Two Dimensional Geometry Calculator

Author: Robert L. Walton <walton@acm.org>
Date: Wed Aug 4 22:08:37 EDT 2021

1 Overview

This document describes a group of 2-dimensional geometry problems involving vectors, points, and straight lines (but not polygons). The problems are embedded in a simple calculator for ease of testing.

2 Boolean, Scalars, Vectors, and Transforms

A **boolean** here is either the symbol **true** or the symbol **false**.

A **scalar** here is a double precision floating point number.

A **vector** here is a pair (x, y) of double precision floating point numbers, x denoting an X-coordinate value, and y denoting a Y-coordinate value. Such a vector can denote a point (p_x, p_y) in the XY-plane, or can denote a translation (v_x, v_y) of the plane:

$$(p_x, p_y) \longmapsto (p_x + v_x, p_y + v_y)$$

The product of a scalar s and a vector $v = (v_x, v_y)$ is represented using $*$ as $s * v = (sv_x, sv_y)$. The sum of a vector $v = (v_x, v_y)$ and a vector $w = (w_x, w_y)$ is represented using $+$ as $v + w = (v_x + w_x, v_y + w_y)$.

A **linear transformation** here is a pair of vectors, $[\ell_x, \ell_y]$ representing the map:

$$(v_x, v_y) \longmapsto v_x * \ell_x + v_y * \ell_y$$

Application is represented using $*$ as $[\ell_x, \ell_y] * v = v_x * \ell_x + v_y * \ell_y$.

A **vector list** here is a list of vectors $(v_1, v_2, \ldots)$. This can be used to represent the list of vertices of a polygon or a list of points in the plot of a function.

3 The Calculator Language

The calculator language of this problem is very simple. An example is:

Input	**Output**
`x=4`	`x=4`
`y = 0.5`	`y = 0.5`
`z = x + y`	`z = x + y = 4.5`
`v=(3,-2)`	`v=(3,-2)`
`w=z*v`	`w=z*v = (13.5,-9)`
`# this is a comment`	`# this is a comment`

The syntax is:

program ::= *statement**
statement ::=
    comment eol
  | *variable* **=** *value eol*
  | *variable* **=** *operator* { *variable operator* }* *variable*$^?$ *eol*
  | *variable* **=** *variable* { *operator variable* }* *operator*$^?$ *eol*
  | *variable* **=** *function variable** *eol*
eol ::= end-of-line (line-feed)
comment ::= **#** *character-other-than-line-feed**
variable ::= a single letter (case matters)
value ::= *boolean* | *scalar* | *vector* | *linear-transform* | *vector-list*
boolean ::= **`true`** | **`false`**
scalar ::= double precision floating point number (e.g., 1.4142135)
vector ::= **`(`** *scalar*`,` *scalar* **`)`**
linear-transform ::= **`[`** *vector*`,` *vector* **`]`**
vector-list ::= **`()`** | **`(`** *vector* { `,` *vector* }* **`)`**
operator ::= one or more non-letter, non-digit graphic characters
    (e.g. + or <=)
function ::= three or more letters (e.g. `sin` or `area`)
    no *function* shall be a prefix of another *function*

There are multiple versions of the calculator, each an extension of previous versions:

Version	Implements
Scalar	scalar and boolean types
Vector	the vector type
Linear Transform	the linear transformation type
Rotations and Reflections	rotation and reflection linear transforms
Products	vector scalar and cross products
Line and Point	algorithms involving a line and a point
Line and Line	algorithms involving two lines

The Input/Output Rules are:

- Whitespace in *statements* is optional and should be deleted before the *statements* are parsed. There may be whitespace before the '`#`' that begins a comment.
- The output of a *statement* is a copy of the *statement* line (including whitespace) if the *statement* is a comment or is of the form '*variable* **=** *value eol*'. Otherwise the output is one line containing a copy of the *statement* followed by '*space* **=** *space value eol*', where the *value* is that assigned to the *variable* beginning the *statement*.
- Input lines have a maximum of 100,000 characters (long lines are only needed for inputting vector lists). Output lines have no limit (output vector lists can be arbitrarily long).
- Each calculator version should be able to input and output values of the types being tested by that version (beginning with *boolean* and *scalar* values) and process comments and statements of the form '*variable* **=** *variable*'.
- *Scalars* should be output with 6 digits of precision (which is the default for C/C++). To pass tests, scalar values must be accurate to better than 5 digits of precision or to better than $\pm 10^{-10}$.
- The total number of vector list elements that need to be allocated during one program execution will be small enough that these list elements do not need to be garbage collected (need not be freed when no longer needed).
- Input will not contain the character '`$`', which can then be used as per Coding Hints below.

4 Scalar Calculator

Implement the calculator with just *scalar* and *boolean* value types and the *operators* and *functions*:

Assume:	`x`=x, `y`=y are scalars
	`d`=d is an integer scalar, $-15 \leq d \leq +15$
then:	
`x+y`	returns $x + y$
`x-y`	returns $x - y$
`x*y`	returns $x \times y$
`x/y`	returns x/y
`x%y`	returns $x - n * y$
	where $n = x/y$ rounded to the integer nearest zero
`-x`	returns $-x$
`|x|`	returns $\|x\|$, the absolute value of x
`cos x`	returns $\cos((\pi/180) * x)$ [`x` is in degrees]
`sin x`	returns $\sin((\pi/180) * x)$ [`x` is in degrees]
`tan x`	returns $\tan((\pi/180) * x)$ [`x` is in degrees]
`x<y:d`	returns **`true`** if $x < y - 0.5 * 10^{-d}$, else **`false`**
`x<=y:d`	returns **`true`** if $x < y + 0.5 * 10^{-d}$, else **`false`**
`x==y:d`	returns **`true`** if $\|x - y\| < 0.5 * 10^{-d}$, else **`false`**
`x:d`	returns x rounded to be an exact multiple of 10^{-d}

Angles are measured in degrees, with positive angles going counter-clockwise and negative angles going clockwise.

Comparisons assume the scalars being compared are approximations to precise numbers that are in units of 10^{-d}, and therefore to test for equality the scalars should be (in effect) rounded to the nearest unit. In addition, the scalars have to be good enough approximations. As double precision floating point numbers are accurate to at most one part in 10^{16}, this means that if the units are 10^{-d} the scalar should not have absolute value above 10^{15-d}.

Division by `0` will produce results such as `inf` (plus infinity), `-inf` (minus infinity), and `nan` (not a number; i.e., result could not be computed).

Sample Input Files:	**00-XXXX-scalar-vec2d.in**
Sample Output Files:	**00-XXXX-scalar-vec2d.ftest**
Sample Run File:	**sample-scalar-vec2d.run**
Submit Run File:	**submit-scalar-vec2d.run**

You can test your program using the indicated sample input and output and you can submit your program using the indicated submit run file. Only scalar functionality is tested.

There is also the file:

Special Input File: **00-000-special-vec2d.in**

which may be used to see the results of divide by zero and other anomalous scalar computations. The output of your program for this special input file will depend upon the programming language you use, so the output is not testable except by human inspection. Therefore there is no companion `.ftest` file, and the special input file is not included in any `.run` files.

5 Coding Hints

The calculator memory just maps each variable to a value. You can map each ASCII character to a value using a vector of 128 elements, and simply not use the elements that do not correspond to letters.

To parse a line, remove its whitespace characters. Then

- If the line begins with '`#`', it is a *comment*.
- Else if the line contains a digit, it has the form '*variable* **=** *value*', where *value* is:
  - A linear-transform if the '**=**' is followed by '**[**'.
  - A vector list if the '**=**' is followed by '**((**'.
  - A vector if the '**=**' is followed by '**(**' but not '**((**'.
  - A scalar otherwise.
- If the line does not contain a digit, match the line to a pattern using the rules:

- The pattern and the line must be the same length.
- A pattern character other than '**$**' must match the line character exactly.
- The '**$**' pattern character must match a letter (which will be a *variable*).

Examples:

The Pattern	Matches	
`$=$`	`x=y`	
`$=true`	`t=true`	
`$=()`	`x=()`	
`$=$+$`	`b=c+d`	
`$=sin$`	`x=sinA`	[Normally input as `x=sin A`]
`$=$<=$:$`	`x=y<=z:d`	[Normally input as `x = y<=z:d`]

Remember that when you output a line, you are outputting the original line before whitespace characters were removed.

Solutions to the Scalar Calculator problem are available in the files:

`c-scalar-vec2d.txt`	for C
`cc-scalar-vec2d.txt`	for C++
`java-scalar-vec2d.txt`	for JAVA
`py-scalar-vec2d.txt`	for PYTHON

You can build solutions for successive calculator versions by renaming and extending one of these files, if you like.

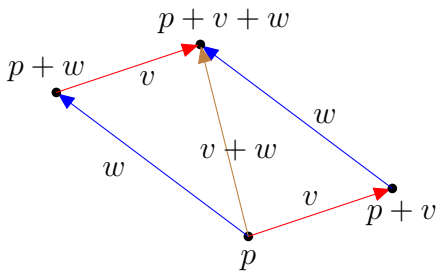
The first section of each of these files contains data declarations. Using these should make it easier for other people familiar with this problem to read your code, should that be important. It is recommended that you use this first section even if you do not use the rest of the file.

6 Operations on Vectors

We will define the **vector sum** of vectors $v = (v_x, v_y)$ and $w = (w_x, w_y)$ to be $v + w = (v_x + w_x, v_y + w_y)$. If you think of a point $p = (p_x, p_y)$ in the XY-plane as a vector, you can associate a translation τ_v of the XY-plane with the vector v using the equation $\tau_v(p) = p + v$. Then

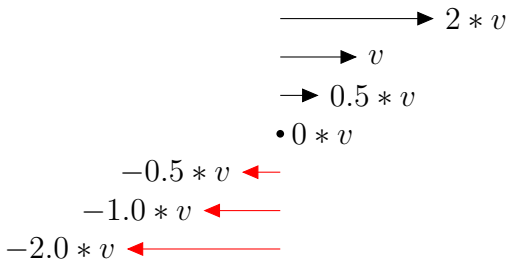
$$\tau_w(\tau_v(p)) = p + v + w = \tau_{v+w}(p)$$

See the picture.



Vector sums are commutative, i.e., $v + w = w + v$, and associative, i.e., $u + (v + w) = (u + v) + w$.

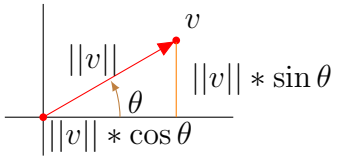
We define the **scalar product** of a scalar s and a vector $v = (v_x, v_y)$ as $s * v = (s * v_x, s * v_y)$. Multiplication of v by $s > 0$ does not change the direction of v, but multiplies the length of v by s. Multiplication of v by $s < 0$ reverses the direction of v and multiplies its length by $|s|$. See the picture.



Scalar products are associative in that $s_1 * (s_2 * v) = (s_1 * s_2) * v$ and distributive in that $s * (v_1 + v_2) = s * v_1 + s * v_2$ and $(s_1 + s_2) * v = s_1 * v + s_2 * v$.

The negative of a vector is defined as $-v = (-1) * v = (-v_x, -v_y)$. Subtraction of vectors is defined as $v - w = v + (-w) = (v_x - w_x, v_y - w_y)$.

A vector $v = (v_x, v_y)$ has a length and a direction. The **length**, denoted by $||v||$, can be computed using Pythagoras's Theorem: $||v|| = \sqrt{v_x^2 + v_y^2}$. The direction is given by the counterclockwise angle θ from the positive X-axis direction to the vector direction. This is called the **azimuth** of v, and in computer languages may be computed (in radians) by the `atan2` function as $\theta = \text{atan2}(v_y, v_x)$ (note v_y comes before v_x).



We measure angles, including azimuths, in degrees. To convert degrees to radians, multiply by $\pi/180$, and to convert radians to degrees, multiply by $180/\pi$. Positive angles are counter-clockwise, and negative angles are clockwise.

For a vector `v`, `azm v` and `||v||` are said to be the **polar coordinates** of `v`. The vector with the polar coordinates θ (azimuth) and ℓ (length) is $(\ell \cos\theta, \ell \sin\theta)$. For the vector `v`=$(v_x, v_y)$:

$$\texttt{||v||} = \sqrt{v_x^2 + v_y^2}$$
$$\texttt{azm v} = (180/\pi) \texttt{ * atan2(}v_y\texttt{,}v_x\texttt{)}$$

See the picture.

Adding integer multiples of 360 to a vector's azimuth does not change the vector.

7 Vector Calculator

Implement additions to the scalar calculator for just *vector* value types and the *operators* and *functions*:

Assume:	`s`=s, `x`=x, `y`=y, `l`=ℓ, `t`=$(180/\pi) * \theta$ are scalars `d`=d is an integer scalar, $-15 \le d \le +15$ `v`=(v_x, v_y), `w`=(w_x, w_y) are vectors `L` is a list of vectors	
then:		
`(x,y)`	returns (x, y)	
`v.x`	returns v_x (`x` is not a variable here)	
`v.y`	returns v_y (`y` is not a variable here)	
`v==w:d`	returns $v_x == w_x : d$ AND $v_y == w_y : d$	
`v:d`	returns $(v_x : d, v_y : d)$	
`s*v`	returns (sv_x, sv_y)	
`-v`	returns $(-v_x, -v_y)$, `v` with direction reversed	
`v+w`	returns $(v_x + w_x, v_y + w_y)$	
`v-w`	returns $(v_x - w_x, v_y - w_y)$	
`||v||`	returns $\sqrt{v_x^2 + v_y^2}$	[length]
`azm v`	returns $(180/\pi) * \text{atan2}(v_y, v_x)$ adjusted to be in the range [0,360). Calculated values very near 0 or 360 may be unstable. The returned value is undefined if $\|\|v\|\| = 0$.	[azimuth]
`l^t`	returns $(\ell \cos\theta, \ell \sin\theta)$, the vector with length `l` and azimuth `t` degrees, where $\theta = (\pi/180) * \texttt{t}$ radians	
`cons v L`	returns the list made by prepending `v` to `L`	

If `||v|| = 0` then `azm v` is undefined (you can do extra programming to make it defined as `nan` if you like, but this is not necessary).

Sample Input Files:	**00-XXXX-vector-vec2d.in**
Sample Output Files:	**00-XXXX-vector-vec2d.ftest**
Sample Run File:	**sample-vector-vec2d.run**
Submit Run File:	**submit-vector-vec2d.run**

You can test your program using the indicated sample input and output and you can submit your program using the indicated submit run file.

8 Display Language

Points, lines, arrows, etc. can be displayed by making a `.pdf` file from an `.in` file and your solution program. The display commands are embedded in `.in` file '`#`' comment lines, so your program does not have to deal with them. The `00-000-vector-vec2d.in` file has examples of all the display commands. Most sample `00-XXXX-YYYY-vec2d.in` files have have display commands if they compute vectors.

The line/area drawing commands are:

Command	Draws
`#!point p [color] [size]`	point `p`
`#!point L [color] [size]`	points in the point list `L`
`#!line pq [color] [opt]`	line from point `p` to point `q`
`#!line L [color] [opt]`	lines with endpoints that are consecutive points in the list of points `L`
`#!infline pA [color] [opt]`	infinite line from point `p` in direction `A` (a scalar angle)
`#!rectangle pq [color] [opt]`	rectangle with opposing corners point `p` and point `q`
`#!circle cr [color] [opt]`	circle of center `c` and radius `r`
`#!ellipse cRA [color] [opt]`	ellipse with center `c`, radii `R.x` and `R.y`, rotated by angle A about its center

In the above `p`, `q`, `c`, `R`, `r`, `A`, `L` are arbitrary *variables*. Each variable used in a display command must have been set or computed by a calculator statement output before the display command. The last value assigned to the variable by a calculator statement before the display command will be used by the display command.

Circle and ellipse radii must be strictly positive. Spaces are as indicated, and must consist of at least one horizontal space character. Coordinates are automatically scaled to fit the logical page. Angles are in degrees.

The possible colors are '`red`', '`blue`', '`brown`', and '`black`'. Black is the default. (Green does not show up when the page is printed in black-and-white, and so is not provided.)

For points, `size` is an integer in the range [1,18] giving the size of each point in units of 1/72'nd inch (1pt). The default is 2pt.

For non-points `opt` is zero or more of:

- `.` dotted line
- `-` dashed line
  (`.` and `-` conflict; if neither line is solid)
- `c` close path (for `#!line L`)
  (implied by `s`, `d`, `h`, `v`)
- `m` add arrow head in middle of each line segment (for `#!line`)
- `e` add arrow head at end of each line segment (for `#!line`)
- `s` fill with solid color
- `d` fill with dots
- `h` fill with horizontal bars
- `v` fill with vertical bars
  (`s`, `d`, `h`, and `v` conflict)
- `f` extend infinite line forward from point (for `#!infline`)
- `b` extend infinite line backward from point (for `#!infline`)
  (`f`, `b` conflict; if neither line extends in both directions)

The text drawing commands are:

Command	Draws
`#!text p [color] opt text...`	the text, place at/near point `p`
`#!text pq [color] opt text...`	ditto but the text is placed at/near the midpoint of the line `pq`

Here `opt` is '`-`' for no options, or given that the point of text placement is `(`x`,`y`)`, is one or more of:

- `t` display about 0.5em above y
- `b` display about 0.5em below y
  (`t` and `b` conflict, if neither center on y)
- `l` display about 0.5em left of x
- `r` display about 0.5em right of x
  (`l` and `r` conflict, if neither center on x)
- `x` make the bounding rectangle of text white
- `c` make the bounding circle of text white
  (`x` and `c` conflict)
- `o` outline any bounding rectangle or circle with a black line of width 1pt

Here '**em**' is the font size; capital letters are typically 0.7em high and lower case letters are typically 0.5em high.

The page commands are:

Command	Action
`#!layout R C [margin]`	Starts new physical page with R rows and C columns of logical pages, each logical page with margins as given in pt (1/72") (margin defaults to 18 or 0.25")
`#!newpage [pq]`	Starts new logical page with opposite corner coordinates optionally given by points p and q (which are not themselves displayed)
`#!header text...`	Adds text line to current logical page header

The output from a test case is one or more physical pages each containing one or more logical pages. The physical pages are numbered and labeled with the test case name (e.g., `00-000-vec2d`). Each '`header text...`' adds one line to the logical page header. If the first command is not a '`#!layout`' command, then '`#!layout 1 1`' is assumed. A '`#!newpage`' command immediately after a '`#!layout`' command may be omitted and will be assumed.

9 Linear Transforms

If we let $\mathcal{R}$ denote the set of real numbers, a.k.a., scalars, the set of all vectors is

$$\mathcal{R} \times \mathcal{R} = \mathcal{R}^2 = \{(v_x, v_y) | v_x, v_y \in \mathcal{R}\}$$

Definition 9.1 *A* **linear transformation** L *is a continuous map* $L : \mathcal{R}^2 \mapsto \mathcal{R}^2$ *such that for all vectors* v *and* w, $L(v + w) = L(v) + L(w)$.

Some examples of linear transformations are (1) the identity map, (2) rotations, (3) reflections about an axis, (4) scale changes (i.e., $(v_x, v_y) \longmapsto (s_x v_x, s_y v_y)$ for scalars s_x and s_y).

(In vector algebra, linear transformations with domains and/or ranges that are vector spaces other than $\mathcal{R}^2$ are studied.)

Lemma 9.2 *Let* L *be a linear transformation,* v *and* w *be vectors, and* $0 = (0, 0)$ *be the zero vector. Then* $L(0) = 0$, $L(-v) = -L(v)$, *and* $L(v-w) = L(v)-L(w)$.

> Proof: $L(0) = L(0 + 0) = L(0) + L(0)$ so subtracting $L(0)$ from both sides, $0 = L(0)$. $0 = L(0) = L(v - v) = L(v) + L(-v)$ so subtracting $L(v)$ from both sides, $-L(v) = L(-v)$. $L(v - w) = L(v + (-w)) = L(v) + L(-w) = L(v) + (-L(w)) = L(v) - L(w)$.

Lemma 9.3 *Let* L *be a linear transformation,* s *be a scalar, and* v *be a vector. Then* $L(s * v) = s * L(v)$.

> Proof: The cases $s = 0, 1, 2, -1, -2$ follow from the previous lemma and using induction proves the lemma for any integer s. For $s = n/d$ a rational number with integers n and d and $d > 0$,
> $n * L(v) = L(n * (d/d) * v) = L(d * (n/d) * v) = d * L((n/d) * v)$
> so dividing by d we get $L((n/d) * v) = (n/d) * L(v)$. For s a non-rational real number, the result follows from a continuity argument that we will leave to the mathematicians, since computer scientists only compute with rational numbers.

Let L be a linear transformation; let $v = (v_x, v_y)$ be a vector; let $u_x = (1, 0)$ and $u_y = (0, 1)$. Then

$$L(v) = L(v_x * u_x + v_y * u_y) = v_x * L(u_x) + v_y * L(u_y).$$

Lemma 9.4 *Given vectors ℓ_x and ℓ_y, the map $L : (v_x, v_y) \longmapsto v_x * \ell_x + v_y * \ell_y$ is a linear transformation.*

Proof: $L(v + w) = (v_x + w_x) * \ell_x + (v_y + w_y) * \ell_y = v_x * \ell_x + v_y * \ell_y + w_x * \ell_x + w_y * \ell_y = L(v) + L(w)$.

So L is determined by $\ell_x = L(u_x)$ and $\ell_y = L(u_y)$, and there is a 1-1 correspondence between linear transformations and vector pairs $[\ell_x, \ell_y]$. Therefore we will represent a linear transformation L by a pair of vectors `[`ℓ_x`,`ℓ_y`]` and represent application of L to the vector `v = (`v_x`,`v_y`)` by

$$L(v) \;=\; \texttt{[}\ell_x\texttt{,}\ell_y\texttt{]} * \texttt{(}v_x\texttt{,}v_y\texttt{)} \;=\; v_x{*}\ell_x{+}v_y{*}\ell_y$$

Here we use [] instead of () to surround the pair of vectors of a linear transformation, because our calculator does this to distinguish linear transformations from lists of vectors.

Examples:

1. `L=[(1,0),(0,1)]` is the identity map. $L(v) = v_x * (1, 0) + v_y * (0, 1) = (v_x, 0) + (0, v_y) = (v_x, v_y) = v$.
2. `L=[(0,1),(-1,0)]` rotates vectors 90°. Specifically, $L(u_x) = u_y$ and $L(u_y) = -u_x$ and $L(v) = v_x * (0, 1) + v_y * (-1, 0) = (0, v_x) + (-v_y, 0) = (-v_y, v_x)$.
3. `L=[(-1,0),(0,-1)]` reverses the direction of a vector. $L(v) = v_x * (-1, 0) + v_y * (0, -1) = (-v_x, 0) + (0, -v_y) = (-v_x, -v_y) = -v$.
4. `L=[(-1,0),(0,1)]` reflects vectors across the Y-axis. $L(v) = v_x * (-1, 0) + v_y * (0, 1) = (-v_x, 0) + (0, v_y) = (-v_x, v_y)$.
5. `L=[(`s_x`,0),(0,`s_y`)]` scales the X-axis by s_x and the Y-axis by s_y. $L(v) = v_x * (s_x, 0) + v_y * (0, s_y) = (s_x v_x, 0) + (0, s_y v_y) = (s_x v_x, s_y v_y)$.

Now suppose we have a scalar `s`, vector `v`, and two linear transformations `K =` `[`ℓ_x^K`,`ℓ_y^K`]`, `L = [`ℓ_x^L`,`ℓ_y^L`]`. Then we can define:

`s*L = [s*`ℓ_x^L`,s*`ℓ_y^L`]` so that `(s*L)*v = s*(L*v)`
`K+L = [`ℓ_x^K`+`ℓ_x^L`,`ℓ_y^K`+`ℓ_y^L`]` so that `(K+L)*v = K*v + L*v`
`K-L = [`ℓ_x^K`-`ℓ_x^L`,`ℓ_y^K`-`ℓ_y^L`]` so that `(K-L)*v = K*v - L*v`
`-L = [-`ℓ_x^L`,-`ℓ_y^L`]` so that `(-L)*v = - L*v`
`K*L = [K*`ℓ_x^L`,K*`ℓ_y^L`]` so that `(K*L)*v = K*(L*v)`

$\star$ and + as defined here are **bi-linear**, which means (supposing t to be another scalar and J to be another linear transformation):

```
s*(K+L) = s*K + s*L
(s+t)*K = s*K + t*K
(J+K)*L = J*L + K*L
J*(K+L) = J*K + J*L
```

Addition of linear transformations commutes, K+L = L+K, but multiplication does not: in general K*L is not equal to L*K (as an example, a rotation does not generally commute with a reflection: see the end of the Rotations and Reflections section below where $R^{\phi} * F^{\omega} = F^{\omega} * R^{-\phi}$).

Let I be the identity transform defined by I*v=v for all vectors v (and therefore I=[u_x, u_y]). It is easy to check that K*I = K = I*K for all linear transforms K.

A linear transform K is defined to be the (two-sided) **inverse** of the linear transform L if and only if K*L = I = L*K. The inverse of L might not exist, but if it does, it is unique since if there were two, K_1 and K_2, then

$$K_1 = K_1 * I = K_1 * (L * K_2) = (K_1 * L) * K_2 = I * K_2 = K_2$$

The notation L^{-1} is used for the inverse of L if it exists. For example, the inverse of a rotation by angle ϕ in the counter-clockwise direction is a rotation by angle ϕ in the clockwise direction.

(One-sided inverses that are not two-sided inverses only exist for linear transformations whose domain and range are vector spaces of different dimensions, unlike our linear transformations whose domain and range are both 2-dimensional.)

10 Linear Transform Calculator

Implement additions to the vector calculator for just *linear-transform* value types and the *operators* and *functions*:

Assume: `s`=s is a scalar
`d`=d is an integer scalar, $-15 \leq d \leq +15$
`v`=`(`v_x`,` v_y`)` and `w`=`(`w_x`,` w_y`)` are vectors
`K`=`[`ℓ_x^K`,` ℓ_y^K`]` and `L`=`[`ℓ_x^L`,` ℓ_y^L`]` are linear transforms

then:

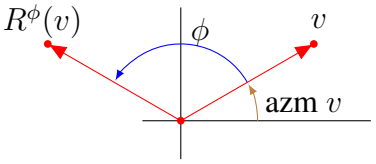
`[v,w]`	returns $[v, w]$	
`K==L:d`	returns $\ell_x^K == \ell_x^L : d$ AND $\ell_y^K == \ell_y^L : d$	
`K:d`	returns $[\ell_x^K : d, \ell_y^K : d]$	
`L.x`	returns ℓ_x^L (`x` is not a variable here)	
`L.y`	returns ℓ_y^L (`y` is not a variable here)	
`L*v`	returns $v_x * \ell_x^L + v_y * \ell_y^L$	[application]
`s*L`	returns `[`$s\ell_x^L$`,` $s\ell_y^L$`]`	
`K+L`	returns `[`$\ell_x^K + \ell_x^L$`,` $\ell_y^K + \ell_y^L$`]`	
`K-L`	returns `[`$\ell_x^K - \ell_x^L$`,` $\ell_y^K - \ell_y^L$`]`	
`-L`	returns `[`$-\ell_x^L$`,` $-\ell_y^L$`]`	
`K*L`	returns `[K*`ℓ_x^L`,K*`ℓ_y^L`]`	[composition]

Sample Input Files: **00-XXXX-linear-vec2d.in**
Sample Output Files: **00-XXXX-linear-vec2d.ftest**
Sample Run File: **sample-linear-vec2d.run**
Submit Run File: **submit-linear-vec2d.run**

11 Rotations and Reflections

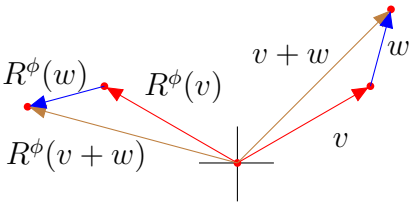
Rotations and reflections are the two kinds of linear transformations that preserve vector lengths (i.e., $||L(v)|| = ||v||$).

A **rotation** R^ϕ by ϕ degrees (counter-clockwise) is defined mathematically by $||R^\phi(v)|| = ||v||$
azm $R^\phi(v) =$ azm $v + \phi$ if $||v|| \neq 0$

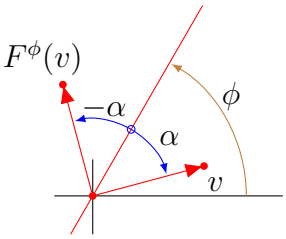


A rotation preserves angles between vectors; that is:
azm $R^\phi(v) -$ azm$R^\phi(w) =$ azm $v -$ azm w if $||v|| \neq 0 \neq ||w||$
Also, adding integer multiples of 360 to ϕ does <u>not</u> change R^ϕ.

A rotation preserves side-lengths and angles of a triangle. Therefore, a rotation preserves vector addition, that is, $R^\phi(v + w) = R^\phi(v) + R^\phi(w)$, and thus satisfies Definition 9.1 and is a linear transformation.

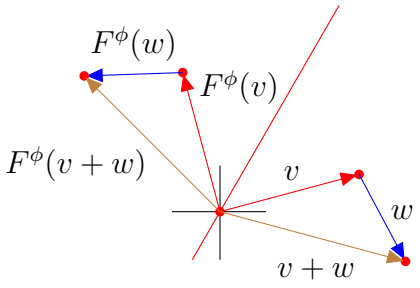


A **reflection** F^ϕ by ϕ degrees, which is a reflection across the line with direction ϕ through the origin, is defined mathematically by
$||F^\phi(v)|| = ||v||$
azm $F^\phi(v) = \phi - \alpha$
where $\alpha =$ azm $v - \phi$ so
azm $F^\phi(v) = 2 * \phi -$ azm v if $||v|| \neq 0$



A reflection negates angles between vectors; that is:
azm $F^\phi(v) -$ azm$F^\phi(w) = -($azm $v -$ azm $w)$ if $||v|| \neq 0 \neq ||w||$
Also, adding integer multiples of 180 to ϕ does <u>not</u> change F^ϕ.

A reflection
preserves side-lengths and negates
angles of a triangle. Therefore,
a reflection preserves vector addition,
that is, $F^\phi(v+w) = F^\phi(v) + F^\phi(w)$,
and thus satisfies Definition 9.1
and is a linear transformation.



A linear map L that preserves vector lengths ($||L(v)|| = ||v||$) is said to be **unitary**. A unitary linear map L also preserves the absolute values of angles between vectors, because L preserves the side-lengths of any triangle made of vectors, and these determine the absolute values of the angles of the triangle (but not the direction of these angles). This means that `azm L(v)` must be either 'constant `+ azm v`' or 'constant `- azm v`'. These two case are rotations and reflections, respectively.

Now L, being linear, is determined by $L(u_x)$ and $L(u_y)$, where $u_x = (1,0)$ and $u_y = (0,1)$, and as u_y is perpendicular to u_x, $L(u_y)$ must be perpendicular to $L(u_x)$ since L is unitary.

Putting all this together we have:

Lemma 11.1 *For unitary $L = [\ell_x, \ell_y]$, ℓ_x and ℓ_y are vectors of unit length, and ℓ_y is perpendicular to ℓ_u.*

If ℓ_y is ℓ_x rotated counter-clockwise by 90 degrees, L is a rotation, whereas if ℓ_y is ℓ_x rotated clockwise by 90 degrees, L is a reflection.

If L is a rotation, the angle of rotation is:

$$\phi = \text{azm } L(u_x) - \text{azm } u_x = \text{azm } L(u_x)$$

and we have: $L(u_x) = (\cos\phi, \sin\phi)$
$L(u_y) = (-\sin\phi, \cos\phi)$

If L is a reflection, the angle of the reflecting line is:

$$\phi = (1/2) * (\text{azm } L(u_x) + \text{azm } u_x) = (1/2) * (\text{azm } L(u_x))$$

and we have: $L(u_x) = (\cos(2*\phi), \sin(2*\phi))$
$L(u_y) = (\sin(2*\phi), -\cos(2*\phi))$

Both rotations and reflections preserve vector lengths and change vector angles according to a linear formula. Using the notation R^ϕ for rotation by angle ϕ, and

F^ϕ for reflection about the line at angle ϕ, we get the following:

$$R^\phi * R^\omega = R^\omega * R^\phi = R^{\phi+\omega} \quad \text{because} \quad (\text{azm } v + \omega) + \phi = \text{azm } v + (\phi + \omega)$$
$$F^\phi * F^\omega = R^{2(\phi-\omega)} \quad \text{because} \quad 2\phi - (2\omega - \text{azm } v) = \text{azm } v + 2(\phi - \omega)$$
$$R^\phi * F^\omega = F^\omega * R^{-\phi} \quad \text{because} \quad (2\omega - \text{azm } v) + \phi = 2\omega - (\text{azm } v + (-\phi))$$

Note that:

$R^\phi * R^\omega = R^{\omega+\phi} = R^\omega * R^\phi$
R^ϕ and R^ω commute
$R^{-\phi}$ is the inverse of R^ϕ
$F^\phi * F^\phi = I$ (the identity) so F^ϕ is its own inverse
$F^\omega * F^\phi$ is the inverse of $F^\phi * F^\omega$
in general F^ϕ and F^ω do not commute

12 Rotations and Reflections Calculator

Implement additions to the linear calculator for the *operators* and *functions*:

Assume:
`p` is a scalar
`v` is a vector
then:

`v^p`	returns `w` such that `||w||=||v||` and `azm w = azm v + p`, `v` rotated by `p` degrees; undefined if `||v|| = 0`
`v|p`	returns `w` such that `||w||=||v||` and `azm w = 2*p-azm v`, `v` reflected across the line with direction `p` degrees; undefined if `||v|| = 0`
`^p`	returns `[`u_x`^p,`u_y`^p]`, the rotation R^p with angle p degrees
`|p`	returns `[`u_x`|p,`u_y`|p]`, the reflection F^p across the line with direction p degrees

Sample Input Files:	**00-XXXX-unitary-vec2d.in**
Sample Output Files:	**00-XXXX-unitary-vec2d.ftest**
Sample Run File:	**sample-unitary-vec2d.run**
Submit Run File:	**submit-unitary-vec2d.run**

13 Products

There are two products of a pair of vectors that play a very important role in 2D computational geometry:

Definition 13.1 *The* **scalar product** *of two vectors*

$$\texttt{v} = (v_x, v_y) \textit{ and } \texttt{w} = (w_x, w_y)$$

is

$$\texttt{v*w} = v_x w_x + v_y w_y$$

The **cross product** *(a.k.a.,* **wedge product***) of the two vectors is*

$$\texttt{v\^{}w} = (-v_y) w_x + v_x w_y$$

Note that $||\texttt{v}||^2 = \texttt{v*v}$.

If we denote the rotation counter-clockwise by 90 degrees by $\texttt{R}^{90}$, then

$$\texttt{v\^{}w} = (-v_y, v_x)\texttt{*w} = \texttt{R}^{90}\texttt{(v)*w}$$

Both the scalar and cross product are **bi-linear**. That is, if `u`, `v`, `w` are vectors and `s` is a scalar:

$$\begin{aligned}
\texttt{(u+v)*w} &= \texttt{u*w} + \texttt{v*w} \\
\texttt{u*(v+w)} &= \texttt{u*v} + \texttt{u*w} \\
\texttt{(s*u)*w} &= \texttt{s*(u*w)} = \texttt{u*(s*w)} \\
\texttt{(u+v)\^{}w} &= \texttt{u\^{}w} + \texttt{v\^{}w} \\
\texttt{u\^{}(v+w)} &= \texttt{u\^{}v} + \texttt{u\^{}w} \\
\texttt{(s*u)\^{}w} &= \texttt{s*(u\^{}w)} = \texttt{u\^{}(s*w)}
\end{aligned}$$

The scalar product is symmetric (`v*w = w*v`) while the cross product is anti-symmetric (`v^w = -w^v`).

Lemma 13.2 *For a unitary transform* `L`*:*

1. `L(v)*L(w) = v*w` *for all vectors* `v` *and* `w`.
2. *If* `L` *is a rotation,* `L(v)^L(w) = v^w` *for all vectors* `v` *and* `w`*;*
   if `L` *is a reflection,* `L(v)^L(w) = - v^w` *for all vectors* `v` *and* `w`.

Proof: Since L is unitary:

$$\begin{aligned} ||v+w||^2 &= (v+w)*(v+w) = v*v + v*w + w*v + w*w \\ &= ||v||^2 + 2*v*w + ||w||^2, \\ \text{so } v*w &= (1/2) \\ &* (||v+w||^2 - ||v||^2 - ||w||^2), \\ \text{so } L(v)*L(w) &= (1/2) \\ &* (||L(v)+L(w)||^2-||L(v)||^2-||L(w)||^2) \\ &= (1/2)*(||v+w||^2 - ||v||^2 - ||w||^2) \\ &= v*w \end{aligned}$$

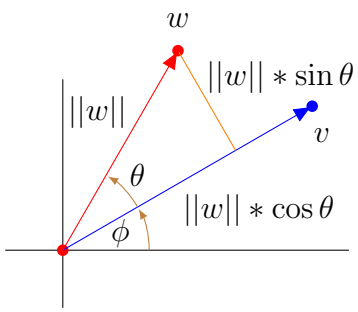
If L is a rotation, $R^{90}*L=L*R^{90}$ so

$$\begin{aligned} L(v)\hat{}L(w) &= R^{90}(L(v))*L(w) = L(R^{90}(v))*L(w) \\ &= R^{90}(v)*w = v\hat{}w \end{aligned}$$

If L is a reflection, $R^{90}*L=L*R^{-90}$ and $R^{-90} = -R^{90}$ so

$$\begin{aligned} L(v)\hat{}L(w) &= R^{90}(L(v))*L(w) = L(R^{-90}(v))*L(w) \\ &= R^{-90}(v)*w = -R^{90}(v)*w = -v\hat{}w \end{aligned}$$

We can use the fact that $*$ and $\hat{}$ are invariant under rotations to give a geometric characterization of these two products. Let v and w be two vectors, let $\theta = \text{azm } w - \text{azm } v$ be the direction angle of w relative to v, and let $\phi = \text{azm } v$ be the azimuth of v (see the picture). If we rotate both vectors by $R^{-\phi}$ we line v up with the positive X-axis, while not changing θ, $||v||$, $||w||$, $v*w$, or $v\hat{}w$. So after rotation we have



$$\begin{aligned} v &= (||v||, 0) & w &= (||w|| \cos\theta, ||w|| \sin\theta) \\ v*w &= ||v|| \times ||w|| \times \cos\theta & v\hat{}w &= ||v|| \times ||w|| \times \sin\theta \end{aligned}$$

From this we deduce two things. First,

$$v*w = ||v|| \times ||w|| \times \cos\theta \quad v\hat{}w = ||v|| \times ||w|| \times \sin\theta$$

before rotation, as well as after rotation, which gives a geometric characterization of the products.

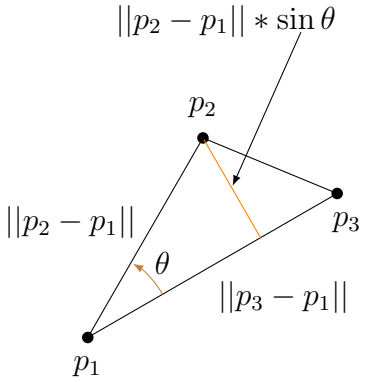
Second, $$||v|| * R^{-\text{azm } v} * w = (v*w, v\hat{}w),$$

which gives us a quick way to change coordinate systems by a rotation times scalar multiplication by $||v||$ so that v has coordinates $(||v||^2, 0)$ in the new coordinate

system. Problems that might be hard to solve in the original coordinate system can be easily solved in the new coordinate system. For example, the end of w is to the left of the infinite directed line extending v if and only if $v\hat{}w > 0$. One can also solve distance problems in the original coordinate system by solving them in the new coordinate system while remembering that the new system distances are $||v||$ times the original system distances. If in addition $||v|| = 1$, distances in the new system equal distances in the original system.

Lastly,
looking at the picture at the right, we see that:

area of triangle $p_1p_2p_3$
= $(1/2) * ||p_2 - p_1|| * \sin\theta * ||p_3 - p_1||$
= $(1/2) * (p_3 - p_1)\hat{}(p_2 - p_1)$
= $-(1/2) * (p_2 - p_1)\hat{}(p_3 - p_1)$



So we can use the cross product to compute triangle areas, but its a bit tricky because what we get is a signed area. The result is positive if the angle θ measured from the first factor of $\hat{}$ to the second factor of $\hat{}$ is in the range $[0, 180]$, and negative if θ is in the range $[-180, 0]$.

14 Products Calculator

Implement additions to the unitary calculator for the following *operators* and *functions*:

Assume:	`v`=v=(v_x, v_y) (a vector) `w`=w=(w_x, w_y) (a vector) $u_x = (1, 0)$ and $u_y = (0, 1)$ `p`=p, `q`=q, `r`=r (vectors representing points)
then:	
`v*w`	returns $v_x w_x + w_y w_y$, the scalar product of `v` and `w`
`v^w`	returns $(-v_y)w_x + v_x w_y$, the cross product of `v` and `w`
`v:w`	returns $\|\|v\|\| * R^{-\text{azm } v} * w = (v * w, v \hat{} w)$
`<v>`	returns $(1/\|\|v\|) * v$, the unit vector in the same direction as v
`v!w`	returns $R^{-\text{azm } v} * w = (< v > * w, < v > \hat{} w)$
`v:`	returns `[`v`:`u_x`,`v`:`u_y`]` (so `(v:)*w = v:w`)
`v!`	returns `[`v`!`u_x`,`v`!`u_y`]` (so `(v!)*w = v!w`)
`area pqr`	returns $0.5 * (r - p) \hat{} (q - p)$, the area of triangle pqr, with positive sign if pqr clockwise, and negative sign if counter-clockwise

If `||v|| = 0` then `<v>` and `v!w` will produce coordinates such as `inf` (plus infinity), `-inf` (minus infinity), and `nan` (not a number; i.e., result could not be computed).

Sample Input Files:	**00-XXXX-product-vec2d.in**
Sample Output Files:	**00-XXXX-product-vec2d.ftest**
Sample Run File:	**sample-product-vec2d.run**
Submit Run File:	**submit-product-vec2d.run**

15 Line and Point

In this section we investigate the following questions:

1. Is a point on, to the left of, or to the right of an infinite directed line? What is the distance from the point to the line and where is the closest point that is on the line?
2. What is the general position of a point relative to a finite line segment? What is the distance from the point to the line segment?
3. What is the general position of a direction relative to two other directions, where all three directions are defined by directed lines passing through a common point?

We begin with a line `pq` from point `p` to point `q`, `p`≠`q`. We may take this line to be either infinite, or to be a finite line segment with ends `p` and `q`. We may take the line to be undirected, or to be directed from `p` to `q`.

We then consider a third point `r` and ask about its relation to `pq`.

The answers to our questions will not be changed if we perform a coordinate translation on all the points, so we begin by moving `p` to the origin `(0,0)` using a translation by `-p`. Our points become:

```
p - p = (0,0)
    q - p
    r - p
```

Next we apply the rotation $R^{-\texttt{azm(q-p)}}$ followed by multiplication by `||q-p||`. This was computed in the last section as the linear transform

$$\texttt{(q-p):} \quad = \texttt{||q-p||} * R^{-\texttt{azm(q-p)}}$$

This transformation preserves angles and multiplies distances by `||q-p||`.

If we apply `(q-p):` after the translation `-p`, we get:

```
          P = (q-p):(p-p) = (0,0)
      Q = (q-p):(q-p) = ( (q-p)*(q-p), 0 )
R = (q-p):(r-p) = ( (q-p)*(r-p), (q-p)^(r-p) )
```

where we use the facts that `p - p = (0,0)` and `(q-p)^(q-p) = 0`.

Next we define `X` and `Y` to be the coordinates of `R` so `R = (X,Y)`, and `L` to be the X coordinate of `Q` so `Q = (L,0)`.

If we want the distance from `r` to the infinite line `pq`, then this is the same as `1/||q-p||` times the distance from `R = (X,Y)` to the infinite line `PQ`, and the latter is the X-axis. So the answer is

`|Y|/||q-p|| = |(q-p)^(r-p)|/||q-p||`

Next we want to find the point `t` on the infinite line `pq` that is closest to `r`. We will represent `t` as `t = p + s*(q-p)` for the appropriate scalar `s`. Then `t-p` is the closest point to `r-p` on the infinite line from `(p-p)` to `(q-p)` and if we apply the linear transformation `(q-p):` $= ||q-p|| * R^{-\texttt{azm(q-p)}}$, we get that `T = (q-p):(t-p)` is the closest point to `R` on the line `PQ`, since rotations preserve distances and multiplication of all coordinates by `||q-p||` preserves the notion of 'closest'. But the closest point to `R = (X,Y)` on `PQ =` the X-axis is `(X,0)`, so `T = (X,0)`. And as `(q-p):` is a linear transformation and `t-p = s*(q-p)`, then `T = s*(Q-P) = s*Q` (since `P = (0,0)`).

so `T = (X,0) = s*Q = (s*L,0)`
so `s = X/L`
therefore `t = p + (X/L)*(q-p)`
or `t = p + ((q-p)*(r-p)/(q-p)*(q-p)) * (q-p)`

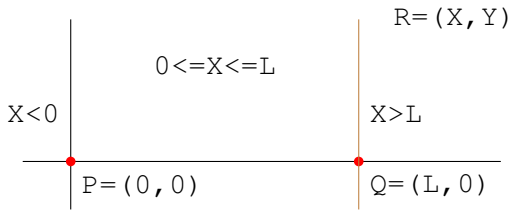
Note that if `p`, `q`, and `r` have rational coordinates, then `t` has rational, and <u>not</u> irrational, coordinates. However the denominator may be large. If the input coordinates have at most `d` decimal places, their denominator is at most 10^d, and the denominator of `t` may be as large as

`10^d*(`numerator of `L)`

where we use the fact that the denominators of `X` and `L` which are 10^{2d} cancel each other out in `X/L` so

`X/L = (`numerator of `X)/(`numerator of `L)`

If we want the distance from `r` to the finite line `pq`, then this is the same as `1/||q-p||` times the distance from `R` to the finite line `PQ`, and there are three cases. If `X<0`, the answer is `||R-P||/||q-p||`. If `X>L`, the answer is `||R-Q||/||q-p||`. Otherwise the answer is `|Y|/||q-p||` as before. See picture.



Here we can simplify the above by using:

```
||R-P||/||q-p|| = ||(q-p):(r-p)||/||q-p|| = ||r-p||
||R-Q||/||q-p|| = ||(q-p):(r-q)||/||q-p|| = ||r-q||
```

Now suppose we want a precise answer to the question: is `r` on the infinite or finite line `pq`? This is the same as the question: is `R` on the infinite or finite line `PQ`? If the coordinates of `p`, `q`, and `r` have at most `d` decimal places, they are rational numbers with denominator 10^{d}, and the products will be rational numbers with denominator 10^{2d}, so the coordinates of `R` and `Q` will have denominator 10^{2d}. Or more precisely, the computed coordinates will be close approximations to such rational numbers. Therefore

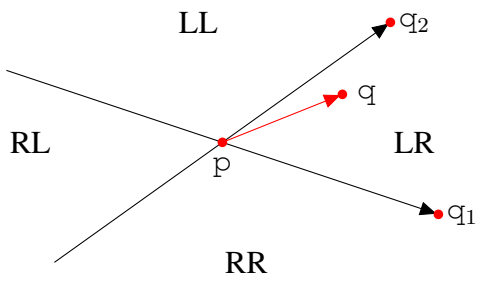
1. `r` is to the left of the infinite directed line `pq` if and only if `0<Y:(2d)`.
2. `r` is to the right of the infinite directed line `pq` if and only if `Y<0:(2d)`.
3. `r` is on the infinite directed line `pq` if and only if `Y==0:(2d)`.
4. `r` is on the finite line `pq` if and only if
   `Y==0:(2d)`, `0<=X:(2d)`, and `X<=L:(2d)`

There is a difference in the treatment of `X < 0` between the problem of computing a distance from `r` to the finite line `pq` and the problem of computing whether `r` is on the finite line `pq`. In both problems the line `X = 0` is a boundary between different cases. In the distance case, the distance is continuous across this boundary, so the computed value, the distance, does not change much when `(X,Y)` moves across the boundary. In the other case, the computed value may change from `false` to `true` as `(X,0)` moves from `X < 0` to `0 <= X`, and thus the computed value is discontinuous.

In the continuous case we need not concern ourselves with how many decimal places `X` has, and we can use `X < 0` instead of `X<0:(2d)`. In the discontinuous case we cannot get a meaningful `true/false` answer unless we restrict the number of decimal places in `X`, though we could get a `true/false/maybe` answer if we knew more about the potential error in our value for `X`.

We will say that `X = 0` is a **boundary** that is **continuous** in the distance computation and **discontinuous** in the `on/not-on` computation. For continuous boundaries we can use inequalities to decide which side of the boundary we are on without worrying about accuracy. For discontinuous boundaries we must worry about accuracy and use tests like `X<0:(2d)`.

Lastly, let us
find the general position of a direction `pq` relative to two other directions, pq_1 and pq_2, where q_2 is to the left of the infinite directed line pq_1 (see picture). The infinite directed lines pq_1 and pq_2 divide space into four regions:



LR for points q that are to the left of pq_1 and to the right of pq_2;

LL for points q that are to the left of pq_1 and to the left of pq_2;

RL for points q that are to the right of pq_1 and to the left of pq_2;

RR for points q that are to the right of pq_1 and to the right of pq_2;

Thus, for example, if q is in region LR, then when pq_1 is rotated counter-clockwise it will line up with pq before it lines up with pq_2.

Therefore we can use what we already know about finding which side of an infinite line a point is on to find the general position of a direction relative to two other directions.

Note that if pq_2 and pq_1 have the same direction, LR and RL are empty, whereas if pq_2 and pq_1 have opposite directions, LL and RR are empty.

16 Line and Point Calculator

Implement additions to the product calculator for the following *operators* and *functions*:

Assume:	`p`, `q`, `r`, `u`, `v` are vectors representing points `D`=D is an integer scalar, $-15 \leq D \leq +15$
then:	
`disti pqr`	returns the distance from point `r` to the infinite line `pq`
`distf pqr`	returns the distance from point `r` to the finite line `pq`
`closei pqr`	returns the point `t` on the infinite line `pq` that is closest to the point `r`
`closef pqr`	returns the point `t` on the finite line `pq` that is closest to the point `r`
`sidei pqrD`	returns `+1` if `r` is to the left of the infinite directed line `pq`, zero if it is on the line, and -1 if it is to the right; if all coordinates are exact multiples of 10^{-D} and have absolute values $\leq 10^{6-D}$
`onf pqrD`	returns `true` if and only if `r` is on the finite line `pq`; if all coordinates are exact multiples of 10^{-D} and have absolute values $\leq 10^{6-D}$
`between puvwD`	returns `true` if and only if the direction `pu` rotated counter-clockwise will line up with the direction `pv` before it lines up with `pw`; but returns `false` if any two of the three directions are identical; if all coordinates are exact multiples of 10^{-D} and have absolute values $\leq 10^{6-D}$

Sample Input Files:	**00-XXXX-point-vec2d.in**
Sample Output Files:	**00-XXXX-point-vec2d.ftest**
Sample Run File:	**sample-point-vec2d.run**
Submit Run File:	**submit-point-vec2d.run**

17 Line and Line

In this section we investigate the following questions:

1. Do two infinite lines intersect, and if so where?
2. Does a finite line intersect an infinite line?
3. What is the distance between a finite line and an infinite line?
4. Do two finite lines intersect?
5. What is the distance between two finite lines?

Let `p` and `q` be a pair of distinct points so that the set of points on the infinite line `pq` is:

$$K = \{\ p + s*(q-p) : s \text{ any scalar}\ \}$$

Let `m` and `n` be another pair of distinct points so that the set of points on the infinite line `mn` is:

$$K = \{\ m + t*(n-m) : t \text{ any scalar}\ \}$$

If these two infinite lines intersect, there must exist values for `s` and `t` such that:

$$p + s*(q-p) = m + t*(n-m)$$

The easy way to solve this is to take the cross product of both sides with `n-m`, because `(n-m)^(t*(n-m)) = t*((n-m)^(n-m)) = t*0 = 0`. So we end up with:

$$s*(n-m)^(q-p) = (n-m)^(m-p)$$

which we can solve for `s`. Unless `(n-m)^(q-p) = 0`.

If $(n-m)^{(q-p)} \neq 0$ then plugging `s` into `p + s*(q-p)` and using

$$(n-m)^{(q-p)} - (n-m)^{(m-p)} = (n-m)^{(q-m)}$$

we get that the intersection point is:

$$\frac{[(n-m)^{(q-m)}]*p + [(n-m)^{(m-p)}]*q}{(n-m)^{(q-p)}}$$

What happens if `(n-m)^(q-p) = 0`?

Then `(n-m)^(q-p) = sin θ*||n-m||*||q-p|| = 0`

where `||n-m|| ≠ 0 ≠ ||q-p||`

and θ `= azm (q-p) - azm (n-m)`

so `sin θ = 0`

so θ `= 0 or ±180`

so `nm` and `pq` are parallel

The converse is also true. If `mn` and `pq` are parallel,

then `azm (n-m) = azm (q-p)`

or `azm (n-m) = azm (q-p) ± 180`

and `||n-m|| ≠ 0 ≠ ||q-p||`

so `n-m = s*(q-p)` for some scalar `s`

so `(n-m)^(q-p) = s*((q-p)^(q-p)) = 0`

Of course, testing a product to see if it is `0` is subject to rounding errors. So we assume that the coordinates of `m`, `n`, `p`, and `q` have at most `d` decimal places, and therefore `mn` is parallel to `pq` if and only if `(n-m)^(q-p)==0:(2d)`.

If `mn` and `pq` are parallel infinite lines, they may be disjoint or identical. To test which, test whether `m` is on the infinite line `pq`

The question of whether a finite line `mn` intersects an infinite line `pq` can be answered by applying the translation by `-p` and then the linear transform `(q-p):` to get:

```
        m  ⟼  M = ( (q-p)*(m-p), (q-p)^(m-p) )
        n  ⟼  N = ( (q-p)*(n-p), (q-p)^(n-p) )
infinite pq ⟼  X-axis
```

Then intersection occurs if and only if `M.y` and `N.y` have opposite signs or one is zero, or more precisely:

Theorem 17.1 *If the coordinates of* `m`*,* `n`*,* `p`*, and* `q` *have at most* `d` *decimal places, finite* `mn` *intersects infinite* `pq` *if and only if:*

`M.y > 0:(2d)` *and* `N.y < 0:(2d)`

or

`M.y < 0:(2d)` *and* `N.y > 0:(2d)`

or

`M.y == 0:(2d)`

or

`N.y == 0:(2d)`

where `M.y == (q-p)^(m-p)` *and* `N.y == (q-p)^(n-p).`

The question of intersection has a discontinuous true/false answer, so the boundaries `M.y == 0` and `N.y == 0` are discontinuous for this question. However, for the question of the distance of finite `mn` from infinite `pq`, the boundaries are continuous, and remembering that distances in the transformed coordinates are `||q-p||` times distances in the original coordinates, we have:

Theorem 17.2 *The distance between the finite line* `mn` *and the infinite line* `pq` *is* `0` *(due to intersection) unless*

`M.y > 0` *and* `N.y > 0`

or

`M.y < 0` *and* `N.y < 0`

in which case it is

`min (|M.y|, |N.y|) / ||q-p||`

where `M.y == (q-p)^(m-p)` *and* `N.y == (q-p)^(n-p).`

The question of whether two finite lines intersect is made simpler by the following:

Lemma 17.3 *Two finite lines do not intersect if either does not intersect the infinite extension of the other.*

Otherwise the two finite lines intersect at a single point unless the infinite extensions of the two finite lines are identical.

Proof: The first statement is clear.

If the infinite extensions of the finite lines are parallel, either the infinite extensions are identical, or neither finite line intersects the infinite extension of the other line. So if both the finite lines intersect the infinite extension of the other, and if these infinite extensions are not identical, then the infinite extensions are not parallel, so they must intersect at a single point. So each finite line must intersect the infinite extension of the other line at a single point.

Then let the finite lines be ℓ_1 and ℓ_2 and let ℓ_i intersect the infinite extension of the other finite line at the point r_i. So $r_1 = r_2 =$ the intersection of the infinite extensions of both lines, and as r_i is on ℓ_i, this point is on both finite lines.

We can check whether the infinite extensions of the mn and pq are the same by checking whether m and n are both on the infinite extension of pq, or in other words, whether:

(q-p)^(m-p)==0:(2d) and 0==(q-p)^(n-p):(2d)

If the two infinite extensions of the two finite lines pq and mn are the same infinite line, we need a new approach. We need to establish a coordinate system on this infinite line.

If r is a point on this infinite line then we can map

r ↦ R = ((q-p)*(r-p), (q-p)^(r-p)) = (R.x,0)

and use the map

r ↦ R.x = (q-p)*(r-p)

as coordinates on this line. In this system of coordinates distances are ||q-p|| times distances in the original system of coordinates.

In this coordinate system,

P.x = (q-p)*(p-p) = 0 Q.x = (q-p)*(q-p) > 0
M.x = (q-p)*(m-p) N.x = (q-p)*(n-p)

We will switch m and n if necessary so that M.x < N.x. Then the two finite lines are pq represented by the interval [P.x,Q.x] and mn represented by the interval [M.x,N.x]. Therefore pq and mn

do not intersect if N.x < P.x:(2d) or Q.x < M.x:(2d)
do intersect if N.x ≥ P.x:(2d) and Q.x ≥ M.x:(2d)

Deciding whether or not pq and mn intersect is a discontinuous problem. However finding the length of the intersection under the assumption that both lines have the same infinite extension is a continuous problem. The intersection of pq and mn, expressed using our line coordinates, is:

$$[P.x,Q.x]\cap[M.x,N.x] = [\max(P.x,M.x),\min(Q.x,N.x)]$$

so the length of the intersection is

$$\frac{\min(Q.x,N.x) - \max(P.x,M.x)}{||q-p||} \text{ if this is } \geq 0$$

or 0 otherwise

This length is continuous across the boundary

$$\min(Q.x,N.x) = \max(P.x,M.x)$$

Lastly we consider the problem of finding the distance between the finite line mn and the finite line pq. We assume:

Theorem 17.4 *If* mn *and* pq *are finite lines, the function from pairs of points* r_{mn} *on* mn *and* r_{pq} *on* pq *to the distance* $||r_{mn} - r_{pq}||$ *has a minimum.*

We will not give a proof of this, which involves the mathematics of compact sets.

But we will use this theorem to prove:

Theorem 17.5 *If either* mn *and* pq *are finite lines that do not intersect, or* mn *and* pq *are finite lines whose infinite extensions are identical, then points* r_{mn} *on* mn *and* r_{pq} *on* pq *may be chosen so that* $||r_{mn} - r_{pq}||$ *is minimal and one of the chosen points is an end-point of its finite line.*

Therefore the distance from mn *to* pq *is the minimum of:*

the distance from m *to* pq
the distance from n *to* pq
the distance from p *to* mn
the distance from q *to* mn

Proof: By the previous theorem, r_{mn} in mn and r_{pq} in pq can be chosen so that $||r_{mn} - r_{pq}||$ is minimal.

First assume that the infinite extensions of mn and pq are not identical, and therefore mn and pq do not intersect. Consider the infinite line ℓ through r_{mn}

and r_{pq}, and assume that neither r_{mn} or r_{pq} are endpoints. Then ℓ may be translated to its left or right by a small enough amount that the translation continues to intersect both finite lines. Let the translation be ℓ' and its intersections with the finite lines be r'_{mn} and r'_{pq}. If mn and pq are not parallel, then $||r'_{mn} - r'_{pq}||$ will increase as ℓ is translated in one direction and decrease as it is translated in the other direction, but a decrease cannot happen by assumption of minimality of $||r_{mn} - r_{pq}||$, so by contradiction either r_{mn} or r_{pq} must be an endpoint.

If on the other hand mn and pq are parallel, $||r'_{mn} - r'_{pq}||$ will remain constant for all translations of ℓ, and ℓ can be translated in either direction until it encounters an endpoint.

Second, assume that the infinite extensions of mn and pq are identical (mn and pq may or may not intersect). If $||r'_{mn} - r'_{pq}|| > 0$ and either r_{mn} or r_{pq} is not an endpoint, the one that is not an endpoint can be translated right or left to decrease $||r'_{mn} - r'_{pq}||$, contradicting the minimality of $||r'_{mn} - r'_{pq}||$. If on the other hand $||r'_{mn} - r'_{pq}|| = 0$ and $r_{mn} = r_{pq}$ is not an endpoint, it can be translated right or left until it becomes an endpoint.

The translation argument just given is an example of a **perturbation argument**. Perturbation arguments are common in computational geometry.

Of course if the finite lines intersect the distance between them is zero.

Corollary 17.6 *If two finite lines* mn *and* pq *intersect, the distance between them is* 0. *Otherwise the distance is the minimum of:*

the distance from m *to* pq
the distance from n *to* pq
the distance from p *to* mn
the distance from q *to* mn

This corollary gives an algorithm to compute the distance between two finite lines as follows. Let A be the answer to the question: do the lines intersect? Let D be the distance computed by the corollary if A is false. Then if A is true, return 0, else return D.

But now we have a difficulty: the algorithms we have developed above for computing A are discontinuous and impose precision requirements on the data. What

we want is a continuous algorithm to compute A, or more specifically, an algorithm that makes errors only if D = 0 to within computational accuracy. Then if A is erroneously false, D will be returned, which is the same value that would be returned if A is true, to within computational accuracy. And if A is erroneously true, 0 would be returned, which is the same as D to within computation accuracy. Then our distance algorithm will be continuous and not impose precision requirements on the data.

The following algorithm A has the require continuity property:

Algorithm 17.7 *Algorithm that returns true if finite lines* `mn` *and* `pq` *intersect, and false otherwise.*

Make the coordinate change:

```
m  ⟼  M = ( (q-p)*(m-p), (q-p)^(m-p) )
n  ⟼  N = ( (q-p)*(n-p), (q-p)^(n-p) )
p  ⟼  P = ( (q-p)*(p-p), (q-p)^(p-p) ) = (0,0)
q  ⟼  Q = ( (q-p)*(q-p), (q-p)^(q-p) ) = (L,0)
       for L = (q-p)*(q-p) > 0
```

so infinite `pq` $\longmapsto$ *X-axis.*

Intersect the finite line `MN` *with the region*

$$S = [0, L] \times (-\infty, +\infty) = \{(x, y) : 0 \leq x \leq L, -\infty < y < +\infty\}$$

If the intersection is empty, return false.

Otherwise let `M'` *and* `N'` *be the endpoints of the intersection (*`M' = M` *and* `N' = N` *are possible).*

If `M'.y > 0` *and* `N'.y > 0`
or `M'.y < 0` *and* `N'.y < 0`
return false
else return true

We want to state the required continuity problem of Algorithm 17.7 more precisely. Let

$$R = \{(m,n,p,q) : m \neq n, p \neq q\} \subset \mathcal{R}^8$$
$$I = \{(m,n,p,q) \in R : MN \text{ intersects } S\}$$
$$T = \{(m,n,p,q) \in R : \text{Algorithm 17.7 returns true}\}$$
$$B = \text{boundary of } T \text{ in } R$$
D be the minimum of:
the distance from `m` to `pq`
the distance from `n` to `pq`
the distance from `p` to `mn`
the distance from `q` to `mn`

Then we want to prove:

Theorem 17.8 *$D = 0$ on B.*

In the proof of this theorem we will use the following:

Lemma 17.9 *If $b \in I$ then:*

`M'(`b`).y = 0` *implies* $D = 0$ *at* b
`N'(`b`).y = 0` *implies* $D = 0$ *at* b

Proof of Lemma:

Let `M'.y = 0`, so `M'` is on the line `PQ` and also on the line `MN`. The cases are:

(a) `M' = M` so `M` is on the line `PQ` and $D = 0$.
(b) `M'.x = P.x` in which case `M' = P` (as `M'.y = 0 = P.y`), so `P` is on `MN` and $D = 0$.
(c) `M'.x = Q.x` in which case `M' = Q` (as `M'.y = 0 = Q.y`), so `Q` is on `MN` and $D = 0$.

Similarly for `N'.y = 0`.

Proof of Theorem:

First note that since $R \setminus I$ and $R \setminus T$ are open, I and T are closed. In addition, `M`, `N`, `P`, and `Q` are continuous functions on R, and `M'` and `N'` are continuous functions on I.

Let $b \in B$. We need to show that $D = 0$ at b.

Since T is closed and B is the boundary of T, $B \subset T$, and $b \in T \subset I$. Therefore $\texttt{M'}(b)$ and $\texttt{N'}(b)$ exist and one of the following is true:

(a) $\texttt{M'}(b)\texttt{.y} = 0$ (and therefore by the lemma $D = 0$ at b)
(b) $\texttt{N'}(b)\texttt{.y} = 0$ (and therefore by the lemma $D = 0$ at b)
(c) $\texttt{M'}(b)\texttt{.y} > 0 > \texttt{N'}(b)\texttt{.y}$
(d) $\texttt{M'}(b)\texttt{.y} < 0 < \texttt{N'}(b)\texttt{.y}$

Note that $\texttt{M'}(b)\texttt{.y} > 0 < \texttt{N'}(b)\texttt{.y}$ is impossible as $b \in T$, and similarly $\texttt{M'}(b)\texttt{.y} < 0 > \texttt{N'}(b)\texttt{.y}$ is impossible.

So we need only prove that (c) and (d) are impossible or lead to the conclusion that $D = 0$ at b.

Since b is on the boundary of T in R, there is an infinite sequence of points in $R \setminus T$ converging to b. Using the fact that $R \setminus T$ is the disjoint union of $R \setminus I$ and $I \setminus T$, we divide the argument into two cases:

Case 1: There is a sequence of points in $I \setminus T$ that converges to b.

Then there is a subsequence s of points such that either:

(e) if $b_s \in s$ then $\texttt{M'}(b_s)\texttt{.y} > 0 < \texttt{N'}(b_s)\texttt{.y}$;
therefore by continuity $\texttt{M'}(b)\texttt{.y} \geq 0 \leq \texttt{N'}(b)\texttt{.y}$;
(f) if $b_s \in s$ then $\texttt{M'}(b_s)\texttt{.y} < 0 > \texttt{N'}(b_s)\texttt{.y}$;
therefore by continuity $\texttt{M'}(b)\texttt{.y} \leq 0 \geq \texttt{N'}(b)\texttt{.y}$;

But (e) is incompatible with both (c) and (d), and similarly (f) is incompatible with both (c) and (d).

Case 2: There is a sequence of points in $R \setminus I$ that converges to b.

Then there is a subsequence s of points such that either:

(g) if $b_s \in s$ then $\texttt{M}(b_s)\texttt{.x} < \texttt{P.x} > \texttt{N}(b_s)\texttt{.x}$;
therefore by continuity $\texttt{M}(b)\texttt{.x} \leq \texttt{P.x} \geq \texttt{N}(b)\texttt{.x}$;
therefore since $b \in T \subset I$, $\texttt{M'}(b)\texttt{.x} = \texttt{P.x} = \texttt{N'}(b)\texttt{.x}$;
(h) if $b_s \in s$ then $\texttt{M}(b_s)\texttt{.x} > \texttt{Q.x} < \texttt{N}(b_s)\texttt{.x}$
therefore by continuity $\texttt{M}(b)\texttt{.x} \geq \texttt{Q.x} \leq \texttt{N}(b)\texttt{.x}$;
therefore since $b \in T \subset I$, $\texttt{M'}(b)\texttt{.x} = \texttt{Q.x} = \texttt{N'}(b)\texttt{.x}$;

If M(b).x $\neq$ N(b).x then either (g) or (h) imply M′(b) = N′(b) which is incompatible with both (c) and (d).

Otherwise (g) implies that M(b).x = N(b).x = P(b).x and either (c) or (d) then imply that P(b) is on the line MN at b and hence $D = 0$ at b.

Similarly (h) implies that M(b).x = N(b).x = Q(b).x and either (c) or (d) then imply that Q(b) is on the line MN at b and hence $D = 0$ at b.

18 Line and Line Calculator

Implement additions to the point and line calculator for the following *operators* and *functions*:

Assume:	`m`, `n`, `p`, `q` are vectors representing points `D`=D is an integer scalar, $-15 \leq D \leq +15$
then:	
`intersecti mnpqD`	returns `true` if the infinite lines `mn` and `pq` intersect (including the case where the lines are identical) and `false` otherwise; if all coordinates are exact multiples of 10^{-D} and have absolute values $\leq 10^{6-D}$
`intersectf mnpqD`	ditto but for finite lines `mn` and `pq`
`overlapf mnpq`	returns the length of the overlap of finite lines `mn` and `pq` assuming the infinite extensions of these lines are identical
`commoni mnpq`	returns the common point of infinite lines `mn` and `pq` assuming these lines are not parallel
`distf mnpq`	returns the distance between finite lines `mn` and `pq` (may be zero if lines intersect)

Sample Input Files:	**00-XXXX-line-vec2d.in**
Sample Output Files:	**00-XXXX-line-vec2d.ftest**
Sample Run File:	**sample-line-vec2d.run**
Submit Run File:	**submit-line-vec2d.run**