que a intersección dos semiplanos na que estamos buscando puntos é o baleiro. Eses mesmos puntos serán reconsiderados en sentido antihorario noutra iteración do algoritmo.

Neste algoritmo para un conxunto $S$ de $n$ puntos estamos comprobando se cada un dos $n$ puntos está no interior de todos os triángulos posibles. Son 4 bucles aniñados. O custo computacional do algoritmo é da orde $O(n^4)$. É un algoritmo intuitivo e conceptualmente sinxelo, pero é preferible buscar un algoritmo computacionalmente menos custoso.

A Observación 2.14 suxírenos un xeito de aforrar iteracións innecesarias evitando escoller varias veces os triángulos en sentido horario e antihorario. Para iso empregaremos combinacións en lugar de permutacións, e outro xeito de comprobar se o punto se atopa no interior do triángulo, que consistirá en verificar o seguinte código.

```python
class Triangle:
    def __init__(self, a, b, c):
        self.a = Point(*a)
        self.b = Point(*b)
        self.c = Point(*c)

    def area_2(self):
        a, b, c = self.a, self.b, self.c
        return (b.x - a.x)*(c.y - a.y) - (c.x - a.x)*(b.y - a.y)

    def is_inside(self, point):
        clockwise = self.area_2()
        if clockwise == 0:
            return False
        else:
            return (
                clockwise * Triangle(self.a, self.b, point).area_2() >= 0
                    and
                clockwise * Triangle(self.b, self.c, point).area_2() >= 0
                    and
                clockwise * Triangle(self.c, self.a, point).area_2() >= 0
            )

    def __contains__(self, point):
        return self.is_inside(point)
```

```python
def nonextreme_points(points):
    for p1, p2, p3 in combinations(points, 3):
        for q in points:
            if q not in (p1, p2, p3) and q in Triangle(p1, p2, p3):
                yield q
```

Como vemos, agora os signos das nosas operacións cancélanse, logo a condición é independente do sentido no que se percorran os vértices do triángulo.

Deste xeito, para un conxunto $S$ de $n$ puntos estamos comprobando se cada un dos $n$ puntos se atopa no interior dos $\binom{n}{3}$ posibles triángulos, logo temos $n\frac{n!}{3!(n-3)!} = \frac{n^2(n-1)(n-2)}{6}$ iteracións. Polo tanto e pese a aforrar iteracións, a complexidade é de $O(n^4)$.

### 2.2.2. Segmentos Extremos

Será algo menos custoso identificar segmentos extremos. Como especificamos, un segmento orientado é extremo se todos os puntos do seguinte conxunto quedan á súa esquerda.

Redactado negativamente, un orientado é non extremo se hai algún punto do noso conxunto que está estritamente á dereita del. Isto é o que empregaremos no seguinte algoritmo facendo uso de novo da nosa función `is_left_on`. A súa saída será do tipo I, xa que se hai un punto $z$ contido nun segmento extremo $xy$, os segmentos $xz$ e $zy$ tamén serán segmentos extremos.

```
def extreme_edges(points):
    for p1, p2 in combinations(points, 2):
        edge = Segment(p1, p2)
        all_left = all(
            edge.is_left_on(q) for q in points if q != p1 and q != p2
        )
        opposite = edge.opposite()
        all_right = all(
            opposite.is_left_on(q) for q in points if q != p1 and q != p2
        )
        if all_left or all_right:
            yield edge
```

Este algoritmo execútase en $O(n^3)$, xa que hai 3 bucles aniñados. Para cada $n^2$ pares de puntos, o test avalía $n$ puntos.

*Observación* 2.15. Este algoritmo sen modificacións adicionais nos dará a saída de información I. Isto débese a que o criterio emprega o "$\geq$" ao comprobar a posición dos puntos respecto de cada segmento, logo inclúense puntos sobre o bordo da envolvente convexa que non son extremos.

### 2.2.3. Embalaxe de agasallo

Nesta sección comezamos a ver algoritmos menos intuitivos pero realistas e computacionalmente menos custosos. Este algoritmo foi suxerido por *Chand & Kapur* [6] (1970), e a súa idea é empregar un segmento extremo para atopar o seguinte. Isto é posible porque sabemos que os segmentos extremos forman un polígono convexo (empregando a caracterización de envolvente convexa da Proposición 2.5.1). En cada iteración exploraranse $n$ candidatos, e como máximo o número de segmentos extremos será $n$, polo tanto a complexidade redúcese a $O(n^2)$. Comezaremos a explicar o algoritmo supoñendo que non hai 3 puntos colineares en $S$, logo as saídas III e IV serán a mesma. Supoñamos que o algoritmo atopou un segmento extremo con punto extremo $x$. Sabemos que debe haber outro segmento extremo $e$ compartindo este punto extremo $x$. A chave é que, a liña $L$ pasando por $e$ ten a propiedade de formar o menor ángulo antihorario respecto á liña que pasa polo segmento anterior. Polo tanto non precisamos testar se todos os puntos se atopan á

esquerda do segmento, basta con calcular o ángulo para cada posible punto e seleccionar o máis pequeno: ese punto será extremo. Agora ben, para inicializar o algoritmo empregaremos o punto coa menor coordenada $y$ e a liña inicial para calcular o ángulo será horizontal pasando por ese punto. Observemos agora que podemos eliminar a suposición de que os puntos non sexan colineares e o algoritmo funcionará sen problema, o que sucede é que pode que algúns puntos que se atopen no interior de segmentos extremos se inclúan na saída e outros non, pero estarán en orde de percorrido do bordo.

En canto ao cálculo do ángulo, empregaremos a librería `math` e un cálculo sinxelo mediante a fórmula:

$$\sphericalangle(v, w) = \arccos\left(\frac{v \cdot w}{|v||w|}\right).$$

Onde $v = b - a$ e $w = c - a$ e consideramos $a = (a_1, a_2)$, $b = (b_1, b_2)$ e $c = (c_1, c_2)$.

$$\sphericalangle(v, w) = \arccos\left(\frac{(b_1 - a_1, b_2 - a_2) \cdot (c_1 - a_1, c_2 - a_2)}{\sqrt{(b_1 - a_1)^2 + (b_2 - a_2)^2}\sqrt{(c_1 - a_1)^2 + (c_2 - a_2)^2}}\right).$$

A implementación á computación será a seguinte, onde os argumentos $u, v$ son da clase `Vector` e teñen como atributo as súas compoñentes.

```
def scalar_product(v, w):
    return v.x*w.x + v.y*w.y
```

```
def angle(v, w):
    return acos(scalar_product(v, w) / (v.norm()*w.norm()))
```

Sexa $e = st$ o último segmento extremo atopado polo algoritmo e $t$ o último punto extremo. Para calcular o ángulo que nos interesa empregaremos a función `angle` descrita na sección anterior aplicado a $\vec{st}$ e $\vec{tp}$ para todo $p$ de $S$. Xa temos todo para presentar o noso algoritmo.

```
def gift_wrapping(points):
    first = last = Point(*min(points, key=lambda point: (point.y, point.x))
        )
    previous = last - Vector(1, 0)
    while True:
        yield last
        _, q = min(
            (angle(last-previous, p-last), tuple(p))
            for p in points
            if p != last and p != previous
        )
        previous = last
        last = Point(*q)
```

```
        if last == first:
            break
```

*Observación* 2.16. O algoritmo garda en cada iteración o valor do último punto extremo e non calcula o seu ángulo na seguinte iteración, isto aforra algunha iteración e evita problemas numéricos.

*Observación* 2.17. Neste algoritmo o criterio para escoller o seguinte punto extremo se o ángulo é o mesmo, é o que teña a menor coordenada $x$. Isto fará que a saída poida conter puntos que non son extremos, pero non significa que conteña todos os puntos que están sobre o bordo da envolvente convexa. Para unha saída de información máis específica necesitaríamos modificar ditas condicións.

O algoritmo conclúe en $n$ iteracións no peor dos casos e en cada unha calcula e compara $n$ ángulos. Logo hai 2 bucles aniñados e o custo computacional é de $O(n^2)$.

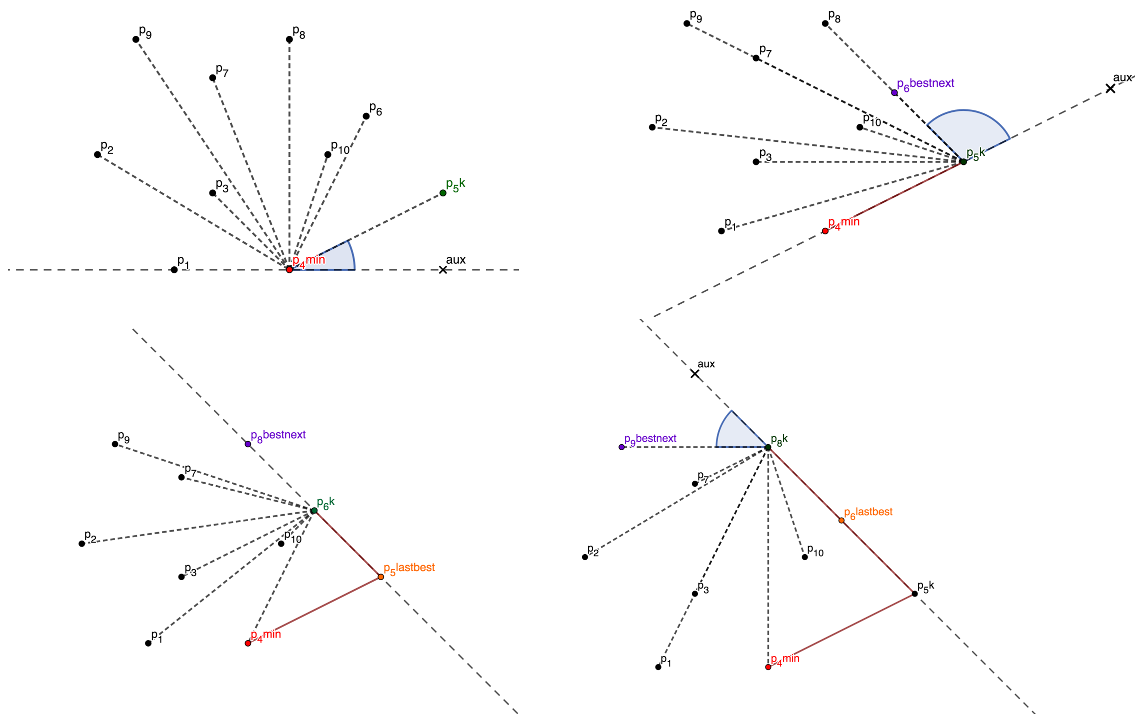


Figura 2.2: Inicialización e tres iteracións do algoritmo embalaxe de agasallo sobre un exemplo.

### 2.2.4. QuickHull

Continuamos o noso catálogo de algoritmos cunha idea que foi suxerida independentemente por varios investigadores a finais da década dos 70. *Preparata & Shamos* [7] (1985) é o algoritmo que estudaremos.

Para a maioría dos conxuntos de puntos é sinxelo descartar puntos do interior da envolvente e concentrarnos nos máis achegados á fronteira. A inicialización do algoritmo QuickHull consiste en atopar dous puntos extremos distintos, o que se atope máis baixo entre os que estean máis á dereita e o o que se atope máis alto entre os que estean máis á esquerda, $x$ e $y$ respectivamente. Con esta escolla, temos asegurado que serán puntos extremos distintos. O segmento $xy$ dividirá a envolvente convexa no que chamaremos a "envolvente superior" (sobre $xy$) e a "envolvente inferior" (baixo $xy$).

Por outro lado, definiremos unha función *QuickHull(a, b, S)* que atopará un punto extremo $c$ á dereita do segmento $ab$ e descartará todos os puntos contidos en $\triangle abc$. Esta función operará recursivamente nos segmentos $ac$ e $cb$. A chave é que en *QuickHull(a, b, S)* $a$ e $b$ serán puntos extremos e $S$ será o conxunto de todos os puntos estritamente á dereita de $ab$, que pode ser baleiro. Atopar o punto extremo $c \in S$ consistirá en seleccionar o punto de $S$ coa maior distancia ao segmento $ab$.
Tendo claro isto, tras a nosa inicialización bastará con facer as chamadas *QuickHull(x, y, $S_1$)* e *QuickHull(y, x, $S_2$)*, onde $S_1$ e $S_2$ serán respectivamente as envolventes superior e inferior.

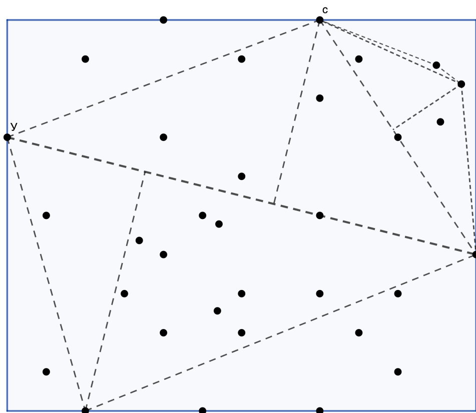


Figura 2.3: Esquematización do algoritmo QuickHull sobre un exemplo.

*Observación* 2.18. Faremos un inciso na selección do punto $c$. Se seguimos a esquematización do algoritmo de O'Rourke debemos achar o punto $c$ coa maior distancia ao segmento

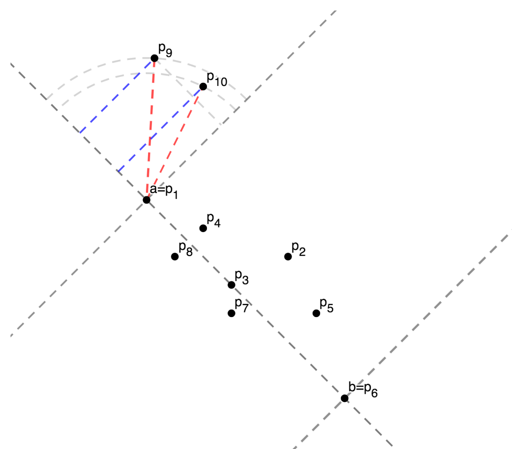


Figura 2.4: En vermello as distancias consideradas polo algoritmo de O'Rourke, en azul as distancias consideradas polo algoritmo de Preparata & Shamos.

$ab$, procederemos do seguinte xeito. Se trazamos dúas liñas ortogonais a $ab$ pasando por $a$ e $b$ respectivamente, dividiremos o cálculo nas seguintes posibilidades:

1. $c$ é coincidente con $a$ ou $b$, logo a distancia será 0 e teremos rematado.

2. $c$ está estritamente entre as dúas liñas, logo a distancia será calculada ortogonalmente sobre $ab$.

$$d(ab, c) = \frac{|\vec{ab} \times \vec{ca}|}{|\vec{ab}|}.$$

3. Nas posicións restantes, calcularemos a distancia de $c$ ao extremo do segmento que quede no mesmo semiplano ca $c$.

$$d(ab, c) = |\vec{ac}| \quad d(ab, c) = |\vec{bc}|.$$

Mostraremos a función `ort_distance` onde empregaremos os ángulos $\sphericalangle(b, a, c)$ e $\sphericalangle(a, b, c)$ como condición para a división de casos anterior, pero non empregaremos esta versión de QuickHull.

Agora ben, se seguimos os pasos de Preparata e Shamos teremos como condición para atopar $c$ que dito punto maximice a área do triángulo $\triangle(a, b, c)$. Isto é equivalente a atopar $c$ coa maior distancia ortogonal sobre a recta pasando polo segmento $ab$. Ilustramos na Figura 2.4 un exemplo no que ambos algoritmos escollen puntos $c$ distintos na primeira iteración. Decatémonos entón de que os algoritmos que propoñen O'Rourke e Preparata & Shamos son distintos, e poden rematar nun número distinto de iteracións dependendo do noso conxunto de puntos $S$.

```
def _areas(segment, points):
    a, b = segment
    return [
        (d, p) for p in points
        if p != b and (d := -Triangle(a, b, p).area_2()) > 0
    ]

def _quick_hull(segment, distance_points):
    if distance_points:
        _, farthest = max(distance_points, key=lambda pair: pair[0])
        points = [p for _, p in distance_points]
        segment1 = Segment(segment.start, farthest)
        segment2 = Segment(farthest, segment.end)
        yield from _quick_hull(segment1, _areas(segment1, points))
        yield farthest
        yield from _quick_hull(segment2, _areas(segment2, points))
```

```
def quick_hull(points):
    right_lower = Point(*max(points, key=lambda point: (point.x, -point.y))
        )
    left_upper = Point(*min(points, key=lambda point: (point.x, -point.y)))
    yield right_lower
    segment = Segment(right_lower, left_upper)
    yield from _quick_hull(segment, _areas(segment, points))
    yield left_upper
    opposite = segment.opposite()
    yield from _quick_hull(opposite, _areas(opposite, points))
```

Analizamos agora a complexidade computacional do algoritmo. Atopar os valores extremos $x$ e $y$, e particionar $S$ en $S_1$ e $S_2$ ten un custo de $O(n)$. Para a función recursiva, supoñamos que $|S| = n$. Logo leva $n$ pasos determinar o punto extremo $c$, pero o custo das chamadas recursivas depende dos tamaños de $A$ e $B$. Sexa $|A| = \alpha$ e $|B| = \beta$ tales que $\alpha + \beta \leq n - 1 = O(n)$ (a suma é $n-1$ porque $c$ non está incluído en $A$ nin en $B$).

Sexa $T(n)$ a complexidade ao chamar a QuickHull($a$, $b$, $S$) con $|S| = n$. Podemos expresalo como $T(n) = O(n) + T(\alpha) + T(\beta)$. Considerando o mellor caso posible, nunha división co mesmo número de puntos nun lado e noutro da recta $xy$: $\alpha = \beta = \frac{n}{2}$, temos que $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$. Esta é unha relación recurrente moi familiar, cuxa solución é $T(n) = O(n \log n)$.

Agora ben, no peor caso $\alpha = 0$ e $\beta = n - 1$, teríamos a relación: $T(n) = T(n-1) + cn$. Temos entón $T(n) = O(n^2)$. Pese a que en xeral o algoritmo QuickHull é frecuentemente rápido, segue tendo unha complexidade computacional cuadrática no peor caso. Na seguinte

sección culminaremos cun algoritmo cuxo peor caso é de $O(n \log n)$.

## 2.3. Algoritmo de Graham

O mérito ao descubrimento do primeiro algoritmo en atopar a envolvente convexa dun conxunto de puntos en $O(n \log n)$ é para *Graham* en 1972 como resposta á necesidade de *Bell Laboratories* de atopar a envolvente convexa a un conxunto de preto de 10000 puntos.

A idea básica do algoritmo é sinxela, e ilustrarémola cun exemplo. Supoñamos que tomamos un punto $x$ no interior do conxunto e ordenamos todos os demais puntos polo ángulo que forman coa horizontal pasando por $x$ en sentido antihorario. Chamarémolos por orden $a, b, \ldots, j$. Supoñamos ademais que temos os dous primeiros puntos da envolvente convexa no noso conxunto de puntos da envolvente $S = (b, a)$ (eliminaremos esta suposición con posterioridade). Por convención, enumeraremos os puntos de arriba a abaixo e de esquerda a dereita. Agora ben, o algoritmo comprobará se $c$ é un xiro á esquerda respecto da recta que pasa por $ab$, e nese caso actualizamos $S = (b, a, c)$. O seguinte punto será $d$, que como vemos, é un xiro á dereita respecto da recta que pasa por $bc$, logo eliminamos $c$ de $S$. Vemos agora que $d$ é un xiro á esquerda respecto da recta que pasa por $ab$, logo $S = (d, b, a)$. Continuamos con $e$ e $f$ que se engaden ao conxunto $S$ para posteriormente ser

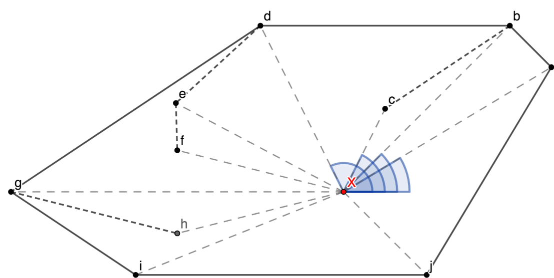


Figura 2.5: Esquematización do algoritmo sobre o exemplo descrito.

eliminados a causa de $g$, neste momento $S = (g, d, b, a)$. E así sucesivamente. Observemos que se $a$ é da envolvente convexa, esta pecharase soa, resultando $S = (j, i, g, d, b, a)$. É aparente que o bucle itérase en $O(n)$. Se contamos as iteracións que engaden puntos a $S$ e o máximo número de iteracións que eliminan puntos de $S$ o bucle iterarase como moito $2n$ veces. Polo tanto o algoritmo executarase en tempo linear despois do paso de ordenación, que ten unha complexidade de $O(n \log n)$.

### 2.3.1. Inicialización

Un bucle tan sinxelo como este pode ser complicado de inicializar e parar. Sobre todo será complexo parar o bucle se $a$ non está na envolvente convexa. Tamén será difícil de inicializar se $b$ non está na envolvente, xa que na segunda iteración $S = (a)$. Observemos que basta con escoller $a$ e $b$ puntos da envolvente convexa para aforrarnos todos os posibles problemas de inicialización e parada.

Tiñamos asumido que $x$ estaba no interior da envolvente convexa. Eliminaremos esta suposición e empregaremos como punto inicial o punto que estea máis á dereita entre os que están máis abaixo, que claramente se atopa na envolvente convexa. Agora reordenamos os puntos mediante o seu ángulo en sentido antihorario respecto deste primeiro punto e empregaremos como seguinte punto de $S$ o punto co menor destes ángulos, que tamén se atopará na envolvente convexa. Temos entón $S = (p_0, p_1)$ contendo dous puntos da envolvente convexa, o que evitará problemas de inicialización e parada do bucle.

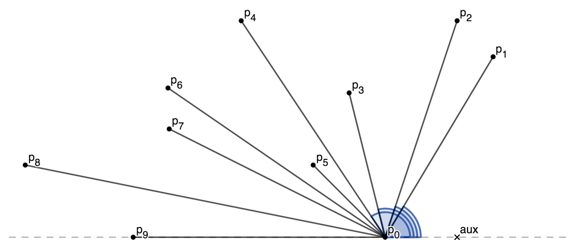


Figura 2.6: Inicialización do algoritmo sobre o exemplo anterior.

### 2.3.2. Colinearidades

Agora interesaranos eliminar a suposición de que non hai tres puntos colineares no noso conxunto.

En primeiro lugar, para as colinearidades de fronteira bastará con esixir como condición para engadir un punto a $S$ en cada iteración do algoritmo que os puntos $(p_{t-1}, p_t, p_i)$ formen un xiro estrito á esquerda. Logo se os tres puntos son colineares, $p_t$ será eliminado da envolvente convexa e o algoritmo prosegue sen problemas.

En segundo lugar interesaranos optimizar o programa eliminando puntos colineares, de xeito que non existan dous puntos co mesmo ángulo respecto de $p_0$ e a horizontal.

Bastará entón con comprobar se dous puntos $a$ e $b$ teñen o mesmo ángulo, e nese caso, se $|a - p_0| < |b - p_0|$. No caso de que as condicións se verifiquen, podemos eliminar $a$ do conxunto de puntos co que traballaremos.

### 2.3.3. Implementación

Esquematizaremos o algoritmo recopilando o xeito de proceder descrito ata agora:

1. Atopar o punto inicial (abaixo á dereita) $p_0$.
2. Ordenar todos os demais puntos empregando o ángulo respecto de $p_0$ e a horizontal.
3. Eliminar os puntos que poidamos buscando colinearidades.
4. Definir $S = (p_0.p_1)$, $previous = p_0$, $last = p_1$, $current = p_2$.
5. Mentres a lista non se acabe:

   a) Se *current* está estritamente á esquerda de $p_{t-1}p_t$

      Engadimos *current* a $S$.

      Actualizamos os valores de *previous*, *last* e *current*.

   b) Noutro caso

      Eliminamos o anterior punto de $S$.

      Actualizar os valores de *previous* e *last*.

Un último apunte antes de comezar a desenvolver o código do algoritmo é que precisaremos dunha nova clase `_GrahamPoint` que consistirá en puntos cun atributo `skip` que servirá para ignorar os puntos dos que poidamos prescindir (por colinearidade ou motivos vistos previamente) á hora de calcular os ángulos de Graham. Este atributo terá por defecto o valor `False`.

```
class _GrahamPoint(Point):
    def __init__(self, x, y):
        super().__init__(x, y)
        self.skip = False

def _remove_and_advance(node):
    aux = node
    node = node.next
    aux.remove()
    return node

def angle_closer(base):
```

```
    def compare(p, q):
        area = -Triangle(base, p, q).area_2()
        if area != 0:
            return area
        else:
            x = abs(p.x - base.x) - abs(q.x - base.x)
            y = abs(p.y - base.y) - abs(q.y - base.y)
            if x < 0 or y < 0:
                p.skip = True
                return -1
            elif x > 0 or y > 0:
                q.skip = True
                return 1
            else:
                if not p.skip and not q.skip:
                    q.skip = True
                return 0
    return compare
```

```
def graham(points):
    lower_right = min(points, key=lambda point: (point.y, -point.x))
    points = sorted(
        [_GrahamPoint(*p) for p in points if p != lower_right],
        key=cmp_to_key(angle_closer(lower_right))
    )
    hull = LinkedList(chain([lower_right], points))
    previous = hull.first
    last = previous.next
    while last.value.skip:
        last = _remove_and_advance(last)
    last.value = Point(*last.value)
    current = last.next
    while current is not None:
        if not current.value.skip:
            if Segment(previous.value, last.value).is_left(current.value):
                current.value = Point(*current.value)
                previous, last, current = last, current, current.next
            else:
                last.remove()
                last, previous = previous, previous.previous
        else:
            current = _remove_and_advance(current)
    return hull
```

### 2.3.4. Cota inferior da complexidade

Tendo presentado un algoritmo cuxa complexidade é de $O(n \log n)$ para a construción da envolvente convexa a seguinte pregunta é: podémolo facer mellor? Parece factible que atopar un algoritmo de $O(n)$ sexa posible. Mostraremos a continuación que non o é, e que $n \log n$ é cota inferior da complexidade do algoritmo. Para probar isto, empregaremos que a cota inferior da complexidade do algoritmo consistente en ordenar $n$ elementos é $n \log n$ [11].

**Proposición 2.19.** *$n \log n$ é cota inferior da complexidade computacional do algoritmo de Graham.*

*Demostración.* Supoñamos que $A$ é un problema do que coñecemos a cota inferior, e que $A$ se pode reducir ao problema $B$, é dicir, que o algoritmo para resolver $B$ pode resolver o problema $A$. Estamos entón, por hipótese, ofrecendo unha cota inferior para a complexidade do algoritmo que resolve $B$, xa que no caso de que dito algoritmo tivese unha complexidade computacional menor cá cota inferior dada, estariamos violando esta cota para o problema $A$.

Seguindo esta estratexia, empregaremos o noso algoritmo que calcula a envolvente convexa para ordenar unha lista de números $(x_1, \ldots, x_n)$ tales que $x_i \geq 0$para cada $i = 1, \ldots, n$. Supoñamos que o noso algoritmo da envolvente convexa de $n$ puntos ten unha complexidade de $T(n)$. O noso obxectivo é empregar dito algoritmo para ordenar en tempo $T(n) + O(n)$, onde a $O(n)$ adicional representa o tempo que se emprega para convertir a solución dun problema nunha solución do outro. Creamos un conxunto bidimensional de

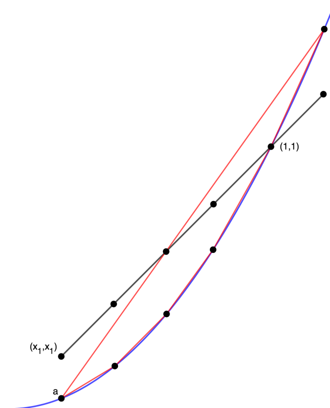


Figura 2.7: Empregando a envolvente convexa para ordenar.

puntos $(x_i, x_i^2)$ formando unha parábola. Empregamos o algoritmo da envolvente convexa para construíla a partir destes puntos (Figura 2.7, en vermello a envolvente convexa).

Claramente todos os puntos están na envolvente. O algoritmo atopará o punto coa menor coordenada $y$ en $O(n)$, e este coincidirá co máis pequeno dos nosos $x_i$ por ser $x_i \geq 0$ $i = 1, \ldots, n$ e elevar ó cadrado unha función crecente e bixectiva nese conxunto. Polo tanto, a primeira coordenada dos vértices da envolvente convexa que seguen serán os $x_i$ ordenados, e teremos resolto o problema. Finalmente temos empregado o algoritmo que atopa unha envolvente convexa para resolver o problema de ordenar un conxunto finito de puntos, logo a cota inferior do algoritmo para atopar a envolvente convexa será $n \log n$. $\square$

# Capítulo 3

# Diagramas de Voronoi

Neste capítulo estudaremos os diagramas de Voronoi, unha estrutura xeométrica cunha gran importancia e moi relacionada coa envolvente convexa. En certo modo, os diagramas de Voronoi conteñen a información que podemos precisar sobre a proximidade nun conxunto de puntos. Este concepto foi discutido en 1850 por Dirichlet e en 1908 nun artigo de Voronoi. Entre as aplicacións destes diagramas atópanse a localización de comercios, plans de ruta para robots, cristalografía.

## 3.1. Definicións e propiedades básicas

Sexa $P = \{p_1, p_2, \dots, p_n\}$ un conxunto de puntos no plano euclídeo. Neste capítulo chamaremos a cada un destes puntos un *sitio*.

**Definición 3.1.** Sexa $P$ un conxunto de sitios. Particionamos o plano mediante a asignación do sitio máis cercano para cada punto do plano. Cada conxunto desta partición recibe o nome de *rexión de Voronoi* e denótase $V(p_i)$:

$$V(p_i) = \{x \in \mathbb{R}^2 \colon |p_i - x| \leq |p_j - x| \quad \forall j \neq i\}.$$

*Observación* 3.2. As rexións de Voronoi son conxuntos pechados. Algúns puntos non pertencen a nunha única rexión de Voronoi.

**Definición 3.3.** O coxunto de todos os puntos que pertencen a varias rexións de Voronoi é o que se coñece como o *diagrama de Voronoi*, e denotarase como $\mathcal{V}(P)$.

**Exemplo 3.4.** Sexa $P = \{p_1, p_2\}$ con $p_1 \neq p_2$. Definimos $B(p_1, p_2) = B_{12}$ a mediatriz do segmento $p_1p_2$. Logo todo punto de $B_{12}$ é equidistante a $p_1$ e $p_2$ e $B_{12} = \mathcal{V}(P)$.

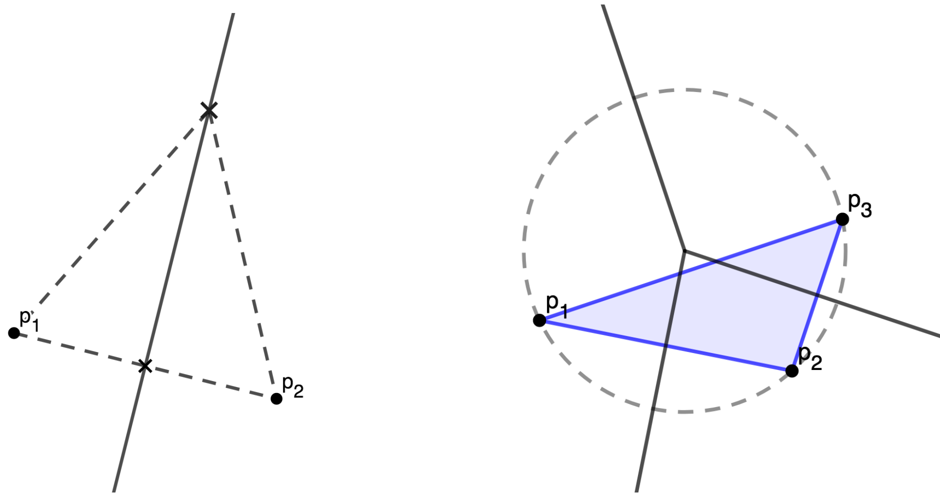


Figura 3.1: Esquema dos exemplos descritos.

**Exemplo 3.5.** Sexa $P = \{p_1, p_2, p_3\}$, temos agora tres mediatrices $B_{12}, B_{23}, B_{31}$. Agora estas mediatrices teñen un circuncentro, o punto onde a distancia aos 3 sitios é a mesma. O diagrama de Voronoi neste caso é o que se ilustra na Figura 3.1 en cor negra.

### 3.1.1. Semiplanos

Veremos a continuación o caso xeral $|P| = n$, no que as mediatrices $B_{ij}$ xogarán un papel clave.

*Notación* 3.6. Denotaremos como $H(p_i, p_j)$ ó semiplano pechado con fronteira $B_{ij}$ contendo a $p_i$.

**Proposición 3.7.** *Podemos expresar $V(p_i)$ mediante a ecuación:*

$$V(p_i) = \bigcap_{j \neq i} H(p_i, p_j). \tag{3.1}$$

*Demostración.* $H(p_i, p_j)$ pode ser visto como todos os puntos que están máis preto de $p_i$ que de $p_j$. Lembremos que $V(p_i)$ é o conxunto de todos os puntos que son máis cercanos a $p_i$ que a calquera outro punto. □

*Observación* 3.8. As rexións de Voronoi son convexas, xa que se poden obter como intersección de semiplanos, e os semiplanos son convexos.

**Definición 3.9.** Os vértices das rexións de Voronoi chamanse *vértices de Voronoi* e os segmentos que os unen son os *segmentos de Voronoi.*

**Definición 3.10.** Diremos que un vértice de Voronoi é de *grao* $n$ se é equidistante a $n$ sitios.

**Definición 3.11.** Un *grafo plano* é un grafo que pode ser debuxado no plano sen que ningún arco se cruce.

**Exemplo 3.12.** Sexa $P = \{p_1, p_2, p_3, p_4\}$ temos diversas posibilidades, ilustraremos dous exemplos concretos.

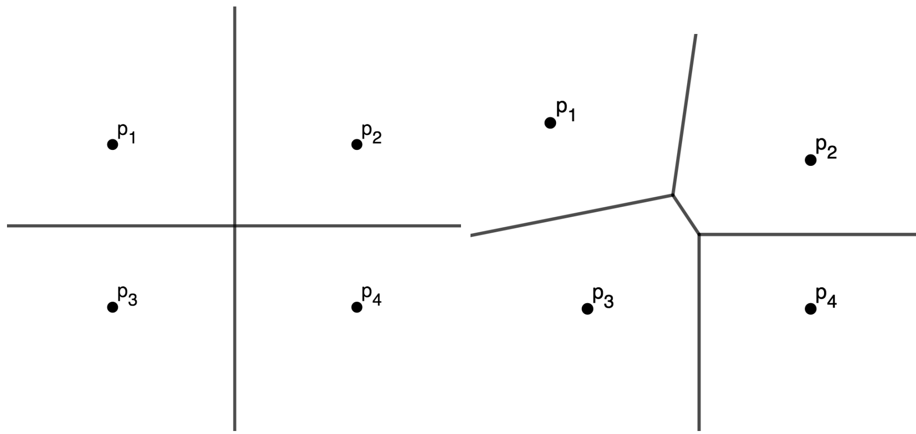


Figura 3.2: Esquema dos exemplos descritos.

No primeiro temos catros puntos que forman as esquinas dun rectángulo. Podemos ver que o único vértice de Voronoi é de grao 4.

No segundo exemplo temos os mesmos sitios salvo $p_1$ que foi desplazado. Isto provoca a aparición dun novo vértice; agora os dous vértices de Voronoi son de grao 3.

**Definición 3.13.** Diremos que un vértice de Voronoi é *dexenerado* se é de grao 4 ou superior.

**Exemplo 3.14.** Ilustraremos un diagrama de Voronoi un pouco máis grande.

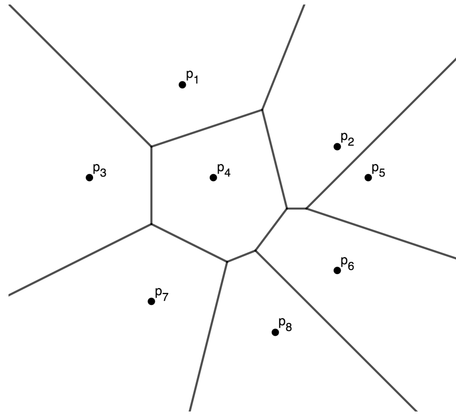


Figura 3.3: Diagrama de Voronoi con 8 sitios.

### 3.1.2. Tamaño do diagrama

Nesta sección falaremos do tamaño dun diagrama de Voronoi de $n$ sitios.

**Teorema 3.15** (Fórmula de Euler [9]). *Para un grafo plano de $V$ vértices, $L$ lados e $C$ caras, verifícase:*

$$V - L + C = 2.$$

**Corolario 3.16.** *Para unha triangulación dun grafo plano de $V$ vértices, $L$ lados e $C$ caras, verifícanse:*

$$L \leq 3V - 6 \quad C \leq 2V - 4,$$

*Demostración.* Cada lado do grafo sempre determina dúas caras. Polo tanto, a suma do número de lados que delimitan cada cara é igual ao doble do número de lados. Ademais, cada cara está delimitada por como moito 3 lados, entón $3C \leq 2L$. Agora empregando a fórmula de Euler, temos

$$V - L + C = 2 \Rightarrow V - L + \frac{2}{3}L \leq 2 \Rightarrow L \leq 3V - 6.$$

E volvendo a substituír, esta vez deixando despexado o número de caras:

$$C \leq \frac{2}{3}L \leq 2V - 4,$$

así queda probado o resultado. □

Agora ben, se falamos dos diagramas de Voronoi hai exactamente $n$ rexións de Voronoi para $n$ sitios. Sería concebible que o número de elementos (vértices e segmentos de Voronoi) do diagrama sexa cuadrático, pero veremos que isto non é así.

*Observación* 3.17. Podemos construír o dual $G$ dun diagrama de Voronoi $\mathcal{V}(P)$ como segue: cada nodo de $G$ é un sitio de $\mathcal{V}(P)$, e dous nodos están enlazados se as súas rexións de Voronoi comparten un segmento de Voronoi.

**Exemplo 3.18.** Ilustramos o dual do diagrama de Voronoi do exemplo anterior.

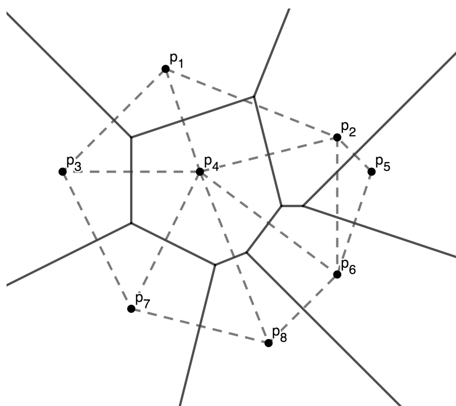


Figura 3.4: Dual superposto dun diagrama de Voronoi con 8 sitios.

## 3.2. Triangulación de Delaunay

En 1934 Delaunay probou que se debuxamos o grafo do dual dun diagrama de Voronoi $\mathcal{V}(P)$ no que non hai 4 sitios cocirculares, obtemos unha triangulación do conxunto de sitios $P$. Isto será o que chamaremos a *Triangulación de Delaunay*, e denotarémola como $\mathcal{D}(P)$. Temos ilustrado un exemplo na Figura 3.4.

**Proposición 3.19** (Propiedades da Triangulación de Delaunay [12])**.** *Verifícanse:*

*Da) $\mathcal{D}(P)$ é o dual de $\mathcal{V}(P)$.*

*Db) Hai unha correspondencia entre cada triángulo de $\mathcal{D}(P)$ e cada vértice de Voronoi de $\mathcal{V}(P)$.*

*Dc) Hai unha correspondencia entre cada segmento de $\mathcal{D}(P)$ e casa segmento de Voronoi de $\mathcal{V}(P)$.*

*Dd) Hai unha correspondencia entre cada nodo de $\mathcal{D}(P)$ e cada rexión de $\mathcal{V}(P)$.*

*De) A fronteira de $\mathcal{D}(P)$ é a envolvente convexa de $P$.*

*Df) O interior de cada triángulo de $\mathcal{D}(P)$ non contén ningún sitio.*

**Proposición 3.20** (Propiedades dos diagramas de Voronoi [12])**.** *Verifícanse:*

*Va) Cada rexión de Voronoi* $V(p_i)$ *é convexa.*

*Vb)* $V(p_i)$ *é non limitado se e só se* $p_i$ *está na envolvente convexa de* $P$.

*Vc) Se* $v$ *é un vértice de Voronoi correspondente á intersección de* $V(p_i), V(p_j)$ *e* $V(p_k)$ *distintos, entón* $v$ *é o centro do círculo* $C(v)$ *determinado por* $p_i, p_j$ *e* $p_k$. *(esta propiedade pode xeneralizarse para vértices de Voronoi de calquera grado)*

*Vd) O interior de* $C(v)$ *non contén sitios.*

*Ve) Se* $p_j$ *é o sitio máis preto de* $p_i$*, entón* $(p_i, p_j)$ *é un segmento de* $\mathcal{D}(P)$.

Queda aínda unha propiedade por enunciar. É a menos intuitiva, pero é unha importante caracterización dos segmentos de $\mathcal{D}(P)$, e empregarase en resultados posteriores, probarémola formalmente no seguinte teorema:

**Teorema 3.21.** $ab \in \mathcal{D}(P)$ *se e só se existe unha circunferencia pasando polos sitios* $a$ *e* $b$ *que non conteña ningún outro sitio.*

*Demostración.* "$\Rightarrow$" Se $ab$ é un segmento de Delaunay, $V(a)$ e $V(b)$ comparten un segmento de lonxitude positiva $e \in \mathcal{V}(P)$. Consideramos un círculo $C(x)$ con centro $x$ no interior de $e$, cun radio con distancia $d(x, a)$ ou $d(x, b)$. Supoñamos que existe un sitio $c$ no interior do círculo. Entón, $x$ estaría en $V(c)$, pero non é posible xa que $x$ só pode estar en $V(a)$ e $V(b)$ por ser un punto do interior de $e$.

"$\Leftarrow$" Supoñamos que existe un círculo baleiro $C(x)$ pasando por $a$ e $b$ con centro en $x$. Queremos probar que $ab \in \mathcal{D}(P)$. Como $x$ é equidistante a $a$ e $b$, está nas rexións de Voronoi $V(a)$ e $V(b)$ sempre e cando non haxa outro punto que interfira entre estes situándose a unha distancia menor de $x$. Pero sabemos que ningún punto pode interferir porque o círculo $C(x)$ é baleiro. Por isto, $x \in V(a) \cap V(b)$, e como non hai sitios na fronteira e $C(x)$ distintos de $a$ e $b$, existe un $\varepsilon > 0$ tal que podemos escoller $y$ dentro de $B(x, \varepsilon)$ mantendo a nova circunferencia $C(y)$ baleira de sitios distintos de $a$ e $b$ (Figura 3.5) . En particular, podemos escoller $y$ sobre $B_{ab}$ mediatriz de $a$ e $b$ mantendo $C(y)$ baleira de sitios unha distancia $d$ tal que $0 < d < 2\varepsilon$. Finalmente $x$ está sobre un segmento de Voronoi de lonxitude maior ca 0, subconxunto de $B_{12}$, logo $ab \in \mathcal{D}(P)$. □

**Lema 3.22.** *Dado un diagrama de Voronoi dun conxunto de puntos* $S$ *sen 4 puntos cocirculares,* $V(p_i)$ *é non limitada se e só se* $p_i$ *é un punto do bordo da envolvente convexa de* $S$.

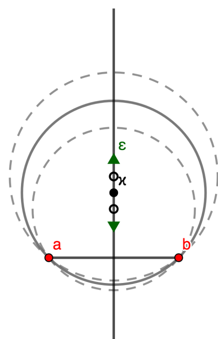


Figura 3.5: Esquema da implicación cara á esquerda na demostración do Teorema 3.21.

*Demostración.* "$\Rightarrow$" Se $p_i$ non está no bordo da envolvente convexa de $S$, pola Proposición 2.5.2 é interno a algún triángulo $\triangle(p_1, p_2, p_3)$. Consideramos os círculos $C(p_1p_2)$, $C(p_2p_3)$ e $C(p_3p_1)$ de xeito que $C(p_1p_2)$ ten centro en $p_i$ e pasa polos puntos $p_1$ e $p_2$. Decatémonos de que todos teñen radio finito. Sexa $A(p_1p_2)$ o arco circular de $C(p_1p_2)$ limitado polo segmento $p_1p_2$ que non contén a $p_i$, todos os puntos de $A(p_1p_2)$ están máis preto de $p_1$ ou $p_2$ que de $p_i$. Sexa $C$ un círculo que conteña a $C(p_1p_2)$, $C(p_2p_3)$ e $C(p_3p_1)$. Vexamos que calquera punto $x$ fóra de $C$ está máis cerca de $p_1$, $p_2$ ou $p_3$ que de $p_i$.

Consideramos o segmento $xp_i$. Polo Teorema 1.5, $xp_i$ interseca un dos lados do triángulo $\triangle(p_1, p_2, p_3)$, tomemos por exemplo $p_1p_2$. Logo tamén interseca $A(p_1p_2)$ nun punto $u$. Como sabemos, $u$ está máis cerca de $p_1$ ou $p_2$ que de $p_i$, de aquí podemos deducir que $V(p_i)$ está contido en $C$, que é limitado (Figura 3.6).

"$\Leftarrow$" Asumamos que $V(p_i)$ é limitado. Sexan $e_1, e_2, \dots, e_k$ ($k \geq 3$) a secuencia dos seus lados. Cada $e_h$, $h = 1, \dots, k$ pertence á bisectriz dun segmento $p_ip'_h$, $p'_h \in S$. Logo $p_i$ é interno ao polígono que ten como vértices $p'_1, p'_2, \dots, p'_k$, polo tanto, non está no bordo da envolvente convexa de $S$. □

**Teorema 3.23.** *O grafo do dual dun diagrama de Voronoi dun conxunto de puntos $S$ sen 4 puntos cocirculares é unha triangulación de Delaunay $S$.*

*Demostración.* Para probar isto, veremos que a envolvente convexa de $S$ particiónase en triángulos determinados polos puntos de $S$. Debemos atopar $\mathcal{T} = \{T(v) : \ v$ é vertice de Voronoi$\}$. Os triángulos construiranse como segue: sexa $v$ un vértice de Voronoi compartido por $V(p_1)$, $V(p_2)$ e $V(p_3)$: $T(v)$ é definido como o triángulo cuxos vértices son $p_1$, $p_2$ e $p_3$.

Decatémonos de que se $T(v)$ interseca o interior de $\mathrm{conv}(S)$, entón $T(v)$ é non dexenerado, é dicir, $p_1$, $p_2$ e $p_3$ son non colineares. Asumamos pola contra que $p_1$, $p_2$, e $p_3$ son colineares, logo os segmentos $p_1p_2$, $p_1p_3$ e $p_2p_3$ pertencen á mesma liña $l$, e as bisectrices

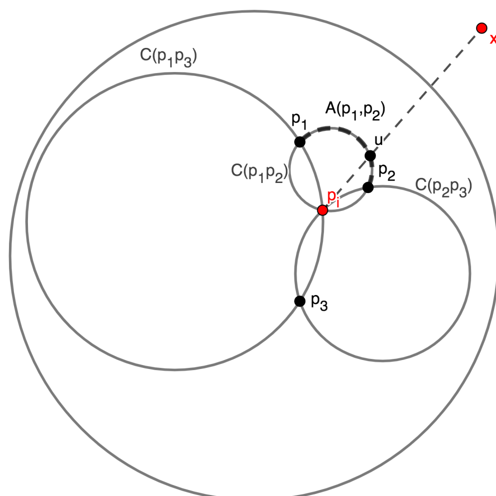


Figura 3.6: Esquema da implicación cara á dereita na Demostración do Lema 3.22.

son paralelas. Logo como $v$ é a intersección das bisectrices e estas son paralelas, $V(1)$, $V(2)$ e $V(3)$ son non limitados, forzosamente $p_1$, $p_2$ e $p_3$ atópanse sobre o bordo da envolvente convexa (noutro caso a envolvente convexa sería non limitada), o que contradí que $T(v)$ se atope no interior de conv$(S)$.

Agora, consideremos dous triángulos $T(v_1)$ e $T(v_2)$ onde $v_1 \neq v_2$, e os círculos $C(v_1)$ e $C(v_2)$ correspondentes. Se $C(v_1)$ e $C(v_2)$ son disxuntos, $T(v_1)$ e $T(v_2)$ tamén o serán. Asumamos que $C(v_1)$ e $C(v_2)$ comparten puntos dos seus interiores. Decatémonos que un círculo non pode estar contido polo outro. Temos entón que $C(v_1)$ e $C(v_2)$ intersecan en dous puntos $q_1$ e $q_2$ que definen unha liña $l$ que separa a $v_1$ e $v_2$. Tamén separa a $T(v_1)$ e $T(v_2)$, xa que noutro caso, habería puntos de $T(v_1)$ máis preto de $v_2$ que de $v_1$. Polo tanto, $T(v_1)$ e $T(v_2)$ non comparten puntos interiores.

Finalmente escollamos un punto $x \in \text{conv}(S)$ e supoñamos que $x \notin T(v)$ para calquera vértice de Voronoi $v$. Isto implica que existe un pequeno disco $B(x, \varepsilon)$ con $\varepsilon > 0$ cuxos puntos teñen a mesma propiedade ca $x$. Sexan $q \in T(v)$ para un $v$ arbitrario, e $y \in B(x, \varepsilon)$ de modo que a liña $l$ pasando por $q$ e $y$ non pase por ningún outro punto de $S$. Entón $l$ intersecará triángulos de $\mathcal{T}$ nun conxunto de intervalos pechados; sexa $t$ o punto final do intervalo máis preto de $x$, con $t$ sobre o bordo do segmento $p_1p_2$ de $T(v_1)$ para algún vértice de Voronoi $v_1$. Agora, sexa $p_3$ o terceiro vértice de $T(v_1)$. Consideramos $l_1$ a recta perpendicular a $p_1p_2$ pasando por $v_1$. A parte inicial de $l_1$ será o segmento entre $V(1)$ e $V(2)$. Como asumimos que $x \in \text{conv}(S)$, e $x$ e $p_3$ están en lados opostos de $p_1p_2$, $p_1p_2$ non pode pertencer ao bordo da envolvente convexa de $S$. Polo tanto, debe existir $v_2$ sobre a recta $l_1$ no lado oposto a $v_1$ respecto de $p_1p_2$. Polo tanto $t$ é interno a $T(v_1) \cup T(v_2)$, e isto

contradí que os puntos de $ty$ non pertencen a ningún triángulo de $\mathcal{T}$. □

**Proposición 3.24.** *O número de elementos (vértices e segmentos de Voronoi) dun diagrama de Voronoi é $O(n)$, onde $n$ é o número de sitios de Voronoi.*

*Demostración.* Por simplicidade, asumimos que non hai 4 puntos cocirculares; logo todos os vértices de Voronoi son de grao 3. Construímos agora o dual $G$ do diagrama de Voronoi $\mathcal{V}(P)$ do modo descrito: cada nodo de $G$ é un sitio de $\mathcal{V}(P)$, e dous nodos están enlazados se as súas rexións de Voronoi comparten un segmento de Voronoi.

Aplicando o Teorema 3.23 baixo a hipótese de que tódolos vértices de Voronoi son de grao 3, temos que todas as caras de $G$ son triángulos. Observemos tamén que é un grafo plano, xa que nunha triangulación os arcos non se intersecan.

Empregando agora o Corolario 3.16, temos que para un grafo plano $G$ que consista nunha triangulación con $n$ nodos hai como moito $3n-6$ arcos incidentes aos nodos e como moito $2n-4$ caras. Como sabemos, corresponden aos $n$ sitios, como moito $3n-6$ segmentos de Voronoi, e como moito $2n-4$ vértices de Voronoi. Finalmente, se eliminamos a suposición de que todos os vértices de Voronoi son non dexenerados, o grafo seguiría sendo plano pero non sería unha triangulación. Pero un grafo plano non triangulado de $n$ nodos só pode ter menos arcos e caras, polo tanto a nosa cota de $O(n)$ mantense. □

## 3.3. Algoritmos

As moitas aplicacións dos diagramas de Voronoi levaron aos investigadores a inventar unha variedade de algoritmos para executalos. Nesta sección examinaremos sen entrar en detalle algúns algoritmos de construción dos diagramas de Voronoi.

### 3.3.1. Intersección de semiplanos

Unha idea inicial é a construción de cada rexión de Voronoi mediante a intersección de $n-1$ semiplanos dada pola ecuación (3.1). Esta intersección de $n$ semiplanos é dual á tarefa de construír a envolvente convexa de $n$ puntos, e pode ser implementada en tempo $O(n \log n)$. Facer isto para cada punto tería un custo de $O(n^2 \log n)$. Debido a este elevado custo non nos interesará ver un algoritmo que siga esta estratexia.

### 3.3.2. Construción incremental

Sexa $\mathcal{V}(P)$ o diagrama de Voronoi para $k = |P|$ puntos xa construído, queremos construír o diagrama $\mathcal{V}(P')$, onde $P' = P \cup \{p\}$ sendo $p$ un punto distinto dos de $P$. Supoñamos que $p$ cae dentro de círculos asociados a vértices de Voronoi, chamémolos $C(v_1), \ldots, C(v_m)$.

Logo estes vértices de $\mathcal{V}(P)$ non poden ser vértices de $\mathcal{V}(P')$, xa que violan a condición de que os círculos sobre os vértices de Voronoi deben estar baleiros de sitios. Por outra parte estes serán os únicos vértices de $\mathcal{V}(P)$ que non serán vértices de $\mathcal{V}(P')$. Estas observacións poden formar un dos algoritmos máis limpos para a construción dos diagramas de Voronoi [10]. O algoritmo emprega $O(n)$ tempo para cada inserción dun punto, resultando unha complexidade total de $O(n^2)$. Pese á complexidade cuadrática, foi un dos métodos máis populares para construír o diagrama, e recentemente revitalizado con aleatorización.

### 3.3.3. Divide e vencerás

O diagrama de Voronoi pode construírse cun complexo algoritmo "divide e vencerás" en $O(n \log n)$ [2]. Cunha complexidade asintoticamente óptima, é un algoritmo cunha implementación complexa. Pasaremos deste algoritmo historicamente importante para centrarnos nalgúns descubrimentos máis recentes.

### 3.3.4. Algoritmo de Fortune

Ata mediados dos 1908's, a maioría das implementacións para a computación do diagrama de Voronoi empregaban o algoritmo incremental $O(n^2)$, aceptando a súa velocidade para evitar as complexidades do "divide e vencerás". Pero en 1985, Fortune inventou un intelixente algoritmo de varrido de plano tan simple como os algoritmos incrementais [3], cunha complexidade no peor caso de $O(n \log n)$. A idea que suxire este algoritmo é varrer unha liña sobre o plano, deixando o problema resolto para a parte na que a liña xa varreu, e sen resolver na parte non alcanzada pola liña. Inicialmente, parece imposible que a liña atope os segmentos de Voronoi da rexión $V(p)$ sen ter alcanzado o punto $p$, pero mostraremos a idea clave que impulsa este algoritmo.

Imaxinemos que os puntos se sitúan no plano $XY$ dun sistema de coordenadas tridimensional. Despregaremos sobre cada sitio $p$ un cono que teña a $p$ como vértice, e cuxa xeneratriz forme 45º coa vertical pasando por $p$. Veremos a terceira dimensión como o tempo. Logo temos círculos en expansión sobre $p$ á velocidade unidade. Agora ben, tendo dous conos sobre os sitios $p_1$ e $p_2$, intersecan nunha curva no espazo, que se atopa contida nun plano vertical, en concreto no plano ortogonal á bisectriz de $p_1p_2$. Logo a proxección desta curva sobre o plano $XY$ é unha liña recta. De aquí podemos deducir que se os conos sobre todos os sitios son opacos e comezan a expandirse dende $z = -\infty$, veríamos o diagrama de Voronoi.

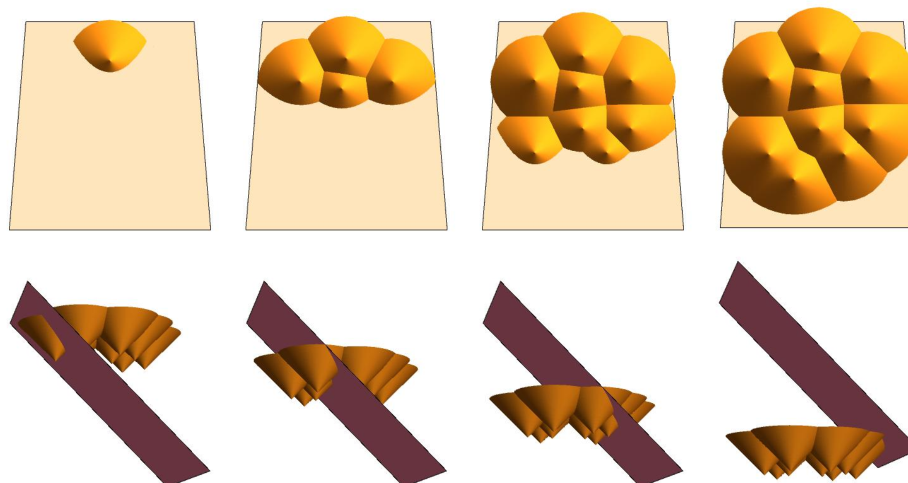

Figura 3.7: Proceso do varrido do plano $\pi$ visto dende abaixo.

Estamos entón listos para describir a idea de Fortune. O seu algoritmo varre os conos cun plano $\pi$ inclinado 45º respecto a $XY$. A liña $L$ que varre é a intersección de $\pi$ co noso plano avanzando no tempo. Asumamos que $L$ é paralelo ao eixo $Y$ e que a coordenada $x$ é $l$. No lado $x < l$ respecto a $L$, o diagrama de Voronoi é visible ata a intersección de $\pi$ coa frontera dereita ($x$ positivo) dos conos. Esta intersección cun só cono é unha parábola, e con varios conos é unha curva obtida como composición de parábolas, a "fronteira parabólica". Dúas destas parábolas na composición enlázanse no punto onde $\pi$ interseca a dous conos, e este debe estar sobre un segmento de Voronoi.

Vemos entón que Fortune resolveu o problema de varrer os segmentos de Voronoi antes de ter varrido os sitios que o xeneran. Como $\pi$ ten o mesmo ángulo que a xeneratriz dos conos, $L$ atopa un sitio $p$ cando $\pi$ interseca o cono que xenera $p$. O diagrama de Voronoi non sempre se constrúe á esquerda de $L$, pero sí se constrúe á esquerda de $L$ polo menos ata a fronteira parabólica. Os puntos que atopamos e se atopan sobre os segmentos de Voronoi son os que trazarán o diagrama a medida que avance o tempo.

Finalmente, o único que o algoritmo precisa almacenar é a fronteira parabólica, e esa información permitirá construír o diagrama. Obter esta fronteira ten un custo computacional de $O(n)$, e moitas veces de $O(\sqrt{n})$, o que supón unha vantaxe significativa cando $n$ é grande. Na práctica, $n$ pode ser grande, se cadra $10^6$ para diagramas, por exemplo, de sistemas de información xeográfica.

# Capítulo 4

# Anexo

Neste anexo adxuntaremos os arquivos completos de código empregados neste traballo.

## 4.1. Estruturas de datos

### 4.1.1. Lista

O código `_abstract_list.py` implementa a funcionalidade básica dunha lista dobremente enlazada [14].

```
import weakref

class EmptyListError(Exception):

class _LinkedList():
    class Node():
        def __init__(self, value, previous=None, next_=None, owner=None):
            self.value = value
            self.previous = previous
            self.next_ = next_
            self._owner = weakref.ref(owner)

        def __repr__(self):
            return repr(self.value)

        def remove(self):
            self._owner()._remove_node(self)

    def __init__(self, iterable=()):
        self._length = 0
```

```
        self.extend(iterable)

    def __repr__(self):
        cls_name = type(self).__qualname__
        values = list(self)
        return f'{cls_name}({values!r})'

    def __iter__(self):
        for node in self.nodes():
            yield node.value

    def __len__(self):
        return self._length

    def __eq__(self, other):
        if not isinstance(other, type(self)) or len(self) != len(other):
            return False
        return all(x == y for x, y in zip(self, other))

    def __getitem__(self, n):
        for index, node in enumerate(self):
            if index == n:
                return node

    def extend(self, iterable):
        for element in iterable:
            self.append(element)

    def _remove_node(self, node):
        if node.previous is not None:
            node.previous.next_ = node.next_
        if node.next_ is not None:
            node.next_.previous = node.previous
        self._length -= 1
        if self.first is node:
            self.first = node.next_
        if self._length == 0:
            self.first = None
        node.previous = node.next_ = None

    def index(self, target):
        for index, value in enumerate(self):
            if target == value:
                return index
            else:
```

```
                raise ValueError(f'{target} is not in list')

    def remove(self, target):
        for node in self.nodes():
            if target == node.value:
                self._remove_node(node)
                break
            else:
                raise ValueError(f'{target} is not in list')

    def rotate(self, offset=1):
        for _ in range(offset):
            self._rotate_one()
```

### 4.1.2. Lista dobremente enlazada

O código `linked_list.py` implementa a funcionalidade básica dunha lista dobremente enlazada.

```
from _abstract_list import _LinkedList, EmptyListError

class LinkedList(_LinkedList):
    def __init__(self, iterable=()):
        self.first = None
        self.last = None
        super().__init__(iterable)

    def append(self, value):
        new = self.Node(value, previous=self.last, owner=self)
        if self.last is not None:
            self.last.next_ = new
        self.last = new
        if self.first is None:
            self.first = new
        self._length += 1

    def appendleft(self, value):
        new = self.Node(value, next_=self.first, owner=self)
        if self.first is not None:
            self.first.previous = new
        self.first = new
        if self.last is None:
            self.last = new
        self._length += 1
```

```
    def _remove_node(self, node):
        if self.last is node:
            self.last = node.previous
        super()._remove_node(node)

    def pop(self):
        if self.last is None:
            raise EmptyListError('Cannot␣remove␣from␣empty␣list.')
        value = self.last.value
        self._remove_node(self.last)
        return value

    def popleft(self):
        if self.first is None:
            raise EmptyListError('Cannot␣remove␣from␣empty␣list.')
        value = self.first.value
        self._remove_node(self.last)
        return value

    def nodes(self):
        current = self.first
        while current is not None:
            yield current
            current = current.next_

    def _rotate_one(self):
        if self.first is not None:
            self.last.next_ = self.first
            self.last = self.first
            self.first = self.first.next_
            self.first.previous = None
            self.last.next_ = None
```

### 4.1.3. Lista circular dobremente enlazada

O código `circular_list.py` é un módulo que define unha lista circular enlazada básica.

```
from operator import attrgetter

from attr import attr

from _abstract_list import _LinkedList, EmptyListError

class CircularList(_LinkedList):
```

```
    def __init__(self, iterable=()):
        self.first = None
        super().__init__(iterable)

    def append(self, value):
        if self.first is None:
            self.first = self.Node(value, owner=self)
            self.first.previous = self.first
            self.first.next_ = self.first
            self._length = 1
        else:
            new = self.Node(
                value,
                previous=self.first.previous,
                next_=self.first,
                owner=self,
            )
            self.first.previous.next_ = new
            self.first.previous = new
            self._length += 1

    def pop(self):
        if self.first is None:
            raise EmptyListError('Cannot␣remove␣from␣empty␣list.')
        value = self.first.previous.value
        self._remove_node(self.first)
        return value

    def nodes(self, start=None):
        if self.first is None:
            return
        start = self.first if start is None else start
        current = start
        while True:
            yield current
            current = current.next_
            if current is start:
                break

    def _rotate_one(self):
        if self.first is not None:
            self.first = self.first.next_

    def rotated_nodes(self, offset=1):
        if offset >= 0:
```

```
            advance = attrgetter('next_')
        else:
            advance = attrgetter('previous')
            offset = -offset
        current = self.first
        for _ in range(offset):
            current = advance(current)
        yield from self.nodes(current)

    def rotated(self, offset=1):
        for node in self.rotated_nodes(offset):
            yield node.value
```

## 4.2. Programas principais

### 4.2.1. Point

O código `point.py` é un módulo que define a clase `Pair`, `Point` e `Vector` cunha serie de funcións operando sobre ditos elementos.

```
from math import acos, sqrt
from numbers import Real

class _Pair:
    __slots__ = ('__x', '__y')

    def __init__(self, x, y):
        self.__x = x
        self.__y = y

    @property
    def x(self):
        return self.__x

    @x.setter
    def x(self, _):
        self._reasign_error()

    @x.deleter
    def x(self):
        self._reasign_error()

    @property
```

```
    def y(self):
        return self.__y

    @y.setter
    def y(self, _):
        self._reasign_error()

    @y.deleter
    def y(self):
        self._reasign_error()

    @classmethod
    def _reasign_error(cls):
        cls_name = cls.__qualname__
        error = f'Coordinates␣of␣object␣"{cls_name}"␣cannot␣be␣modified.'
        raise AttributeError(error)

    def __repr__(self):
        return f'{type(self).__qualname__}({self.x!r},␣{self.y!r})'

    def __eq__(self, other):
        if isinstance(other, type(self)):
            return (self.x, self.y) == (other.x, other.y)
        else:
            return NotImplemented

    def __iter__(self):
        yield from (self.x, self.y)

    def  __hash__(self):
        return hash((self.x, self.y))


class Point(_Pair):
    __slots__ = ()

    def __add__(self, vector):
        if isinstance(vector, Vector):
            return type(self)(self.x + vector.x, self.y + vector.y)
        else:

            return NotImplemented

    def __radd__(self, vector):
        return self + vector
```

```
    def __sub__(self, other):
        if isinstance(other, type(self)):
            return Vector(self.x - other.x, self.y - other.y)
        elif isinstance(other, Vector):
            return Point(self.x - other.x, self.y - other.y)
        else:
            return NotImplemented


class Vector(_Pair):
    def __add__(self, vector):
        cls = type(self)
        if isinstance(vector, cls):
            return cls(self.x + vector.x, self.y + vector.y)
        else:
            return NotImplemented

    def __mul__(self, scalar):
        if isinstance(scalar, Real):
            return type(self)(scalar * self.x, scalar * self.y)
        else:
            return NotImplemented

    def __truediv__(self, scalar):
        return type(self)(self.x / scalar, self.y / scalar)

    def __rmul__(self, scalar):
        return self * scalar

    def __neg__(self):
        return (-1)*self

    def __sub__(self, vector):
        return self + (-1)*vector

    def norm(self):
        return sqrt(scalar_product(self, self))

    def unit_orthogonal(self):
        v = Vector(-self.y, self.x)
        return v / v.norm()

    def scalar_product(v, w):
        return v.x*w.x + v.y*w.y
```

```
    def angle(v, w):
        return acos(scalar_product(v, w) / (v.norm()*w.norm()))

    def distance(p, q):
        return (q-p).norm()
```

### 4.2.2. Segment

O código `segment.py` é un módulo que define a clase `Segment`, un segmento no plano. Implementa unha serie de funcións operando sobre un ou varios segmentos.

```
import numbers
from operator import xor
from math import sqrt

from point import Point, scalar_product
from triangle import Triangle, are_collinear


class Segment():
    def __init__(self, p, q):
    self.start = Point(*p)
    self.end = Point(*q)

    def __repr__(self):
    return f'{type(self).__qualname__}({self.start!r},␣{self.end!r})'

    def __iter__(self):
    yield from (self.start, self.end)

    def __eq__(self, other):
    return (
        isinstance(other, type(self)) and
        self.start == other.start and self.end == other.end
    )

    def is_left(self, p):
        return Triangle(self.start, self.end, p).area_2() > 0

    def is_left_on(self, p):
        return Triangle(self.start, self.end, p).area_2() >= 0
```

```
    def is_between(self, p):
        a, b = self
        if not are_collinear(a, b, p):
            return False
        elif a.x != b.x:
            return (a.x >= p.x and p.x >= b.x) or (a.x <= p.x and p.x <= b.
                x)
        else:
            return (a.y >= p.y and p.y >= b.y) or (a.y <= p.y and p.y <= b.
                y)

    def __contains__(self, p):
        return self.is_between(p)

    def opposite(self):
        return type(self)(self.end, self.start)

    def distance(self, point):
        a, b = self
        return scalar_product((b-a).unit_orthogonal(), point-a)


def intersect_properly(segment1, segment2):
    a, b = tuple(segment1)
    c, d = tuple(segment2)
    return (
        not (
            are_collinear(a, b, c) or
            are_collinear(a, b, d) or
            are_collinear(c, d, a) or
            are_collinear(c, d, b)
        ) and (
            xor(segment1.is_left(c), segment1.is_left(d)) and
            xor(segment2.is_left(a), segment2.is_left(b))
        )
    )

def intersect(segment1, segment2):
    return (
        intersect_properly(segment1, segment2) or
        segment2.start in segment1 or
        segment2.end in segment1 or
        segment1.start in segment2 or
        segment1.end in segment2
    )
```

### 4.2.3. Triangle

O código `triangle.py` é un módulo que define a clase `Triangle`, un triángulo no plano. Implementa unha serie de funcións operando sobre un ou varios triángulos.

```
from point import Point

class Triangle:
    def __init__(self, a, b, c):
        self.a = Point(*a)
        self.b = Point(*b)
        self.c = Point(*c)

    def area_2(self):
        a, b, c = self.a, self.b, self.c
        return (b.x - a.x)*(c.y - a.y) - (c.x - a.x)*(b.y - a.y)

    def is_degenerate(self):
        return self.area_2() == 0

    def is_inside(self, point):
        clockwise = self.area_2()
        if clockwise == 0:
            return False
        else:
            return (
                clockwise * Triangle(self.a, self.b, point).area_2() >= 0
                    and
                clockwise * Triangle(self.b, self.c, point).area_2() >= 0
                    and
                clockwise * Triangle(self.c, self.a, point).area_2() >= 0
            )

    def __contains__(self, point):
        return self.is_inside(point)

def are_collinear(a, b, c):
    return Triangle(a, b, c).is_degenerate()
```

### 4.2.4. Polygon

O código `polygon.py` é un módulo que define as clases `Vertex` e `Polygon`, vértice e polígono no plano. Implementa unha serie de funcións operando sobre un polígono, entre elas se atopan varias funcións de triangulación descritas no traballo.

```python
from circular_list import CircularList as LinkedList
from point import Point
from triangle import Triangle
from segment import Segment, intersect

class Vertex(Point):
    def __init__(self, x, y):
        super().__init__(x, y)
        self.is_ear = None

class Polygon():
    def __init__(self, vertices):
        self.vertices = LinkedList(Vertex(*vertex) for vertex in vertices)

    def __repr__(self):
        return f'{type(self).__qualname__}({list(self.vertices)!r})'

    def area_2(self, point=None):
        if point is None:
            point = self.vertices[0]
        pairs = zip(self.vertices, self.vertices.rotated())
        return sum(
            Triangle(point, vertex, next_).area_2()
            for vertex, next_ in pairs
        )

    def area_1_14(self):
        pairs = zip(self.vertices, self.vertices.rotated())
        return sum(
            vertex.x*next.y - vertex.y*next.x
            for vertex, next in pairs
        )

    def area(self):
        return self.area_2() / 2

    def _is_possible_diagonal(self, segment):
        pairs = zip(self.vertices, self.vertices.rotated())
        end_points = tuple(segment)
```

```
        for vertex, next_ in pairs:
            if (
                not Point(*vertex) in end_points and
                not Point(*next_) in end_points and
                intersect(segment, Segment(vertex, next_))
            ):
                return False
            else:
                return True

    def _in_cone(self, start, end):
        segment = Segment(start.value, end.value)
        previous = start.previous.value
        next_ = start.next.value
        if Segment(start.value, next_).is_left_on(previous):
            return (
                segment.is_left(previous) and
                segment.opposite().is_left(next_)
            )
        else:
            return not (
                segment.is_left_on(next_) and
                segment.opposite().is_left_on(previous)
            )

    def _is_diagonal(self, start, end):
        return (
            self._in_cone(start, end) and
            self._in_cone(end, start) and
            self._is_possible_diagonal(Segment(start.value, end.value))
        )

    def _ear_init(self):
        trios = zip(
            self.vertices.nodes(),
            self.vertices.rotated_nodes(1),
            self.vertices.rotated_nodes(2),
        )
        for v0, v1, v2 in trios:
            v1.value.is_ear = self._is_diagonal(v0, v2)

    def triangulate(self):
        new = type(self)(self.vertices)
        yield from new._triangulate()
```

```
    def _triangulate(self):
        self._ear_init()
        while len(self.vertices) > 3:
            five_consecutive = zip(*(
                self.vertices.rotated_nodes(offset)
                for offset in (-2, -1, 0, 1, 2)
            ))
            for v0, v1, v2, v3, v4 in five_consecutive:
                if v2.value.is_ear:
                    yield Segment(v1.value, v3.value)
                    v1.value.is_ear = self._is_diagonal(v0, v3)
                    v3.value.is_ear = self._is_diagonal(v1, v4)
                    v2.remove()
                    break
```

### 4.2.5. Convex Hull

O código `convex_hull.py` é un módulo que define funcións que actúan sobre listas (de puntos), tamén define a clase `GrahamPoint`. As funcións que se tratan neste código serán as que teñan como obxectivo o cálculo da envolvente convexa de puntos.

```
from itertools import combinations, chain
from functools import cmp_to_key
from linked_list import LinkedList

from point import Point, Vector, angle
from triangle import Triangle
from segment import Segment


def nonextreme_points(points):
    for p1, p2, p3 in combinations(points, 3):
        for q in points:
            if q not in (p1, p2, p3) and q in Triangle(p1, p2, p3):
                yield q


def extreme_edges(points):
    for p1, p2 in combinations(points, 2):
        edge = Segment(p1, p2)
        all_left = all(
            edge.is_left_on(q) for q in points if q != p1 and q != p2
        )
        opposite = edge.opposite()
```

```
        all_right = all(
            opposite.is_left_on(q) for q in points if q != p1 and q != p2
        )
        if all_left or all_right:
            yield edge


def gift_wrapping(points):
    first = last = Point(*min(points, key=lambda point: (point.y, point.x))
        )
    previous = last - Vector(1, 0)
    while True:
        yield last
        _, q = min(
            (angle(last-previous, p-last), tuple(p))
            for p in points
            if p != last and p != previous
        )
        previous = last
        last = Point(*q)
        if last == first:
            break

def quick_hull(points):
    right_lower = Point(*max(points, key=lambda point: (point.x, -point.y))
        )
    left_upper = Point(*min(points, key=lambda point: (point.x, -point.y)))
    yield right_lower
    segment = Segment(right_lower, left_upper)
    yield from _quick_hull(segment, _areas(segment, points))
    yield left_upper
    opposite = segment.opposite()
    yield from _quick_hull(opposite, _areas(opposite, points))

def _areas(segment, points):
    a, b = segment
    return [
        (d, p) for p in points
        if p != b and (d := -Triangle(a, b, p).area_2()) > 0
    ]

def _quick_hull(segment, distance_points):
    if distance_points:
        _, farthest = max(distance_points, key=lambda pair: pair[0])
        points = [p for _, p in distance_points]
```

```
        segment1 = Segment(segment.start, farthest)
        segment2 = Segment(farthest, segment.end)
        yield from _quick_hull(segment1, _areas(segment1, points))
        yield farthest
        yield from _quick_hull(segment2, _areas(segment2, points))

def graham(points):
    lower_right = min(points, key=lambda point: (point.y, -point.x))
    points = sorted(
        [_GrahamPoint(*p) for p in points if p != lower_right],
        key=cmp_to_key(angle_closer(lower_right))
    )
    hull = LinkedList(chain([lower_right], points))
    previous = hull.first
    last = previous.next
    while last.value.skip:
        last = _remove_and_advance(last)
    last.value = Point(*last.value)
    current = last.next
    while current is not None:
        if not current.value.skip:
            if Segment(previous.value, last.value).is_left(current.value):
                current.value = Point(*current.value)
                previous, last, current = last, current, current.next
            else:
                last.remove()
                last, previous = previous, previous.previous
        else:
            current = _remove_and_advance(current)
    return hull

class _GrahamPoint(Point):
    def __init__(self, x, y):
        super().__init__(x, y)
        self.skip = False

def _remove_and_advance(node):
    aux = node
    node = node.next
    aux.remove()
    return node

def angle_closer(base):
    def compare(p, q):
        area = -Triangle(base, p, q).area_2()
```

```
        if area != 0:
            return area
        else:
            x = abs(p.x - base.x) - abs(q.x - base.x)
            y = abs(p.y - base.y) - abs(q.y - base.y)
            if x < 0 or y < 0:
                p.skip = True
                return -1
            elif x > 0 or y > 0:
                q.skip = True
                return 1
            else:
                if not p.skip and not q.skip:
                    q.skip = True
                return 0
    return compare
```

# Bibliografía

[1] Joseph O'Rourke (1997), *Computational Geometry in C. Second edition.*

[2] Franco P. Preparata e Michael Ian Shamos (1985). *Computational Geometry.* Springer-Velag, New York, 1985.

[3] Steven Fortune (1992). "Voronoi diagrams and Delaunay triangulations". *Computing in Euclidean Geometry* F. K. Hwang e D. Z. Du editores. World Scientific, Singapur, 1992.

[4] Meisters, G. H. (1975), "Polygons have ears." *American Mathematical Monthly.*

[5] Eckhoff, J. (1993). "Helly, Radon, and Carathéodory type theorems". *Handbook of Convex Geometry.*

[6] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001) [1990]. "33.3: Finding the convex hull". *Introduction to Algorithms* 2ª edición.

[7] Barber, C. Bradford; Dobkin, David P.; Huhdanpaa, Hannu (1 de diciembre de 1996). *The quickhull algorithm for convex hulls. Implementing Quicksort programs.*

[8] Nahin, Paul J. (2006). *Dr. Euler's Fabulous Formula: Cures Many Mathematical Ills.*

[9] Aldous J.M., Wilson R., *Graphs and Applications: An Introductory Approach.* Springer, 2000.

[10] P. J. Green e R. Sibson (1978). "Computing Dirichlet tessellations in the plane". *Computer Journal*, Singapur, 1992.

[11] Sedgewick, R. (1978). "Implementing Quicksort programs". *Communications of the ACM.*

[12] B. Delaunay, "Sur la sphere vide. A la mémoire de Georges Voronoi. Izvestia Akademii Nauk SSSR, Otdelenie Matematicheskikh i Estestvennykh Nauk" *Bulletin of Academy of Sciences of the USSR*, p. 793-800, 1934.

[13] Mark de Berg, Marc van Kreveld, Mark Overmars; *Computational Geometry: Algorithms and Applications.*

[14] Joyanes, Luis y Zahonero Martínez, I. *Algoritmos y estructuras de datos: una perspectiva en C.* 1ª edición, Madrid, McGraw-Hill Interamericana de España S.L, 2004.