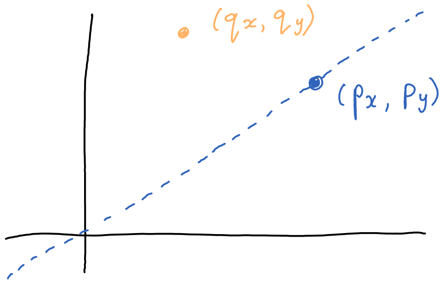
7 Geometrical algorithms

7.1 Segment intersection

Do two line segments intersect? This is a simple question, and a good starting point for many more interesting questions in computational geometry.

Let's start with a simpler problem. Is the point *q* above or below the dotted line? The answer doesn't need anything more than basic school maths.



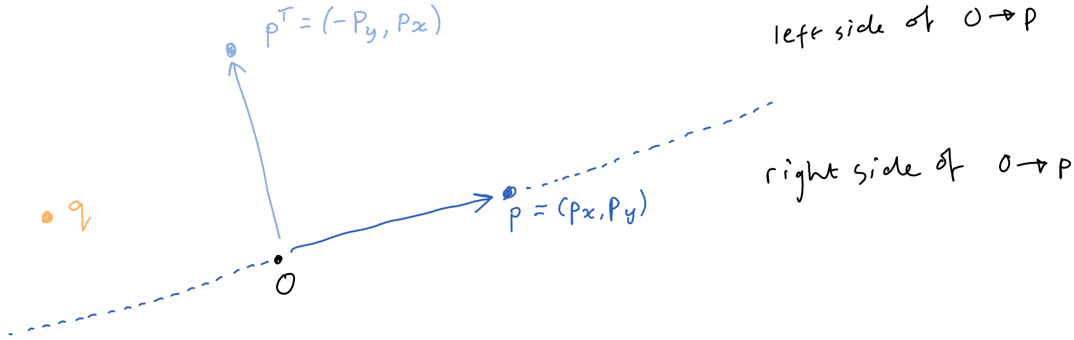
If $\quad q_y > \frac{p_y}{p_x} q_x \quad$ ie $\quad p_x q_y - p_y q_x > 0$: above

If $\quad q_y < \frac{p_y}{p_x} q_x \quad$ ie $\quad p_x q_y - p_y q_x < 0$: below

We should really write out equations for all cases (which quadrant is *p* in? which quadrant is *q* in?), and we quickly get tangled up. Or, we could use slightly cleverer maths, namely dot products, to get to a cleaner answer:

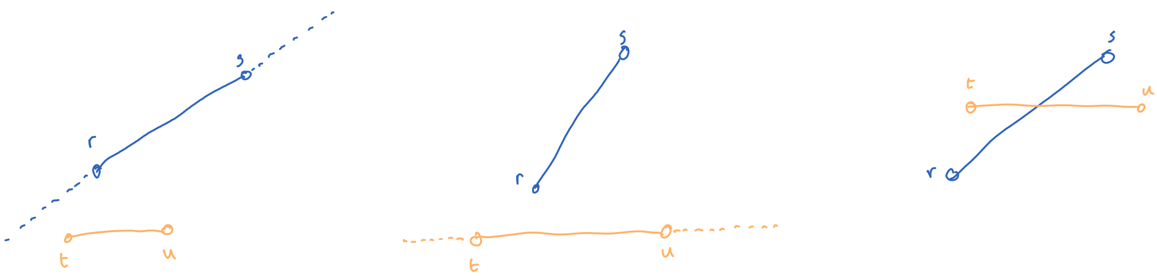
Let $p^T=(-p_y,p_x)$. If we rotate the vector $0\to p$ by 90° anticlockwise, we get $0\to p^T$. Now the sign of $p^T\cdot q$, i.e. of $-p_y q_x+p_x q_y$, tells us which side of the dotted line *q* is on. The dotted line is called the extension of $0\to p$.

- $p^T\cdot q > 0$: *q* is on the left, as you travel along the dotted line in direction $0\to p$
- $p^T\cdot q = 0$: *q* is on the line itself
- $p^T\cdot q < 0$: *q* is on the right



This gives us all the tests we need to decide if two line segments *r*–*s* and *t*–*u* intersect.

1. If *t* and *u* are both on the same side of the extension of $r\to s$, i.e. if $(s-r)^T\cdot(t-r)$ and $(s-r)^T\cdot(u-r)$ have the same sign, then the two line segments don't intersect.
2. Otherwise, if *r* and *s* are both on the same side of the extension of $t\to u$, then the two line segments don't intersect.
3. Otherwise, they do intersect.



Well-written code should test all the boundary cases, e.g. when $r=s$ or when t or u lie on the the extension of $r \to s$. It is however a venial sin to test equality of floating point numbers, because of the vagaries of finite-precision arithmetic, and so the question "How should my segment-intersection code deal with boundary cases?" depends on "What do I know about my dataset and what will my segment-intersection code be used for?" The example sheet asks you to consider a case in detail.

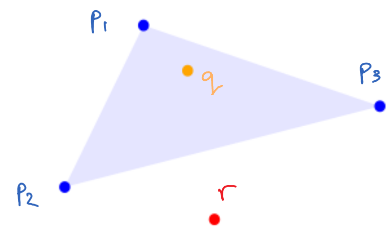
7.2 Jarvis's march

Given a collection of points $p_1, \ldots, p_n$, a **convex combination** is any vector

$q = \alpha_1 p_1 + \ldots + \alpha_n p_n$

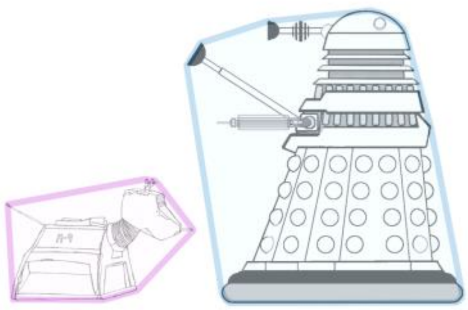
where the coefficients α_i are all ≥ 0 and sum to 1. The **convex hull** of a collection of points is the set of all convex combinations.

Convex hulls are used for example for collision detection: first test whether the convex hulls of two objects collide, and if they do then run an exact but slower test of whether the objects themselves collide.



$q = 0.6\, p_1 + 0.16\, p_2 + 0.24\, p_3$ is a convex combination

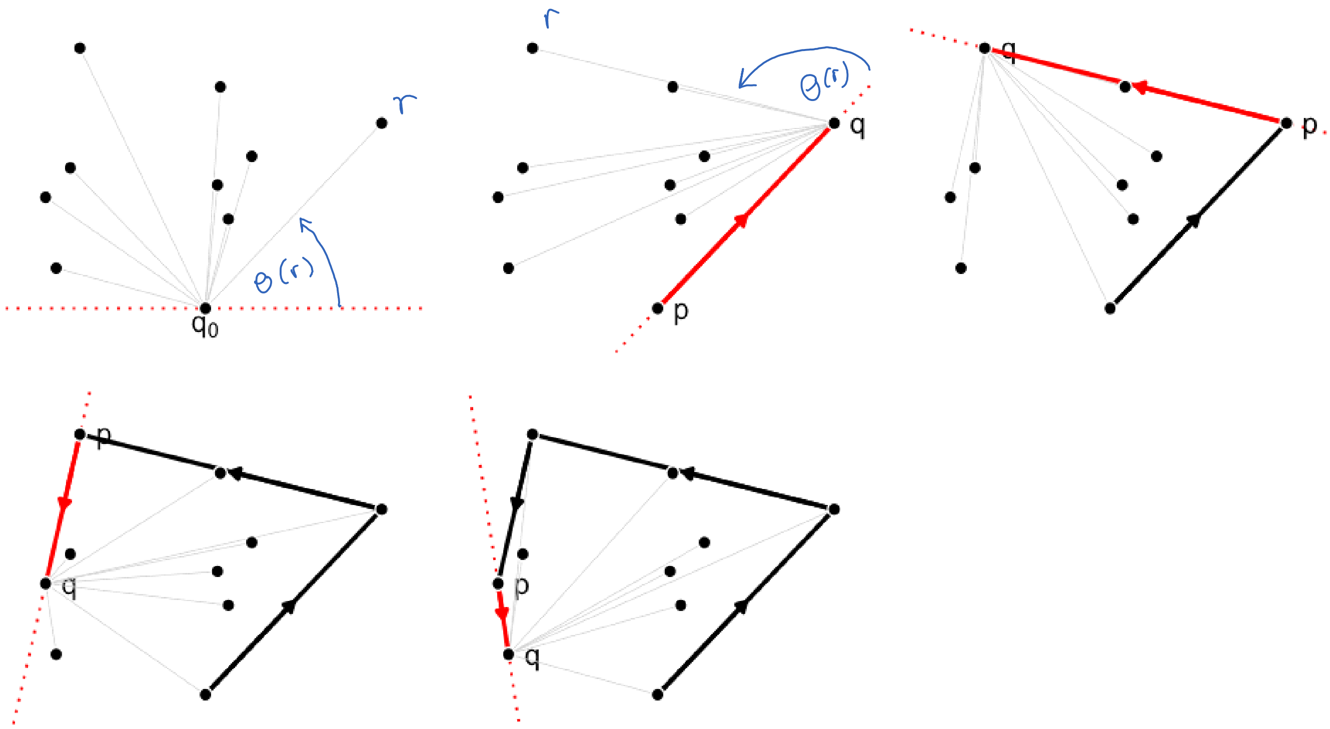
$r = -0.5\, p_1 + 0.9\, p_2 + 0.6\, p_3$ is not a convex combination

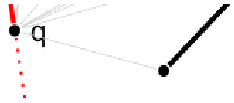


Here is an algorithm to compute the convex hull of a collection of points P. It is due to Jarvis (1973), and discovered independently by Chand and Kapur (1970). (Formally, this algorithm finds the corner points of the convex hull, i.e. the points $p \in P$ such that $p \notin \text{convexhull}(P \setminus \{p\})$. But in this part of the course we shall use intuition rather than definitions.)

Algorithm

The algorithm builds up a list of points iteratively. Given the points that have been added so far, draw a line between the last two points that were added (call them p and q). Then for every other point $r \in P$ find the angle $\theta(r)$ that $q \to r$ makes with the extended $p \to q$ line; pick the point with the smallest angle and add it to the list. If two points have the same smallest angle, use the one that's furthest from p. Stop when we return to the start point.



How do we get the first two points to start the iteration? Start with the point q_0 with the lowest y-coordinate, and if there are several then pick the one that has the largest x-coordinate. This is guaranteed to be a corner point of the convex hull. For the second point, use the same minimum-angle method as above, but measuring angles with respect to a horizontal (left→right) reference line.

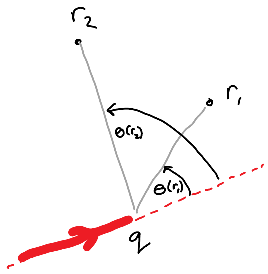
This is very much like selection sort: repeatedly find the item that goes in the next slot. In fact, most convex hull algorithms resemble some sorting algorithm.

Performance note

The algorithm involves "find the point r with the smallest angle $\theta(r)$". We could use trigonometry to compute θ—but there is a trick to make this faster. (Faster in the sense of "games get more frames per second", but no difference in the big-O sense.) If all the points we're comparing are on the same side of dotted line, as they are at all steps of Jarvis's march, then

$\theta(r_1)$ is smaller than $\theta(r_2)$ $\iff$ r_2 is on the left of the extended line $q \to r_1$

and we've seen how to compute this true/false value with just some multiplications and additions.



Analysis

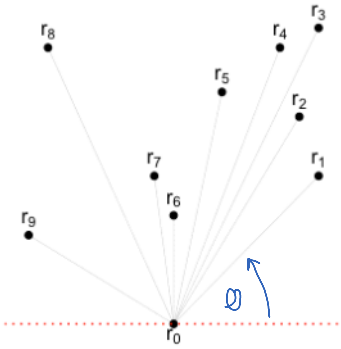
At each step of the iteration, we search for the point $r \in P$ with the smallest angle $\theta(r)$, thus the algorithm takes $O(n\ h)$ where n is the number of points in P and h is the number of points in the convex hull. (As wtih Ford-Fulkerson, running time depends on the content of the data, not just the size.)

7.3 Graham's scan

Here is another algorithm for computing the convex hull, due to Ronald Graham (1972).

Algorithm

Find the point r_0 with the lowest y-coordinate, and if there are several then pick the one that has the largest x-coordinate. This is guaranteed to be a corner point of the convex hull. Next, sort all other points r by the angle that $r_0 \to r$ makes with the horizontal (left→right) line, lowest angle to highest. Call them $r_1 \ldots r_{n-1}$.

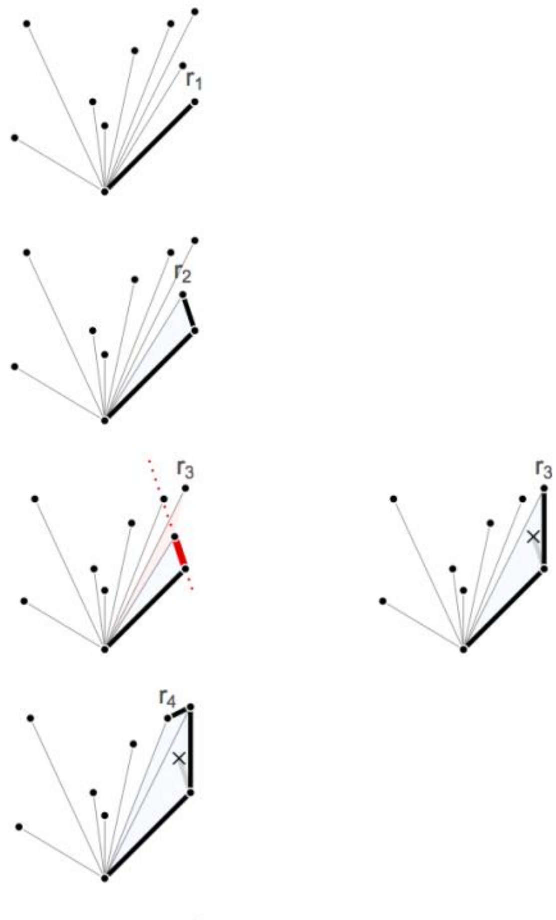


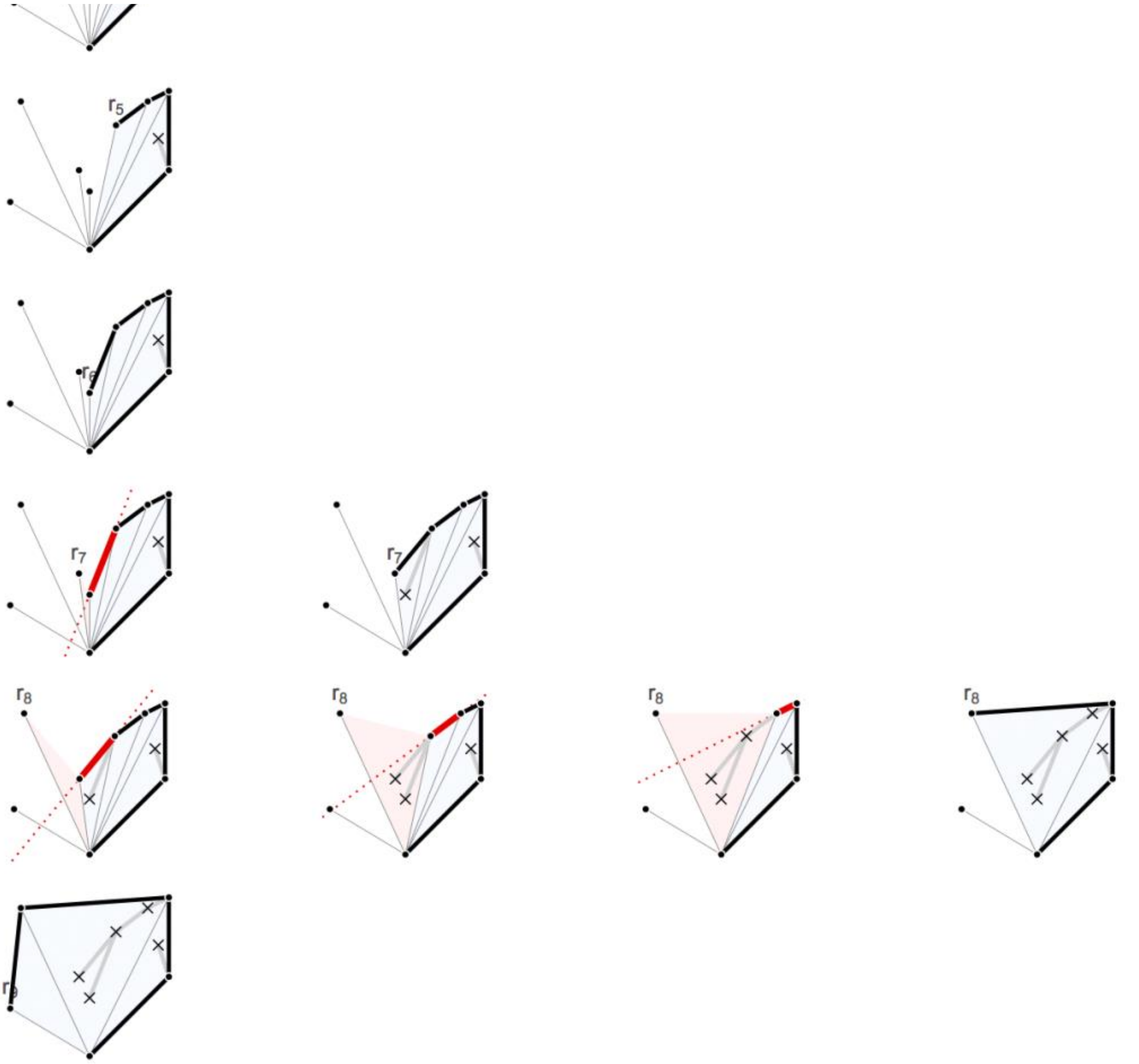
Then, build up a list of points by adding in $r_1 \ldots r_{n-1}$ in order, and backtracking when necessary:

```
h = [r₀, r₁]
for each rᵢ in the sorted list of points, i≥2:
    if rᵢ isn't on the left of the extension of the final segment of h:
        repeatedly delete points from the end of h until this is no longer the case
    append rᵢ to h
```

This code doesn't deal correctly with some boundary cases. Question 5 on the example sheet asks you to spot the problems.

Here is how the algorithm proceeds. The plots show it iterating over the r_i (one row per step of the iteration) and backtracking (side-by-side plots show steps in backtracking).





Performance note

The algorithm involves "sort points r by the angle of the $r_0 \to r$ line". We don't actually need to compute angles in order to sort by angle: all that a sorting algorithm needs is a way to test "does p have a smaller angle than q?", and the trick from Section 7.2 will work here.

Analysis

The initial sort takes time $O(n \log n)$. Every point r_i is added to the list once, and it can be removed at most once, so the loop is $O(n)$.