Lines and Distance of a Point to a Line

by Dan Sunday

Distance computations are fundamental in computational geometry, and there are well-known formulas for them. Nevertheless, due to differences in object representations, there are alternative solutions to choose from. We will give some of these and indicate the situations they apply to.

Throughout, we need to have a metric for calculating the distance between two points. We assume this is the standard Euclidean metric "L_2 norm" based on the Pythagorean theorem. That is, for an n-dimensional vector $\mathbf{v} = (v_1, v_2, \ldots, v_n)$, its length $|\mathbf{v}|$ is given by:

$$|\mathbf{v}|^2 = v_1^2 + v_2^2 + \cdots + v_n^2 = \sum_{i=1}^{n} v_i^2$$

and for two points $P = (p_1, p_2, \ldots, p_n)$ and $Q = (q_1, q_2, \ldots, q_n)$, the distance between them is:

$$\mathrm{d}(P,Q) = |P - Q| = \sqrt{\sum_{i=1}^{n} (p_i - q_i)^2}$$

Lines

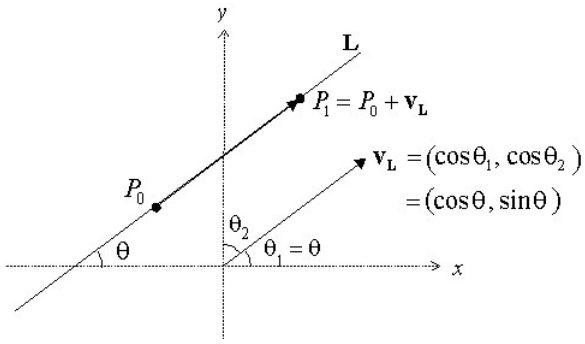
A point is that which has no part. [Book I, Definition 1]
A line is breadthless length. [Book I, Definition 2]
The extremities of a line are points. [Book I, Definition 3]
A straight line is a line which lies evenly with the points on itself. [Book I, Definition 4]

To draw a straight line from any point to any point. [Book I, Postulate 1]
To produce a finite straight line continuously in a straight line. [Book I, Postulate 2]
[Euclid, 300 BC]

The primal way to specify a line **L** is by giving ***two distinct points***, P_0 and P_1, on it. In fact, this defines a finite line segment **S** going from P_0 to P_1 which are the endpoints of **S**. This is how the Greeks understood straight lines, and it coincides with our natural intuition for the most direct and shortest path between the two endpoints. This line can then be extended indefinitely beyond either endpoint producing infinite rays in both directions. When extended simultaneously beyond both ends, one gets the concept of an infinite line which is how we often think of it today. However, for the Greeks, the line was the finite segment which could be extended indefinitely using a straight edge. One nice thing about defining lines this way is that it works for all dimensions: 2D, 3D, or any n-dimensional space. Further, it is common in applications to have lines specified as segments given by their endpoints since finite segments often occur as an edge of a polygon, polyhedron, or an embedded graph.

A line **L** can also be defined by ***a point and a direction***. Let P_0 be a point on **L** and $\mathbf{v}_L$ be a nonzero vector giving the direction of the line. This is equivalent to the two point definition, since we could just put $\mathbf{v}_L = (P^1 - P_0)$. Or, given P_0 and $\mathbf{v}_L$, we could select $P_1 = P_0 + \mathbf{v}_L$ as a second point on the line. If $\mathbf{v}_L$ is "normalized" to be the unit direction vector, $\mathbf{u}_L = \mathbf{v}_L / |\mathbf{v}_L|$, then its components are the direction cosines of **L**. That is, in n-dimensions, let θ_i $(i = 1,n)$ be the angle that **L** makes with the i-th coordinate axis $\mathbf{a}_i$ (for example, in 2D, $\mathbf{a}_1$ is the x-axis and $\mathbf{a}_2$ is the y-axis). Then, the vector $\mathbf{v}_L = (v_i)$, with $v_i = \cos(\theta_i)$, is a direction vector for **L**. In 2D, as shown in the diagram, if θ is the angle **L** makes with the x-axis, then $\cos(\theta_2) = \sin(\theta)$, and $\mathbf{v}_L = (\cos(\theta_1), \cos(\theta_2)) = (\cos(\theta), \sin(\theta))$ is a ***unit direction vector*** for **L**, since $\cos^2(\theta) + \sin^2(\theta) = 1$.

Similarly in any dimension n, the squares of the direction cosines sum to 1, that is $\sum_{i=1}^{n} \cos^2(\theta_i) = 1$.



Line Equations

Lines can also be defined by equations using the coordinates of points on the line as unknowns. The types of equations that one encounters in practice are:

Type	Equation	Usage
Explicit 2D	$y = \mathrm{f}(x) = mx + b$	a non-vertical 2D line
Implicit 2D	$\mathrm{f}(x, y) = ax + by + c = 0$	any 2D line
Parametric	$P(t) = P_0 + t\,\mathbf{v_L}$	any line in any dimension

The 2D ***explicit equation*** is the one most people are first taught in school, but it is not the most flexible one to use in computational software. The ***implicit equation*** is a bit more useful, and the conversion of an explicit to an implicit equation is easy to do. Note that the first two implicit coefficients always define a vector $\mathbf{n}_L = (a,b)$ which is perpendicular to the line **L**. This is because for any two points $P_0 = (x_0, y_0)$ and $P_1 = (x_1, y_1)$ on **L**, we have $\mathbf{n}_L \cdot \mathbf{v}_L = (a,b) \cdot (P_1 - P_0) = a(x_1 - x_0) + b(y_1 - y_0) = \mathrm{f}(P_1) - \mathrm{f}(P_0) = 0$. We say that $\mathbf{n}_L$ is a ***normal vector*** for **L** to mean that it is perpendicular to the line. Further, given any normal vector $\mathbf{n}_L = (a, b)$ to the line **L** and a point P_0 on it, the ***normal form*** of the implicit equation is:

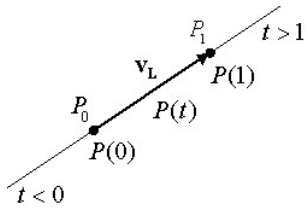
$$\mathbf{n_L} \cdot (P - P_0) = ax + by - \mathbf{n_L} \cdot P_0 = 0$$

This equation is said to be ***normalized*** if $a^2 + b^2 = 1$, and then $\mathbf{n}_L$ is called a ***unit normal vector***, which we often denote as $\mathbf{u}_L$ to emphasize that it has unit 1 length. I know this overuse of the word "normal" may be confusing, but that's the terminology in current use.

Unfortunately, a single implicit (or explicit) equation only defines a line in 2D, whereas in 3D a single linear equation defines a plane and in *n*-dimensions it defines an (*n*-1)-dimensional hyperplane [Hanson, 1994]. This, of course, is useful in its own right, but it is not our interest here. For further information about 3D planes, see the Algorithm 4 discussion about Planes.

On the other hand, in any *n*-dimensional space, the ***parametric equation*** for the line is valid and is the most versatile one to use. For a line defined by two points P_0 and P_1 with a direction vector $\mathbf{v}_L$, the equation can be written several ways; namely:

$$\begin{aligned} P(t) &= P_0 + t\,\mathbf{v_L} \\ &= P_0 + t(P_1 - P_0) \\ &= (1-t)P_0 + tP_1 \end{aligned}$$



where *t* is a real number. With this representation $P(0) = P_0$, $P(1) = P_1$, and $P(t)$ with $0 < t < 1$ is a point on the finite segment between P_0 and P_1 where *t* is the fraction of $P(t)$'s distance along the whole P_0P_1 line segment. That is, $t = \mathrm{d}(P_0, P(t)) / \mathrm{d}(P_0, P_1)$. And, $P(1/2) = (P_0 + P_1)/2$ is the midpoint of the segment. Further, if t < 0 then $P(t)$ is outside the segment on the P_0 side, and if t > 1 then P(t) is outside on the P_1 side.

One can convert from any of these representations to another when convenient. For example, given two points $P_0 = (x_0, y_0)$ and $P_1 = (x_1, y_1)$ on a 2D line, one can derive an implicit equation for it as follows. With $\mathbf{v_L} = (x_v, y_v) = P_1 - P_0 = (x_1 - x_0, y_1 - y_0)$ for the line direction vector, we have $\mathbf{n_L} = (-y_v, x_v) = (y_0 - y_1, x_1 - x_0)$ is a normal vector perpendicular to **L**, since $\mathbf{n}_L \cdot \mathbf{v}_L = 0$. Then, an implicit equation for **L** is:

$$(y_0 - y_1)x + (x_1 - x_0)y + (x_0 y_1 - x_1 y_0) = 0$$

where the coefficients of x and y are the coordinates of $\mathbf{n}_L$.

For another example, in 2D, if a line **L** makes an angle θ with the x-axis, recall that $\mathbf{v_L} = (\cos(\theta), \sin(\theta))$ is a unit direction vector, and thus $\mathbf{n_L} = (-\sin(\theta), \cos(\theta))$ is a unit normal vector. So, if $P_0 = (x_0, y_0)$ is a point on **L**, then a normalized implicit equation for **L** is:

$$-\sin(\theta)x + \cos(\theta)y + (\sin(\theta)x_0 - \cos(\theta)y_0) = 0$$

Further, the parametric line equation is:

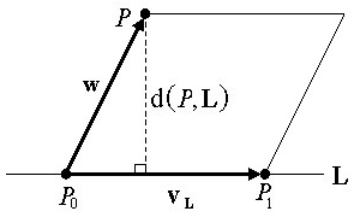
$$P(t) = (x_0 + t\cos\theta, y_0 + t\sin\theta)$$

Distance of a Point to an Infinite Line

Given a line **L** and any point P, let d(P,**L**) denote the distance from P to **L**. This is the shortest distance separating P and **L**. If **L** is an infinite line, then this is the length of a perpendicular dropped from P to **L**. However if **L** is a finite segment **S**, then the base of the perpendicular to the extended line may be outside the segment, and a different determination of the shortest distance needs to be made. We first consider perpendicular distance to an infinite line.

The 2-Point Line

In 2D and 3D, when **L** is given by two points P_0 and P_1, one can use the cross-product to directly compute the distance from any point P to **L**. The 2D case is handled by embedding it in 3D with a third z-coordinate = 0. The key observation to make is that the magnitude of the cross-product of two 3D vectors is equal to the area of the parallelogram spanned by them, since $|\mathbf{v}\times\mathbf{w}| = |\mathbf{v}||\mathbf{w}||\sin(\theta)|$ where θ is the angle between the two vectors **v** and **w**. However, this area is also equal to the magnitude of the base times the height of the parallelogram, and we can arrange the geometry so that the height is the distance d(P,**L**). Let $\mathbf{v_L} = P_1 - P_0$ and $\mathbf{w} = P - P_0$ as in the diagram:



Then, $|\mathbf{v}_L \times \mathbf{w}|$ = Area(parallelogram($\mathbf{v}_L$, **w**)) = $|\mathbf{v}_L|$ d(P,**L**) which results in the easy formula:

$$\mathrm{d}(P,\mathbf{L}) = \frac{|\mathbf{v_L}\times\mathbf{w}|}{|\mathbf{v_L}|} = |\mathbf{u_L}\times\mathbf{w}|$$

where $\mathbf{u}_L = \mathbf{v}_L / |\mathbf{v}_L|$ is the unit direction vector of **L**. If one is computing the distances of many points to a fixed line, then it is most efficient to first calculate $\mathbf{u}_L$.

For the embedded 2D case with $P = (x, y, 0)$, the cross-product becomes:

$$\mathbf{v_L}\times\mathbf{w} = (x_1 - x_0, y_1 - y_0, 0)\times(x - x_0, y - y_0, 0) = \left(0, 0, \begin{vmatrix} (x_1 - x_0) & (y_1 - y_0) \\ (x - x_0) & (y - y_0) \end{vmatrix}\right)$$

and the distance formula is:

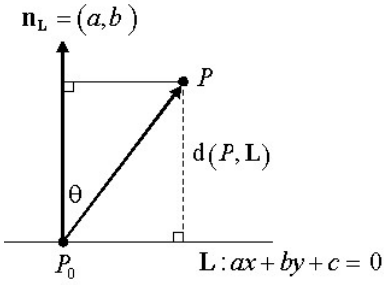
$$\mathrm{d}(P,\mathbf{L}) = \frac{(y_0 - y_1)x + (x_1 - x_0)y + (x_0 y_1 - x_1 y_0)}{\sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2}}$$

We did not take the absolute value of the numerator here making this is a *signed* distance with positive values on one side of **L** and negative distances on the other. This can sometimes be useful. Other times one may want to take the absolute value. Also note the similarity of the numerator and the implicit line equation.

The 2D Implicit Line

In 2D, there are applications where a line **L** is most easily defined by an implicit equation $\mathrm{f}(x,y)=ax+by+c=0$. For any 2D point $P = (x, y)$, the distance d(P,**L**) can be computed directly from this equation.

Recall that the vector $\mathbf{n}_L = (a, b)$ is perpendicular to the line **L**. Using $\mathbf{n}_L$, we can compute the distance of an arbitrary point P to **L** by first selecting any specific point P_0 on **L** and then projecting the vector P_0P onto $\mathbf{n}_L$. as shown in the diagram:



Writing out the details,

(1) since not both a and b are zero, assume $a \neq 0$ and select $P_0 = (-c / a, 0)$ which is on the line [Otherwise, if $a = 0$ then $b \neq 0$, and select $P_0 = (0, -c / b)$ instead, which yields the same final result]

(2) for any P_0 on **L** we have: $\mathbf{n}_\mathbf{L} \bullet P_0P = |\mathbf{n}_\mathbf{L}||P_0P|\cos\theta = |\mathbf{n}_\mathbf{L}|\mathrm{d}(P,\mathbf{L})$

(3) also for our specific P_0: $\mathbf{n}_\mathbf{L} \bullet P_0P = (a,b)\bullet(x+c/a, y) = ax+c+by = \mathrm{f}(x,y) = \mathrm{f}(P)$

and equating (2) and (3) yields the formula:

$$\mathrm{d}(P,\mathbf{L}) = \frac{\mathrm{f}(P)}{|\mathbf{n}_\mathbf{L}|} = \frac{ax+by+c}{\sqrt{a^2+b^2}}$$

Further, one can divide the coefficients of f(x,y) by $|\mathbf{n}_L|$ to prenormalize the implicit equation so that $|\mathbf{n}_L| = 1$. This results in the very efficient formula:

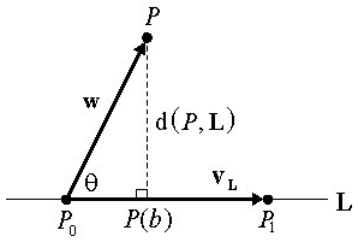
$$\mathrm{d}(P,\mathbf{L}) = \mathrm{f}(P) = ax+by+c \quad \text{when } a^2+b^2=1$$

which has only 2 multiplications and 2 additions for each distance calculation. So, if in 2D one needs to compute the distances of many points to the same infinite line **L**, then one should derive the unit normalized implicit equation and use this formula. Also, if one is just comparing distances (say, to find the closest or farthest point to the line), then normalizing is not even needed since it just changes all computed distances by a constant factor.

Recall that when **L** makes an angle θ with the x-axis and $P_0 = (x_0, y_0)$ is any point on **L**, then the normalized implicit equation has: $a = -\sin(\theta)$, $b = \cos(\theta)$, and $c = x_0 \sin(\theta) - y_0 \cos(\theta)$.

The Parametric Line

To compute the distance d(P,**L**) (in any n-dimensional space) from an arbitrary point P to a line **L** given by a parametric equation, suppose that $P(b)$ is the base of the perpendicular dropped from P to **L**. Let the parametric line equation be given as: $P(t) = P_0 + t\,(P_1 - P_0)$. Then, the vector $P_0P(b)$ is the projection of the vector P_0P onto the segment P_0P_1, as shown in the diagram:



So, with $\mathbf{v}_L = (P_1 - P_0)$ and $\mathbf{w} = (P - P_0)$, we get that:

$$b = \frac{\mathrm{d}(P_0, P(b))}{\mathrm{d}(P_0, P_1)} = \frac{|\mathbf{w}|\cos\theta}{|\mathbf{v}_\mathbf{L}|} = \frac{\mathbf{w}\bullet\mathbf{v}_\mathbf{L}}{|\mathbf{v}_\mathbf{L}|^2} = \frac{\mathbf{w}\bullet\mathbf{v}_\mathbf{L}}{\mathbf{v}_\mathbf{L}\bullet\mathbf{v}_\mathbf{L}}$$

and thus:

$$\mathrm{d}(P,\mathbf{L}) = \left|P - P(b)\right| = \left|\mathbf{w} - b\,\mathbf{v}_{\mathbf{L}}\right| = \left|\mathbf{w} - (\mathbf{w}\bullet\mathbf{u}_{\mathbf{L}})\,\mathbf{u}_{\mathbf{L}}\right|$$

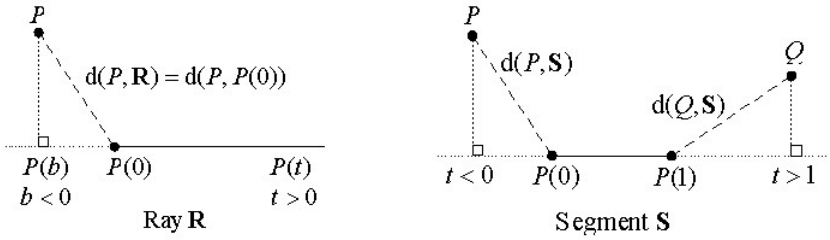
where $\mathbf{u}_{\mathrm{L}}$ is our friend the unit direction vector of **L**.

This computation has the advantage of working for any dimension *n* and of also computing the base point *P*(*b*) which is sometimes useful. In 3D, it is just as efficient as the cross product formula. But in 2D, when *P*(*b*) is not needed, the implicit method is better, especially if one is computing the distances of many points to the same line.

Distance of a Point to a Ray or Segment

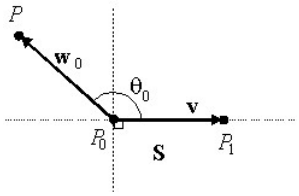
A **ray R** is a half line originating at a point P_0 and extending indefinitely in some direction. It can be expressed parametrically as *P*(*t*) for all $t \geq 0$ with *P*(0) = P_0 as the starting point. A ***finite segment*** **S** consists of the points of a line that are between two endpoints P_0 and P_1. Again, it can be represented by a parametric equation with *P*(0) = P_0 and *P*(1) = P_1 as the endpoints and the points *P*(*t*) for $0 \leq t \leq 1$ as the segment points.

The thing that is different about computing distances of a point *P* to a ray or a segment is that the base *P*(*b*) of the perpendicular from *P* to the extended line **L** may be outside the range of the ray or segment. In this case, the actual shortest distance is from the point *P* to the start point of the ray or one of the endpoints of a finite segment.

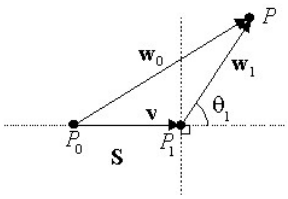


For a ray there is only one choice, but for a segment one must determine which end of the segment is closest to *P*. One could just compute both distances and use the shortest, but this is not very efficient. Also, one must first determine that *P*'s perpendicular base is actually outside the segment's range. An easy way to do this is to consider the angles between the segment P_0P_1 and the vectors P_0P and P_1P from the segment endpoints to *P*. If either of these angles is 90º, then the corresponding endpoint is the perpendicular base *P*(*b*). If the angle is not a right angle, then the base lies to one side or the other of the endpoint according to whether the angle is acute or obtuse. These conditions are easily tested by computing the dot product of the vectors involved and testing whether it is positive, negative, or zero. The result determines if the distance should be computed to one of the points P_0 or P_1, or as the perpendicular distance to the line **L** itself. This technique, which works in any *n*-dimensional space, is illustrated in the diagrams:

$$\mathbf{w}_0 = P - P_0 \text{ and } \theta_0 \in [-180°, 180°]$$
$$\mathbf{w}_0 \bullet \mathbf{v} \leq 0$$
$$\Leftrightarrow |\theta_0| \geq 90°$$
$$\Leftrightarrow \mathrm{d}(P,\mathbf{S}) = \mathrm{d}(P, P_0)$$



$$\mathbf{w}_1 = P - P_1 \text{ and } \theta_1 \in [-180°, 180°]$$
$$\mathbf{w}_1 \bullet \mathbf{v} \geq 0 \Leftrightarrow \mathbf{w}_0 \bullet \mathbf{v} \geq \mathbf{v} \bullet \mathbf{v}$$
$$\Leftrightarrow |\theta_1| \leq 90°$$
$$\Leftrightarrow \mathrm{d}(P,\mathbf{S}) = \mathrm{d}(P, P_1)$$



Since $\mathbf{w}_0 = \mathbf{v} + \mathbf{w}_1$, the two tests can be done just using the two dot products $\mathbf{w}_0 \bullet \mathbf{v}$ and $\mathbf{v} \bullet \mathbf{v}$ which are also the numerator and denominator of the formula to find the parametric base of the perpendicular from *P* to the extended line **L** of the segment **S**. This lets us streamline the algorithm as shown in the pseudo code:

```
distance( Point P,  Segment P0:P1 )
{
        v = P1 - P0
```

```
                    w = P - P0

                    if ( (c1 = w·v) <= 0 )  // before P0
                            return d(P, P0)
                    if ( (c2 = v·v) <= c1 ) // after P1
                            return d(P, P1)

                    b = c1 / c2
                    Pb = P0 + bv
                    return d(P, Pb)
                }
```

Implementations

Here are a few sample "C++" applications using these algorithms. We assume that the low level classes and functions are already given.

```
// Copyright 2001 softSurfer, 2012 Dan Sunday
// This code may be freely used, distributed and modified for any purpose
// providing that this copyright notice is included with it.
// SoftSurfer makes no warranty for this code, and cannot be held
// liable for any real or imagined damage resulting from its use.
// Users of this code must verify correctness for their application.


// Assume that classes are already given for the objects:
//     Point and Vector with
//          coordinates {float x, y, z;} (z=0  for 2D)
//          appropriate operators for:
//               Point  = Point ± Vector
//               Vector = Point - Point
//               Vector = Scalar * Vector
//     Line with defining endpoints {Point P0, P1;}
//     Segment with defining endpoints {Point P0, P1;}
//===================================================================

// dot product (3D) which allows vector operations in arguments
#define dot(u,v)   ((u).x * (v).x + (u).y * (v).y + (u).z * (v).z)
#define norm(v)     sqrt(dot(v,v))     // norm = length of  vector
#define d(u,v)      norm(u-v)          // distance = norm of difference



// closest2D_Point_to_Line(): find the closest 2D Point to a Line
//     Input:  an array P[] of n points, and a Line L
//     Return: the index i of the Point P[i] closest to L
int
closest2D_Point_to_Line( Point P[], int n, Line L)
{
     // Get coefficients of the implicit line equation.
     // Do NOT normalize since scaling by a constant
     // is irrelevant for just comparing distances.
     float a = L.P0.y - L.P1.y;
     float b = L.P1.x - L.P0.x;
     float c = L.P0.x * L.P1.y - L.P1.x * L.P0.y;

     // initialize min index and distance to P[0]
     int mi = 0;
     float min = a * P[0].x + b * P[0].y + c;
     if (min < 0) min = -min;     // absolute value

     // loop through Point array testing for min distance to L
     for (i=1; i<n; i++) {
          // just use dist squared (sqrt not  needed for comparison)
          float dist = a * P[i].x + b * P[i].y  + c;
          if (dist < 0) dist = -dist;    // absolute value
          if (dist < min) {      // this point is closer
               mi = i;              // so have a new minimum
               min = dist;
          }
     }
     return mi;     // the index of the closest  Point P[mi]
}
//===================================================================


// dist_Point_to_Line(): get the distance of a point to a line
//     Input:  a Point P and a Line L (in any dimension)
//     Return: the shortest distance from P to L
float
dist_Point_to_Line( Point P, Line L)
{
```

```
     Vector v = L.P1 - L.P0;
     Vector w = P - L.P0;

     double c1 = dot(w,v);
     double c2 = dot(v,v);
     double b = c1 / c2;

     Point Pb = L.P0 + b * v;
     return d(P, Pb);
}
//===================================================================


// dist_Point_to_Segment(): get the distance of a point to a segment
//     Input:  a Point P and a Segment S (in any dimension)
//     Return: the shortest distance from P to S
float
dist_Point_to_Segment( Point P, Segment S)
{
     Vector v = S.P1 - S.P0;
     Vector w = P - S.P0;

     double c1 = dot(w,v);
     if ( c1 <= 0 )
          return d(P, S.P0);

     double c2 = dot(v,v);
     if ( c2 <= c1 )
          return d(P, S.P1);

     double b = c1 / c2;
     Point Pb = S.P0 + b * v;
     return d(P, Pb);
}
//===================================================================
```

References

David Eberly, "Distance Between Point and Line, Ray, or Line Segment", Geometric Tools (2002)

Euclid, The Elements, Alexandria (300 BC)

Andrew Hanson, "Geometry for N-Dimensional Graphics" in Graphics Gems IV (1994)

Thomas Heath, The Thirteen Books of Euclid's Elements, Vol 1 (Books I and II) (1956)

Jack Morrison, "The Distance from a Point to a Line", in Graphics Gems II (1994)