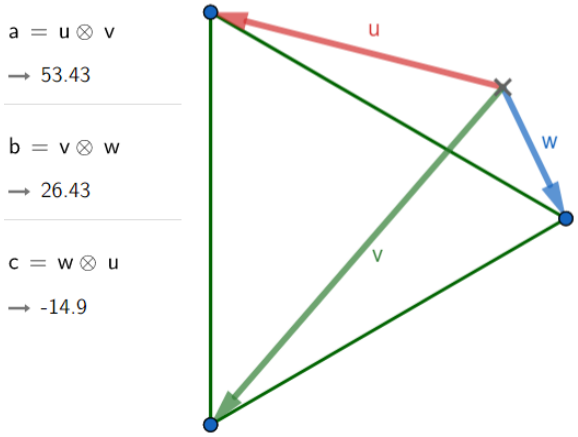
An efficient way to test if a point is in a given convex polygon

This algorithm is designed to facilitate rapid checking, eg by a computer, to determine if a given location, defined by coordinates (x, y), lies within a given convex polygon, defined by a series of similar coordinates, listed in order around the shape (in either direction).
Note also that if all values involved are integers (as if often the case when representing elements on a display: pixels from the left, pixels from the top), this algorithm requires nothing more than integer operations. Expensive operations like trigonometry are bypassed entirely.

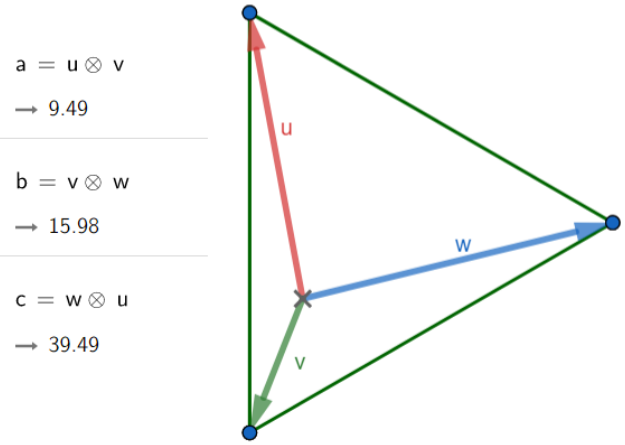
The key idea
If a point lies outside a given polygon, we can construct vectors from it to each point in turn. Consider the angle (and direction of turn) between one vector and the next, including from the last back around to the first:

Case 1: Point outside polygon



The angle from $\boldsymbol{u}$ to $\boldsymbol{v}$ is less than π, and in the positive (anti-clockwise) direction.
Likewise for $\boldsymbol{v}$ to $\boldsymbol{w}$.
However, for $\boldsymbol{w}$ to $\boldsymbol{u}$, the anti-clockwise angle is more than π. Or, equivalently, the smallest angle of turn requires a clockwise rotation.

Case 2: Point within polygon:



The angle from $\boldsymbol{u}$ to $\boldsymbol{v}$ is between 0 and π, in the anti-clockwise direction.
The same is true for $\boldsymbol{v}$ to $\boldsymbol{w}$.
And – crucially – also true for $\boldsymbol{w}$ to $\boldsymbol{u}$.

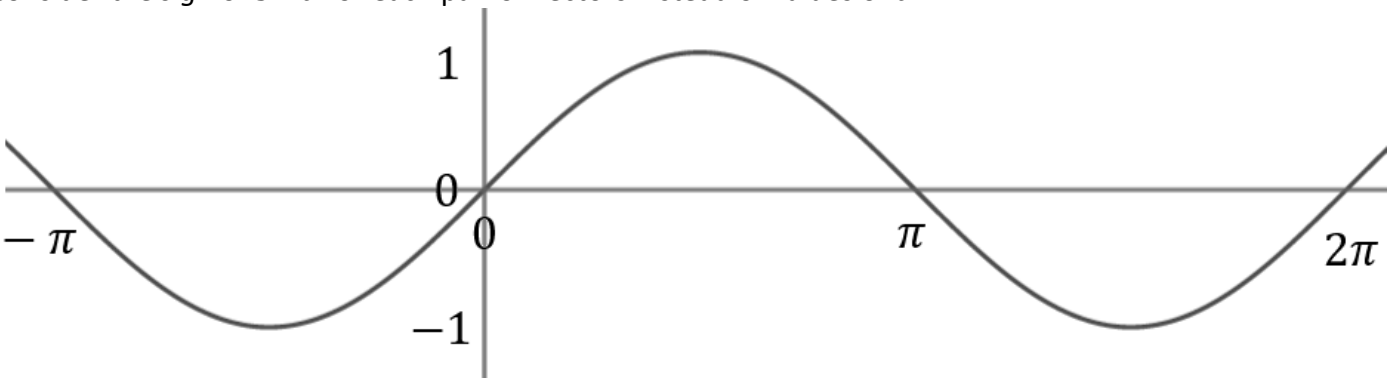
Essentially, when we 'look' at each vertex in turn from a vantage point outside the polygon, we turn first one way and then the other, whereas if we look from a point within the polygon, we turn all the way around, maintaining the same direction of turn throughout.

Other cases:
If the point we are testing lies *exactly* on the boundary of the polygon, the angle of turn will be exactly π (clockwise or anti-clockwise). Depending on whether we use $<$ or $\leq$ in our test, we can choose how strict or inclusive we want to be with our definition of 'inside'.

The illustrations above assume that the order in which vertices are traversed around the polygon is clockwise. If the order is reversed, the direction of turn is also reversed, but for every single turn, so the same distinction occurs: a change of direction if and only if the point in question lies outside the polygon.

A useful insight

Calculating the angle between vectors is computationally expensive. But we don't actually care about the exact angles. In fact, all we want to know in each case is whether they are within the range 0 to π or not. Since $\sin\theta$ gives positive values for angles in the range 0 to π, and negative values otherwise, we could consider the *sign* of $\sin\theta$ for each pair of vectors instead of values of θ.



Notice that there is no distinction between an anticlockwise reflex angle and the equivalent clockwise non-reflex angle: in each case, $\sin\theta$ will be the same (and, importantly, negative).

Sign instead of sine

One further observation is that we aren't interested even in the value of $\sin\theta$, only its *sign* (positive or negative). If there is a *change of sign of sine* at any point as we traverse the vectors, that indicates the point is outside the polygon, and if not, it's inside.

The cross product

There are equally valid geometric justifications for the following formula, but the quickest way to derive it is via the vector cross product. For two vectors $\boldsymbol{u}$ and $\boldsymbol{v}$:

$$\boldsymbol{u} \times \boldsymbol{v} = |\boldsymbol{u}||\boldsymbol{v}| \sin\theta\, \hat{\boldsymbol{n}}$$

Where $\hat{\boldsymbol{n}}$ is a unit normal vector perpendicular to both $\boldsymbol{u}$ and $\boldsymbol{v}$, oriented according to the right-hand rule (when turning anticlockwise from $\boldsymbol{u}$ to $\boldsymbol{v}$ as in the previous illustrations, this means directly up out of the page, or in the positive k direction).

Ignoring the normal

We can safely ignore $\hat{\boldsymbol{n}}$ since we will either be consistently turning anticlockwise or consistently clockwise as we traverse the points of our polygon, depending on how they are defined. Recall that we can treat a turn in the opposite direction simply as a turn in the same direction but by a reflex angle. Notice:

$$\sin\left(\frac{\pi}{6}\right)\begin{pmatrix}0\\0\\-1\end{pmatrix} = \sin\left(\frac{11\pi}{6}\right)\begin{pmatrix}0\\0\\1\end{pmatrix} = \begin{pmatrix}0\\0\\-0.5\end{pmatrix}$$

The big efficiency saver

The huge computational efficiency here comes from the fact that $\sin\theta$ can be calculated using cross product methods, since:

$$\sin\theta\, \hat{\boldsymbol{n}} = \frac{\boldsymbol{u} \times \boldsymbol{v}}{|\boldsymbol{u}||\boldsymbol{v}|}$$

If we only care about the *sign* of $\sin\theta$, we can safely dispense with calculating the size of the two vectors involved, and just find the *sign* of $\boldsymbol{u} \times \boldsymbol{v}$.

Therefore we compute $\boldsymbol{u} \times \boldsymbol{v}$ using the following formula:

$$\boldsymbol{u} = \begin{pmatrix} u_x \\ u_y \\ u_z \end{pmatrix} \quad and \quad \boldsymbol{v} = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} \implies \boldsymbol{u} \times \boldsymbol{v} = \begin{pmatrix} u_y v_z - v_y u_z \\ -(u_x v_z - v_x u_z) \\ u_x v_y - v_x u_y \end{pmatrix}$$

Only two dimensions required

This formula gets nicer still when we recall that, while technically the cross product is defined for 3D vectors, our polygon is situated in 2D space, making the z components zero:

$$\boldsymbol{u} \times \boldsymbol{v} = \begin{pmatrix} 0 \\ 0 \\ u_x v_y - v_x u_y \end{pmatrix}$$

The implications of ignoring $\hat{\boldsymbol{n}}$

We now have:

$$\sin\theta\, \hat{\boldsymbol{n}} = \begin{pmatrix} 0 \\ 0 \\ u_x v_y - v_x u_y \end{pmatrix}$$

But $\hat{\boldsymbol{n}}$ is either always $\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$ if we're traversing the polygon anti-clockwise, or always $\begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix}$ if we're going around it clockwise. Since all we care about is a *change of sign*, it really makes no difference if we flip *all* the signs: if they're all the same, they'll still be all the same, and if some are different, they'll still be different.

Therefore the quantity we will use to compare one vector with the next is just this:

$$valueWeWantTheSignOf = u_x v_y - v_x u_y$$

Since we don't even care about this value, just its sign, our test boils down to the following:

$$Is \quad u_x v_y - v_x u_y > 0?$$

Or, equivalently:

$$Is \quad u_x v_y > v_x u_y \ ?$$

A sketch of the algorithm

The function we're trying to construct would ideally take as its input the coordinates of the test point and, in order around the polygon (either clockwise or anticlockwise – it doesn't matter), the coordinates of the vertices. It would return a simple 'true' or 'false'.

Step 1:

Construct vectors from the test point to each vertex. Eg for a 4 sided shape there would be 4 vectors.

Step 2:

Construct an array of values which correspond to the simplified cross product value given above, by comparing each vector in our list with the next one (and, when we reach the end, comparing the last vector to the first). So for a 4 sided shape, there would be a value corresponding to the angle between vectors 0 and 1, 1 and 2, 2 and 3 and finally 3 and 0, giving 4 values in total. We could use the modulo operator to loop back around, or if it's easier, append a copy of the first element to the end of the vectors array initially.

Step 3:

Loop through each value in the array above, comparing it to the next value in the list. Note that for a list of 4 values, there will be 3 comparisons (compare values 0 and 1, 1 and 2, and finally 2 and 3). Each comparison will check to see if there has been a change of sign. Probably the simplest way is to multiply the values and see if the product is negative.

Step 4:

If a change of sign is detected, the loop can terminate early with a 'return false' result.
Otherwise, if the loop completes without terminating early, it means no sign change was detected, and we can 'return true'.

Here's an example of how this might look in the p5.js JavaScript coding environment:

```
function pointInPolygon(pointVector, vertexVectors){
  let vectors = [];
  // vectors from point to each vertex
  for (let i=0; i<vertexVectors.length; i++){
    vectors.push(p5.Vector.sub(vertexVectors[i], pointVector));
  }
  let values = [];
  // sign of each value corresponds to direction of turn from one vector to the next
  for (let i=0; i<vectors.length; i++){
    let u = vectors[i];
    let v = vectors[(i+1)%vectors.length];
    // compare each vector to the next, including the final vector to the first
    values.push(u.x*v.y-v.x*u.y);
  }
  for (let i=0; i<values.length - 1; i++){
    if (values[i] * values[i+1] < 0){
    // sign change indicates direction change, hence point outside polygon
      return false
    }
  }
  return true;
}
```