10 Computational Geometry

Why Study Computational Geometry?

- **It unifies discrete and continuous mathematics**. Computational geometry sits at the intersection of analysis, topology, and algorithm design.
- **It offers fast and reliable geometric computation**. Many tasks in graphics, robotics, CAD, GIS, and simulation involve computational geometry operations such as intersection tests, convex hulls, triangulations, and nearest-neighbour queries.
- **It provides algorithms with guaranteed performance**. Within computational geometry, combinatorics, graph theory, topology, convexity, and numerical analysis allow the design of algorithms with known time and space bounds. This is crucial for applications that must run in real time, or have to scale to massive datasets.

10.1 Introduction

Computational geometry became a hot topic in computer science in the 1990s, although mathematicians such as Dirichet and Voronoi had been developing similar ideas in the 1850s and 1900s respectively. The modern aim is to develop data structures and algorithms for geometric objects (e.g. for points, lines, line segments, triangles, polygons) that are both exact and fast. Typical application areas include graphics, robotics (e.g. path finding), Geographical Information Systems, CAD/CAM (e.g. for the construction and simplification of objects using set operations), database search, and chip layout.

Devising exact algorithms has proved troublesome because of the computer's inherent limited numerical precision when dealing with floats, and also because many 'intuitive' algorithms become much more complex when edge cases are considered. For instance, should a line segment that is just touching another be considered to be intersecting it, and how should 'touching' be defined?

A common solution to the floats problem is to pepper code with approximation tests that determine if two values are sufficiently 'close'. This works most of the time but isn't mathematically sound. For instance, point *a* may be close to point *b*, and *b* close to point *c*, but *a* may not necessarily be close to *c*, and so transitivity fails.

As a consequence, computational geometry should be treated in the same way as cryptography – develop your own code for the purpose of learning the concepts, but turn to a trusted library when you need to write real applications. We suggest CGAL, the Computational Geometry Library, which is robust, fast (it utilizes C++ internally), and offers a very wide range of algorithms. A downside is that the Python port (`https://github.com/CGAL/cgal-swig-bindings/wiki`) is only partial and its installation is a little tricky. An alternative is Shapely (`https://shapely.readthedocs.io/en/stable/`) which has less features than CGAL.

In this chapter, we'll look at three problems: segment intersection, finding a convex hull, and searching for the closest pair of points. Of course, computational geometry is a huge subject, so for more details please consult one of the excellent standard texts, such as Berg [**dBOCvKO08**] or O'Rourke [**O'R98**].

10.2 Geometry Data Structures

Our Python classes for representing points, line segments, and vectors are stored in `point.py`, `segment.py`, and `vector.py` respectively, and a range of tests are in `GeomTests.py`.

The Point class is based around the manipulation of an *x* and *y* coordinate. For instance, the equality function is:

```
def __eq__(self, p):
  if not Point.isPoint(p):
    return False
  return abs(self.x - p.x) < EPS and \
         abs(self.y - p.y) < EPS
```

Currently, the epsilon value EPS is set to 1e-8.

The class contains functions for calculating the distance between points, the mid-point between points, and the angles between them. There's also a draw() method which uses matplotlib.

10.3 The Dot and Cross Products

The dot and cross products are at the core of many Computational Geometry algorithms. They're implemented in our Vector class:

```
def dot(u, v):   # u and v are vectors
  return ((u.x * v.x) + (u.y * v.y))

def cross(u, v):
  return ((u.x * v.y) - (u.y * v.x))
```

Both operations have trigonometric and geometric interpretations.

10.3.1 The dot product. The dot product can be defined as the multiple of the magnitudes of the two vectors and the cosine of the angle between them, as illustrated in Fig. 10.1.

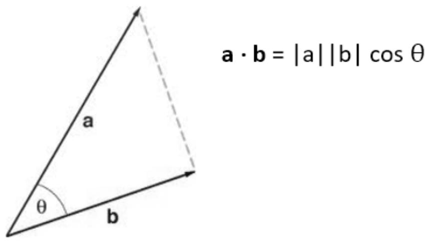

Figure 10.1. The dot product as trigonometry

In terms of geometry, the operation can be viewed as the length obtained when a vector **a** is projected onto the unit vector version of **b**, $\frac{\mathbf{b}}{||b||}$ (see Fig. 10.2).

The sign of the projected length can be employed as a measure of the relative directions of the two vectors. A positive value means that **a** and **b** are roughly pointing in the same direction, a negative value means they are pointing in opposite directions, and a 0 signifies that they are perpendicular (i.e. **a** produces no 'shadow' when projected onto **b**).

It's possible to use the dot product to define a perpendicular, as in Fig. 10.3.

The vector from points o to p is the unit vector $\hat{\mathbf{b}}$ in that direction multiplied by the projected length of **a**:

$$\begin{aligned}\overrightarrow{op} &= \frac{\mathbf{a}\cdot\mathbf{b}}{||b||} * \frac{\mathbf{b}}{||b||} \\ &= \frac{\mathbf{a}\cdot\mathbf{b}}{||b||^2}\mathbf{b}\end{aligned}$$

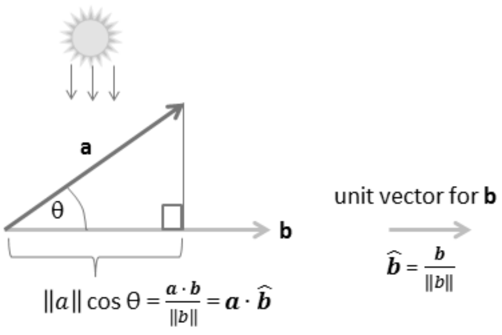
a
θ
b
unit vector for **b**
$\hat{b} = \frac{b}{||b||}$
$||a|| \cos\theta = \frac{a \cdot b}{||b||} = a \cdot \hat{b}$

Figure 10.2. The dot product as geometry

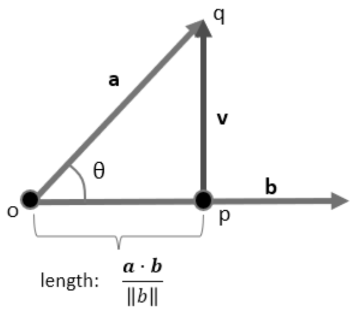

Figure 10.3. The perpendicular vector **v**

The perpendicular **v** is obtained by vector arithmetic:

$$\mathbf{v} = \mathbf{a} - \frac{\mathbf{a} \cdot \mathbf{b}}{||b||^2}\mathbf{b}$$

10.3.2 The cross product. The cross product is usually defined using a determinant. For example, the cross product of two vectors $\mathbf{p} = (x_1, y_1)$ and $\mathbf{q} = (x_2, y_2)$ is:

$$\begin{aligned} \mathbf{p} \times \mathbf{q} &= \begin{vmatrix} x_1 & x_2 \\ y_1 & y_2 \end{vmatrix} \\ &= x_1 y_2 - x_2 y_1 \end{aligned} \tag{10.1}$$

It's helpful to think of the operation as a 3D concept which, given two vectors, defines a perpendicular according to the 'right-hand rule'. It considers the vectors in a counter-clockwise order, as shown in Fig. 10.4(a).

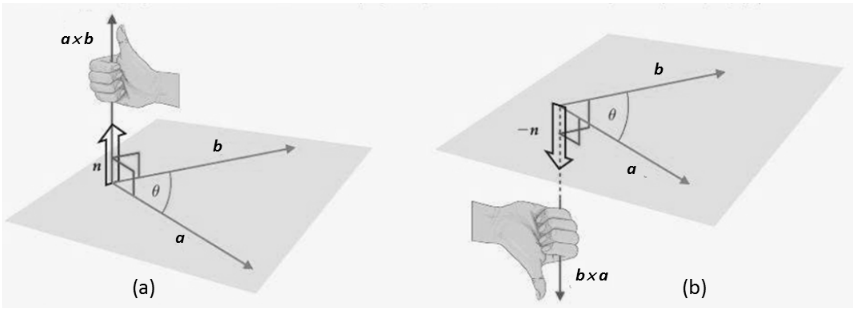

Figure 10.4. The right-hand rule

The cross product of $\mathbf{a} = (a_1, a_2, a_3)$ and $\mathbf{b} = (b_1, b_2, b_3)$ can be written as:

$$\mathbf{a} \times \mathbf{b} = \begin{vmatrix} i & j & k \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix}$$

where $\mathbf{i}, \mathbf{j}$, and $\mathbf{k}$ are unit vectors along the x-, y-, and z- axes. This expands to:

$$\mathbf{a} \times \mathbf{b} = \begin{vmatrix} a_2 & a_3 \\ b_2 & b_3 \end{vmatrix} \mathbf{i} - \begin{vmatrix} a_1 & a_3 \\ b_1 & b_3 \end{vmatrix} \mathbf{j} + \begin{vmatrix} a_1 & a_2 \\ b_1 & b_2 \end{vmatrix} \mathbf{k}$$

A consequence of the right-hand rule is that if the vectors are considered in clockwise order, as in Fig. 10.4(b), then the perpendicular will point in the opposite direction. This offers a way to determine the relative ordering of vectors based on the cross product sign.

The operation also has a trigonometric interpretation:

$$\mathbf{a} \times \mathbf{b} = \|a\| \, \|b\| \sin\theta \, \mathbf{n}$$

The magnitude of the perpendicular vector $\mathbf{n}$ is equal to the area of the parallelogram that the vectors span, as depicted in Fig. 10.5.

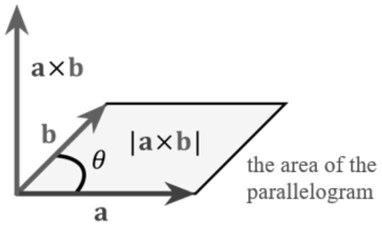

Figure 10.5. The area of a cross product

This area is the base of the parallelogram ($\|\mathbf{a}\|$) multiplied by its height ($\|\mathbf{b}\| \sin\theta$), which also provides a way to obtain the angle θ between the vectors.

The following example comes from GeomTests.py:

```
orig = Point(0, 0)
pt1 = Point(3, -5)
pt2 = Point(2, 6)
print("pt1, pt2:", pt1, pt2)
print("Dot product:", Vector.dot(pt1, pt2))
print("Cross product:", Vector.cross(pt1, pt2))
print(f"Angle: {math.degrees( Vector.angle(pt1, orig, pt2)):.2f} degrees")
```

The output is:

```
pt1, pt2: (3.0, -5.0) (2.0, 6.0)
Dot product: -24.0
Cross product: 28.0
Angle: 130.60 degrees
```

10.4 The ccw() function

Many computational geometry algorithms rely on the ccw() function which determines if the two vectors supplied as three points are in counter-clockwise order. The function returns 1 if they are, -1 if they are in clockwise order, or 0 if they are collinear. Fig. 10.6 shows the three possibilities for `ccw`(p_0, p_1, p_2).

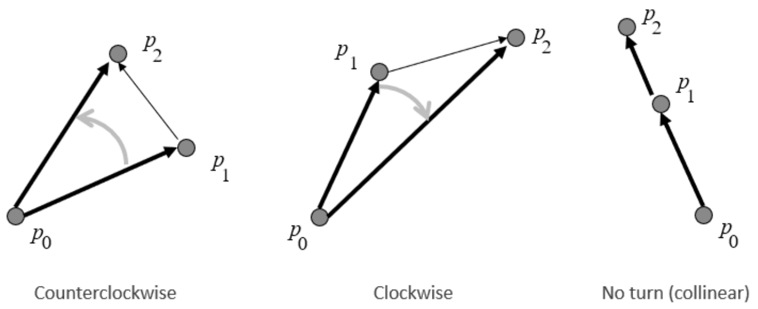

Figure 10.6. Vector ordering using ccw()

ccw() can be defined in a few lines by converting its point arguments to vectors and calling the cross product (see Listing 10.1; vector.py).

```
@staticmethod
def ccw(p, q, r):
  '''
```

```
   returns 0, 1, or -1
    0 if p-->r and p-->q are collinear
    1 if p--> r is left (counter-clockwise) of p-->q
   -1 if p--> r is right (clockwise) of p-->q
  '''
  pq = Vector.toVec(p, q)
  pr = Vector.toVec(p, r)
  cross = Vector.cross(pq, pr)
  if abs(cross) < EPS:
    return 0
  elif cross > 0:
    return 1
  else:
    return -1
```

Listing 10.1. ccw() in the vector class

Float errors can arise in ccw() because of its reliance on Vector.cross() which uses subtraction (see the code for cross() at the start of Sec. 10.3). For example, if one of the points is nearly collinear to the line segment defined by the other two, then rounding errors associated with the subtraction can overwhelm the result. Such a situation is shown in Fig. 10.7, where ccw(p,s,r) *might* incorrectly determine that $\overrightarrow{pr}$ is a clockwise rotation from $\overrightarrow{ps}$.

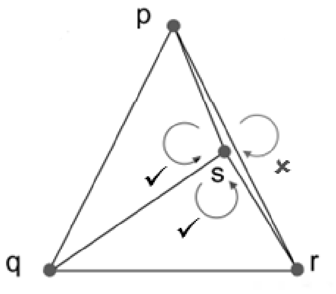

Figure 10.7. Failure of ccw()

This could have other consequences. For instance, it might not be possible to determine that *s* is inside the $\triangle pqr$.

10.5 Segment Intersection

A segment is defined in terms of two endpoints, and its class includes functions related to segment intersection. They can be implemented by using ccw() to compare the directions of the four points making up the two segments. The situation is illustrated in 10.8.

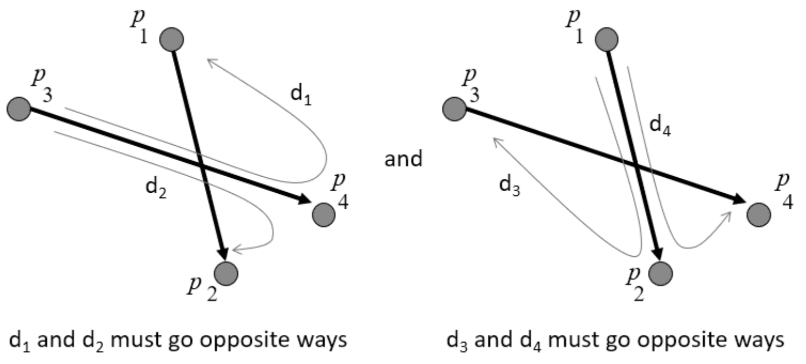
p_1
p_2
p_3
p_4
d_1
d_2
and
d_3
d_4
d_1 and d_2 must go opposite ways
d_3 and d_4 must go opposite ways

Figure 10.8. Segment intersection using ccw()

The code is in Listing 10.2 (segment.py).

```
def intersects(self, seg):
  # does seg intersect with this one?

  # Find the 4 orientations required
  d1 = Vector.ccw(self.p1, self.p2, seg.p1)
  d2 = Vector.ccw(self.p1, self.p2, seg.p2)
  d3 = Vector.ccw(seg.p1, seg.p2, self.p1)
  d4 = Vector.ccw(seg.p1, seg.p2, self.p2)

  # General case
  if ((d1 != d2) and (d3 != d4)):
      return True

  # Special Cases
  # collinear if seg.p1 lies on this segment
  if ((d1 == 0) and self.onSegment(seg.p1)):
      return True

  # collinear if seg.p2 lies on this segment
  if ((d2 == 0) and self.onSegment(seg.p2)):
      return True

  # collinear if this p1 lies on other segment
  if ((d3 == 0) and seg.onSegment(self.p1)):
      return True

  # collinear if this p2 lies on other segment
  if ((d4 == 0) and seg.onSegment(self.p2)):
      return True

  # If none of the cases
```

```
    return False
```

Listing 10.2. intersect() in the segment class

As is typical with computational geometry, most of the function is taken up with checking for boundary cases, when one of the points is located on the other segment.

The following shows the use of intersects() and also the drawing of the segments using matplotlib in Fig. 10.9.

```
seg1 = Segment(Point(1, 1), Point(10, 7))
seg2 = Segment(Point(1, 2), Point(10, 2))
print("seg1, seg2:", seg1, seg2)
print("Intersects?", seg1.intersects(seg2))
graphing()   # initialize matplotlib
seg1.draw("seg1")
seg2.draw("seg2")
plt.show()
```

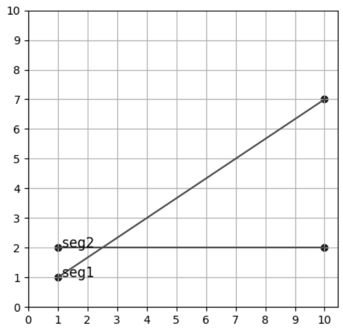

Figure 10.9. Segment intersection

intersects() only returns True or False; to obtain the intersection point, intersectPt() should be called. It treats the two segments as lines of the form $Ax + By + C = 0$ and solves the two equations:

```
def intersectPt(self, seg):
  # where does seg intersect with this segment?

  # Segment AB represented as a1x + b1y = c1
  a1 = self.p2.y - self.p1.y
  b1 = self.p1.x - self.p2.x
  c1 = self.p1.y * b1 + self.p1.x * a1

  # Segment CD represented as a2x + b2y = c2
  a2 = seg.p2.y - seg.p1.y
```

```
    b2 = seg.p1.x - seg.p2.x
    c2 = seg.p1.y * b2 + seg.p1.x * a2

    det = a1 * b2 - a2 * b1
    if det != 0:
      x = (b2 * c1 - b1 * c2) / det
      y = (a1 * c2 - a2 * c1) / det
      return Point(x,y)

    return None
```

Listing 10.3. intersectPt() in the segment class

10.5.1 The Intersection of Multiple Line Segments.

The brute-force approach for finding all the intersections of n line segments is to examine every pair. This will involve $n(n-1)/2$ comparisons in the worst case, and so have a running time of $0(n^2)$. However, if the segments are sparsely distributed, then a lot of that work will be to no purpose; a faster algorithm is needed.

The *Bentley-Ottmann* algorithm utilizes a vertical sweep line that moves from left to right over the segments (which are assumed to be non-vertical themselves). As the line moves it keeps a tally of *event points* which consist of segment end-points and intersections. The points are managed in an queue ordered by their coordinates which means that far fewer comparisons need to be carried out, ultimately reducing the search time to $0(n \log n)$. The drawback is the need for more complex point and segment classes, and a queue which supports the efficient re-ordering of points as the sweep line progresses. Code for Bentley-Ottmann can be found in `sweep.py` and `treeset.py`, but will not be described here.

10.6 Finding the Convex Hull

A planar region R is convex when all the pairs of points p, q in R, form line segments $\overline{pq}$ that lie completely inside R (see the examples in Fig. 10.10).

The convex hull of a set of points P is the smallest convex region that contains all of them. From a physical perspective, the hull is the shape taken by a rubber band stretched around all the points (Fig. 10.11). (Incidentally, this 'band' technique appears to be a constant time operation, easily beating other approaches, but remember that initialization costs must be factored into the running time.)

The *Graham Scan* utilizes a rotating sweep line with running time $O(n \log n)$. Before the sweep begins there are two set-up stages:

(1) The point with the smallest y value is located, and thereafter called the *anchor*. If there are several smallest points, then the one with the largest x value is chosen. This will take linear time.

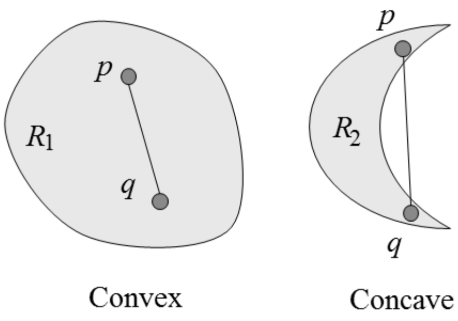

Figure 10.10. Convex and concave regions

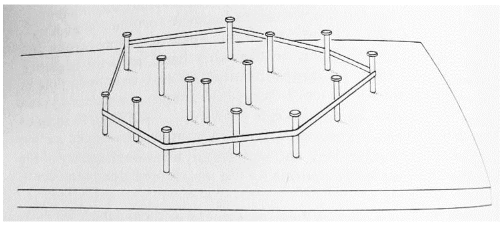

Figure 10.11. A convex hull

(2) The points are sorted based on the angle they make with the anchor. If two points have the same angle, then they're sorted based on their distance from the anchor. This will take $O(n \log n)$ time if merge sort is employed. The angle calculation is assumed to take constant time (see Point.polarAngle() which uses math.atan2()).

The scan starts from the anchor (p_0 in Fig. 10.12), and rotates counter-clockwise through the sorted points.

The sweep line applies a turning test to the last three points encountered (they are labeled as A, B, C in Fig. 10.13). Is the current point (C) to the left or right of the line segment formed by the other two points ($\overline{AB}$)?

If the current point requires a left turn then it's added to the hull and the sweep line moves on to the next point. This occurs in Fig. 10.13 (1) and the new set of points becomes B, C, and D. The question now becomes whether D is to the left or right of $\overline{BC}$?

A right turn means that the point at the end of the line segment can not be part of the convex hull and should be removed. This occurs in Fig. 10.13 (2) when C is discarded. D remains the current point but the line segment changes to $\overline{AB}$.

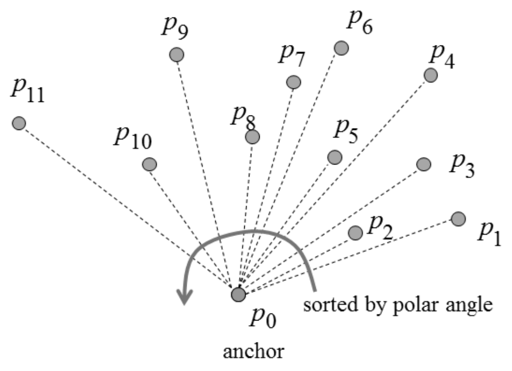

Figure 10.12. The Graham scan

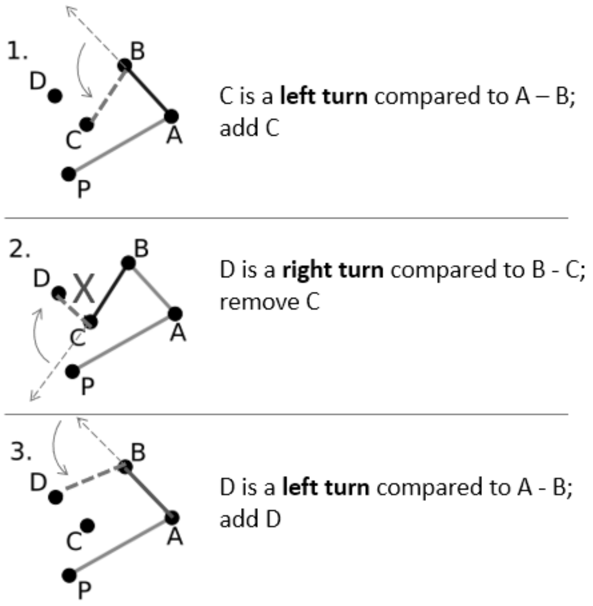
1.
B
D
C
A
P
C is a **left turn** compared to A – B; add C
2.
B
D
X
C
A
P
D is a **right turn** compared to B - C; remove C
3.
B
D
C
A
P
D is a **left turn** compared to A - B; add D

Figure 10.13. Turn testing in the Graham scan

Point removal will continue for as long as the current point and the preceding line segment form a right turn. This will require earlier points in the hull to be stored so they can be used to build new line segments when points are deleted.

The convexHull() code in Listing 10.4 (convexHull.py) utilizes ccw() to determine the turn type.

```
def convexHull(pts):
  # Find a point with the lowest y value,
  anchorPt = pts[0]
  for _, pt in enumerate(pts):
    if pt.y < anchorPt.y:
      anchorPt = pt
    elif pt.y == anchorPt.y and pt.x > anchorPt.x:
      anchorPt = pt
  # print("Anchor:", anchorPt)

  # Sort all the points based on the angle they make with the anchor
      pt.
  ptAngles = []
  for pt in pts:
    ptAngles.append([pt, anchorPt.polarAngle(pt)])
  ptAngles.sort( key=lambda x:x[1])
  sortedPts =  [ pt for [pt,_] in ptAngles]

  hull = [anchorPt, sortedPts[0]]
  # test turning if points
  for pt in sortedPts[1:]:
    while Vector.ccw(hull[-2], hull[-1], pt) <= 0:
      del hull[-1] # removal
    hull.append(pt)

  return hull
```

Listing 10.4. Implementing the convex hull

A series of ccw() tests are applied to the $n-1$ points spread around the anchor. Since ccw() is a constant time operation, the sweep has running time $O(n)$. When this is combined with the earlier set-up costs, Graham's scan takes $O(n \log n)$.

The test code in convexHull.py calls convexHull() with 100 random points. The returned hull is used to draw line segments around the points as shown in Fig. 10.14.

The ∘ point in Fig. 10.14 is the result of a call to isInside() (Listing 10.5) (convexHull.py) which returns True or False depending on whether the point is inside the hull. The code uses ccw() to check if the point is on the left of every segment making up the hull.

```
def isInside(poly, p):
  n = len(poly)
  if n < 3:
    return False
  for i in range(n):
    if Vector.ccw(poly[i], poly[(i+1)%n], p) == -1:  # on right
      return False
```

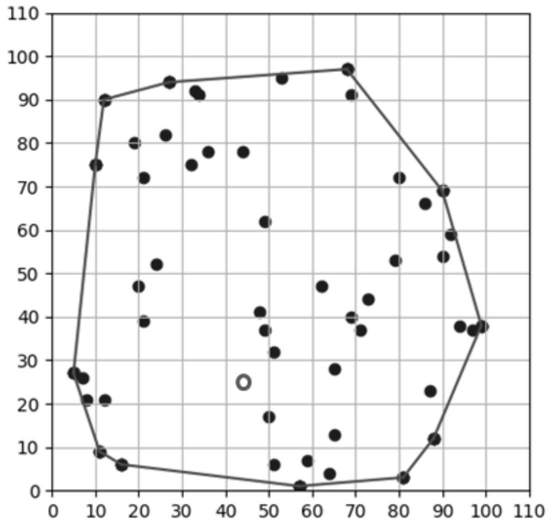

Figure 10.14. The convex hull of random points

```
    # p always on left or collinear
    return True
```

Listing 10.5. isInside() in convexHull.py

10.7 Finding the Closest Pair of Points

Another classic CG problem is finding a pair of points p, q such that $|p-q|$ is the minimum of all $|p_i - p_j|$ in a set of n points $P = \{p_1, \cdots p_n\}$.

As with the convex hull, there's an $O(n^2)$ algorithm which computes the absolute distances between all pairs of points. There's also a sweep line technique which has running time $O(n \log n)$; we won't discuss either of these, although their Python code can be found in nearestPts.py and closestSweep.py respectively. Instead, we'll utilize a divide-and-conquer approach.

The divide phase determines the median x-coordinate of the points, which is used to split the search into two halves. This requires the points to be sorted beforehand, which can be done in $O(n \log n)$ time.

The division process recursively progresses until a single distance remains. As the function calls return, the smallest distance is selected and passed up. At the top-level, the search produces two distances (labeled as d_1 and d_2 in Fig. 10.15).

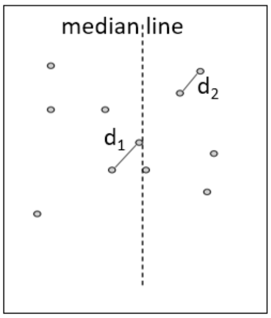

Figure 10.15. Shortest distances in two halved spaces

Result generation is not just a matter of selecting the smaller of the two distances. The search may have missed close points across the median line – occurring in a narrow vertical strip of width $2d$, where d is the minimum of d_1 and d_2 (see Fig. 10.16).

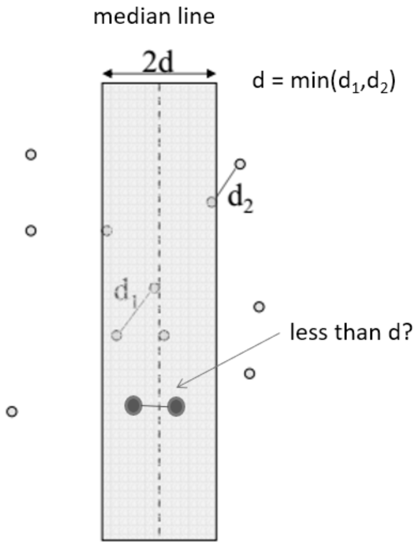

Figure 10.16. The unchecked strip along the median

An efficient way to scan this strip is to sort all of its points into increasing y-order, so that the search needs only move up. The current point being examined (the one intersecting by dotted box in Fig. 10.17) only has to look at the points in a $d \times 2d$ region above it (the dotted area in the figure).

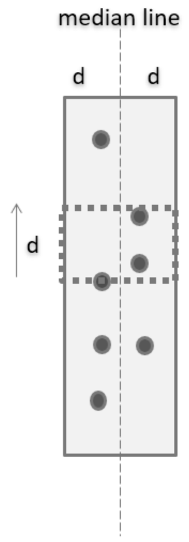

Figure 10.17. Scanning the strip

Searching this area takes a constant amount of time since there can only be a maximum of seven points in the region (see Fig. 10.18). The points cannot be more than a d distance apart, which means that there can be at most one point in each $d/2 \times d/2$ sub-square making up the $d \times 2d$ rectangle, and one of those points is currently being tested.

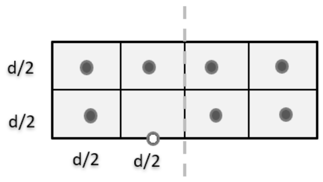

Figure 10.18. Points in the rectangle

The recurrence relation for the running time of the entire algorithm is:

$$T(n) = 2T(n/2) + O(7n)$$

The recursive calls to T() represent the recursive division of the space while the last term is the cost of examining all n point in the median strip. This relation simplifies to $O(n \log n)$.

The code for this algorithm is in Listing 10.6 (closestDC.py).

```
def closestPair(pts, n):
  if n <= 3:
    return pairsSearch(pts, n)

  mid = n//2
  midPoint = pts[mid]
  # divide and conquer
  dl, mpl = closestPair(pts, mid)  # left
  dr, mpr = closestPair(pts[mid:], n-mid) # right

  # use the minimum distance (and its pair)
  if dl < dr:
    d = dl
    mp = mpl
  else:
    d = dr
    mp = mpr

  strip = []
  # build a 2d-wide strip of points
  for i in range(n):
    if abs(pts[i].x - midPoint.x) < d:
      strip.append(pts[i])
  return checkStrip(strip, d, mp)


def checkStrip(strip, minDist, minPair):
  size = len(strip)

  # sort so can move upwards through the strip
  strip = sorted(strip, key=lambda pt: pt.y)

  for i in range(size):
    for j in range(i+1, size):
      # only examine minDist upwards
      if (strip[j].y - strip[i].y) >= minDist:
          break
      d = strip[i].distTo(strip[j])
      if d < minDist:
        minDist = d
        minPair = (strip[i], strip[j])

  return minDist, minPair
```

Listing 10.6. Finding the closest pair

One complication is that the function not only calculates the minimum distance between the points, but also returns the minimum points as a tuple. Another optimization is that when the number of points being considered is small (≤ 3) then the pairs are examined iteratively by pairsSearch().

The test-rig in closestDC.py generates 100 random points, calls closestDC() and uses matplotlib to plot the result, as shown in Fig. 10.19. The two ∘ points are the closest pair, and their coordinates are also printed to standard output.

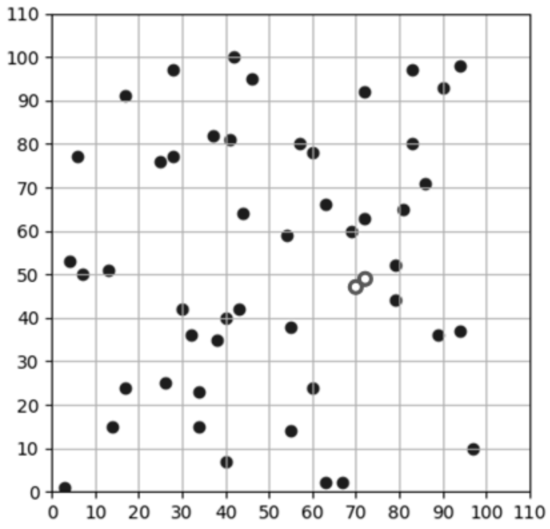

Figure 10.19. Finding the closest points by divide-and-conquer

Exercises

(1) Find an expression that relates the number of vertices of a regular polygon to the number of its diagonals.

(2) Given an arbitrary set of points, find the coordinates of the vertices of a rectangular box that just encloses it.

(3) A polygon can be formed from a sequence of line segments. One complication is that the result may not be *simple*, meaning that some of the shape's line segments cross. Write an algorithm which determines whether an n-vertex polygon is simple. The straightforward approach has running time $O(n^2)$; can you reduce this to $O(n \log n)$?

(4) Given a point $p_0 = (x_0, y_0)$, a right horizontal *ray* from p_0 is the set of points to the right of p_0: $\{p_i = (x_i, y_i) : x_i \geq x_0 \texttt{ and } y_i = y_0\}$. Show how to determine whether a given right horizontal ray from p_0 intersects a line segment $\overline{rs}$ in constant time. *Hint*: reduce the problem to one of determining if two segments intersect.

(5) One way to determine whether a point p_0 is inside a polygon P is to generate a ray from p_0 and check that it intersects the edge of P an *odd* number of times. Show how to compute this in linear time. *Warning*: make sure your algorithm is correct when the ray intersects the polygon boundary at a vertex and when the ray overlaps a side of the polygon.

(6) *Picking up sticks*. There are n sticks piled up in some random order. Each stick is specified by its endpoints defined as (x, y, z) coordinates. You can assume that no stick is standing vertical. Your task is to pick up all the sticks, one at a time, subject to the condition that you can only pick up a stick if there isn't another stick on top of it.

a) Write code that takes two sticks a and b and reports whether a is above, below, or unrelated to b.

b) Describe an algorithm that determines whether it is possible to pick up all the sticks, and if so, provides a legal order for achieving this.

(7) Let S be a set of n circles in the plane. Describe a sweep algorithm to compute all of the intersection points between the circles. (Note that two circles do not intersect if one lies entirely inside the other.)