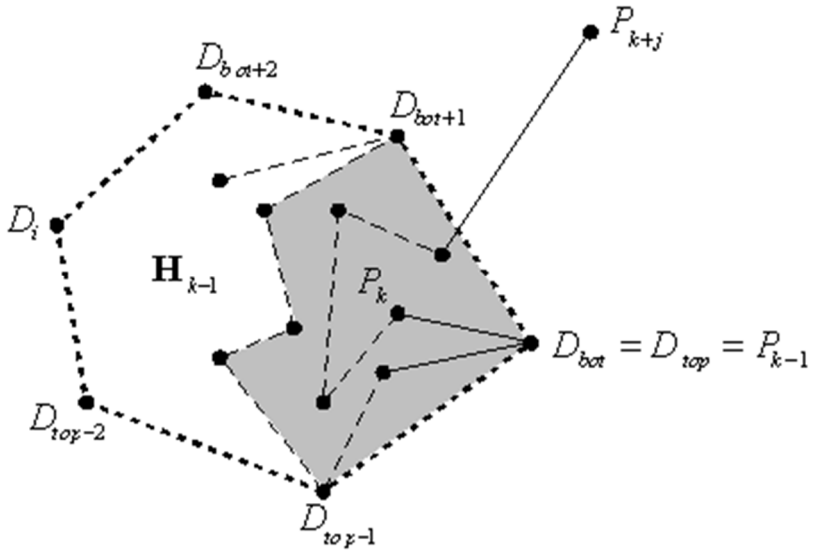
Practical Geometry Algorithms

with C++ Code

Daniel Sunday PhD

PRACTICAL GEOMETRY ALGORITHMS

with C++ Code



Daniel Sunday PhD

ISBN 9798749449730

Self-published with Amazon KDP
First Printing: May 2021.

TABLE OF CONTENTS

PREFACE

This book presents practical geometry algorithms with computationally fast C++ code implementations. It covers algorithms for fundamental geometric objects, such as points, lines, rays, segments, triangles, polygons, and planes. These determine their basic 2D and 3D properties, such as area, distance, inclusion, and intersections. There are also algorithms to compute bounding containers for these objects, including a fast bounding ball, various convex hull algorithms, as well as polygon extreme points and tangents. And there is a fast algorithm for polyline simplification using decimation that works in any dimension.

These algorithms have been used in practice for several decades, and are robust, easy to understand, code, and maintain. And they execute very rapidly in practice, not just in theory. For example, the winding number point in polygon inclusion test, first developed by the author in 2000, is the fastest inclusion algorithm known, and works correctly even for nonsimple polygons. There is also a fast implementation of the Melkman algorithm for the convex hull of a simple polyline. And much more.

If your programming involves geometry, this will be an invaluable reference.

Also, all code given in this book can be downloaded in a zip file available at GeometryAlgorithms.com/code.html.

INTRODUCTION

In 2000, I started the website GeometryAlgorithms.com, which has now been active for 2 decades. Originally, I used it to support master-level courses in Computer Graphics and Computational Geometry that I was teaching in the Johns Hopkins University Engineering for Professionals (JHEP) program. Eventually, that website took on a life of its own and became an international resource with a constant stream of visitors. All the algorithms presented were completed by 2001, although for the next two decades, there was some feedback from users that resulted in improvements. So, basically, these algorithms and the C++ code implementations have withstood the test of time. As a result, I have now (in 2021), decided to convert that website to a book that would be a permanent record of these fundamental algorithms. The website still exists but has reduced content. And all C++ code from that site is in this book can be downloaded from GeometryAlgorithms.com/code.html.

Many of the algorithms given here were inspired by the excellent book "Computational Geometry in C" by Joseph O'Rourke (1998), which I used in my classes. This book can be viewed as a supplement for and an extension of O'Rourke's book, with new improved material. For example, O'Rourke presents a "point in polygon" algorithm that uses the polygon crossing number of a ray from the point. We present a similar algorithm but use the winding number instead by counting signed crossings. This gives the correct result for nonsimple polygons and is implemented with code that executes faster than the crossing number code. Other algorithms we present either extend or improve some of those given by O'Rourke. However, O'Rourke's book has material that we do not cover, such as the triangulation of polygons. But we present a lot of new material, such as bounding containers, the convex hull of a simple polyline, and polyline decimation. These all have practical fast C++ code implementations.

This book starts with a chapter on basic linear algebra that provides the math needed to implement the algorithms. It was not relegated to an appendix since the ideas and notation are integral to the algorithm descriptions. Then, the next eight chapters cover basic geometric objects in 2D and 3D, such as lines, rays, segments, triangles, polygons, and planes. Then, the final eight chapters give algorithms for geometric bounding containers, convex hulls, extreme points and tangents of polygons, and polyline reduction using decimation. There are six chapters are about convex hulls and applications that use hulls to find extreme points and tangents. New algorithms and code are given for constructing hulls, namely, Andrew's monotone chain, the fast BFP hull approximation, and the Melkman O(n) algorithm for the hull of a simple polyline. Finally, we give a polyline decimation approximation algorithm, with an improved implementation for the Douglas-Peucker algorithm.

LINEAR ALGEBRA

The algorithms we will present only use basic linear algebra. This will only include points and vectors, but not matrices. However, we occasionally use 2x2 and 3x3 determinants to simplify some formulas.

Notation

This book uses the following conventions for notation.

Scalars	*lower case italic*	a, b, x, y
Points	*UPPER CASE ITALIC*	P, Q, R $P = (p_1, p_2, \dots, p_n)$
Vectors	**lower case bold**	$\mathbf{a}, \mathbf{b}, \mathbf{u}, \mathbf{v}, \mathbf{w}$ $\mathbf{v} = (v_1, v_2, \dots, v_n)$
Geometric Objects	**UPPER CASE BOLD**	$\mathbf{L}, \mathbf{R}, \mathbf{S}, \mathbf{T},$ $\mathbf{\Delta}, \mathbf{\Omega}, \mathbf{\Psi}$
Functions	Regular font	Area($\mathbf{\Omega}$), f(x)
	Font of the result	$P(t) = \big(x(t), y(t)\big)$ $\mathbf{v} = (x, y)$

Coordinate Systems

Points are positions in space. When first introduced in Greek geometry, there was no formal method for measuring where a point was located. Points were primitive entities. Much later in history, in the early seventeenth century, Fermat and Descartes introduced the idea of using a linear coordinate system to specify point locations, using algebraic equations to describe geometric objects (such as lines), and using algebra to solve geometric problems (such as computing the intersection point of two lines). We will only use rectangular linear coordinates, and no other methods for referencing points, such as polar or spherical coordinates.

A (rectangular linear) **coordinate system** has an **origin** as an absolute reference point whose location is fixed, and non-zero **axes** as a set of direction vectors that determine the directions in which to make measurements. A coordinate system can specify points in *n*-dimensional space as an ordered *n*-tuple of numbers $(x_1, x_2, \ldots, x_n)$ and the numeric rule: start at the origin, go distance x_1 in the direction of the first axis, stop, now go distance x_2 in the direction of the second axis, stop, and so on until done with x_n. The final location that one reaches is the point specified. This is much like finding the treasure on a pirate's map; for example, start at the old oak tree as the origin, first go east 10 paces, second go north 20 paces, and third dig down 3 feet to locate the treasure. The ordered set of numbers $(x_1, x_2, \ldots, x_n)$ is called the **coordinate** of the point it specifies. And a coordinate system is said to span the set of points that it can specify. The set of all these points is called the **coordinate space**. A specific set of axes spanning a coordinate space is called a **coordinate frame of reference** or a **basis** for the space. Different coordinate systems can span the same coordinate space, just as the pirate may have used any of several origin trees or direction axes for his treasure map.

The axes are said to be **independent** if they do not depend on each other, meaning that none of them can written as a combination of the others. When independent, they are a minimal set of axes for the coordinate space, and the number of axes in the set is the dimension of the space.

Points and Vectors

Basic Definitions

A **scalar** represents magnitude and is given by a real number *a, b, x, y*.

A **point** in *n*-dimensional space is given by an *n*-tuple $P = (p_1, p_2, \ldots, p_n)$ where each coordinate p_i is a scalar number. We will write $P = (p_i)$ as a shorthand for this *n*-tuple. This position of a point is relative to a coordinate system with an origin $\mathbf{0}$ = (0,0,…,0) and unit axes $\mathbf{u}_1$ = (1,0,…,0), $\mathbf{u}_2$ = (0,1,…,0), ..., and $\mathbf{u}_n$ = (0,0,…,0,1). So, a 3-dimensional (3D) point is given by a triple $P = (p_1, p_2, p_3)$ whose coordinates are relative to the axes (1,0,0), (0,1,0), and (0,0,1) which are commonly referred to as the *x, y*, and *z*-axes. Because of this established convention, we sometimes write $P = (x, y)$ for 2D points and $P = (x, y, z)$ for 3D points.

A **vector** represents magnitude and direction in space and is given by an *n*-tuple $\mathbf{v} = (v_1, v_2, \ldots, v_n)$ where each coordinate v_i is a scalar. We also write $\mathbf{v} = (v_i)$ as a shorthand for the vector's *n*-tuple. The vector $\mathbf{v}$ can be viewed as the magnitude and direction of the line segment going from the origin $\mathbf{0}$ = (0,0,…,0) to the point $V = (v_1, v_2, \ldots, v_n)$. However, the vector $\mathbf{v}$ is not this point, which only gives a standard way of visualizing the vector. But we can also visualize the vector as a directed line segment from any initial point $P = (p_i)$ to a second point $Q = (q_i)$. Then, the vector from P to Q is given by:

Point Difference Definition	$\mathbf{v} = Q - P = (q_i - p_i)$

showing that the difference of any two points is considered to be a vector. So, vectors do not have a fixed position in space, but can be located at any initial base point *P*. For example, a traveling vehicle can be said to be going east (direction) at 50 mph (magnitude) no matter where it is located. We could even visualize a field of vectors, one at each point of a geometric object, such as vectors for the wind direction and speed at each point on the surface of the earth.

Vector Addition

The **sum** of two vectors is given by adding their corresponding coordinates. That is, for two vectors $\mathbf{v} = (v_i)$ and $\mathbf{w} = (w_i)$, we have:

Vector Addition Definition	$\mathbf{v} \pm \mathbf{w} = (v_i \pm w_i)$

Also, one can add a vector $\mathbf{v} = (v_i)$ to a point $P = (p_i)$ to get another point, given by:

Point+Vector Addition Definition	$Q = P + \mathbf{v} = (p_i + v_i)$

The resulting point Q is considered to be the displacement, or "translation", of the point P in the direction of and by the magnitude of the vector $\mathbf{v} = Q - P$.

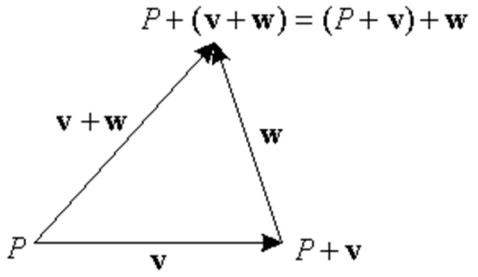
Vector addition satisfies the following properties:

$\mathbf{u} + (\mathbf{v} + \mathbf{w}) = (\mathbf{u} + \mathbf{v}) + \mathbf{w}$ [Association]

$\mathbf{v} + \mathbf{w} = \mathbf{w} + \mathbf{v}$ [Commutation]

$P + (\mathbf{v} + \mathbf{w}) = (P + \mathbf{v}) + \mathbf{w}$ [Point+Vector Association]

The property of "Point+Vector Association" means that result of translating a point by two sequentially applied vector displacements is the same as a single translation by the sum of those two vectors.



Scalar Multiplication

Multiplication of a vector $\mathbf{v} = (v_i)$ by a scalar number a is given by

Scalar Multiply Definition	$a\,\mathbf{v} = (a\,v_i)$

This represents scaling the size of a vector by a magnification factor of a. So, for example, $2\mathbf{v}$ is twice the size of $\mathbf{v}$, and $\mathbf{v}/2$ is half of $\mathbf{v}$.

Scalar multiplication has the properties:

$a(b\mathbf{v}) = (ab)\mathbf{v}$ [Scalar Association]

$(a \pm b)\mathbf{v} = a\mathbf{v} \pm b\mathbf{v}$ [Scalar Distribution]

$a(\mathbf{v} \pm \mathbf{w}) = a\mathbf{v} \pm a\mathbf{w}$ [Vector Distribution]

Scalar multiplication can be used to interpolate positions between two points P and Q. To get an intermediate point R between P and Q, given by a ratio r, one first scales the vector $\mathbf{v} = Q - P$ to $r\mathbf{v}$, and then adds it to P to get $R = P + r\,\mathbf{v} = P + r(Q - P) = (1 - r)P + r\,Q$. For example, to get the midpoint M between P and Q, use $r = 1/2$ to compute $M = (P + Q)/2$.

This interpolation equation is also used to represent the line through P and Q as a function of a scalar parameter, with the ***parametric line equation***

$$P(t) = (1 - t)P + tQ$$

which is valid because one can use "affine addition" to combine the points in a coordinate-free manner.

Affine Addition

We have already seen that the difference between two points can be considered as a vector. However, in general, it makes no sense to add two points together. Points denote an absolute position in space independent of any coordinate system describing them. Blindly adding individual coordinates together would give different answers for different coordinate reference frames.

Nevertheless, there is one special case, known as **affine addition,** where one can add points together as a weighted sum. In fact, in the previous section on scalar multiplication, we did just that to represent the points on a line going through two fixed points P and Q. More generally, given m points $P_0, \ldots, P_{m-1}$, one can define:

Affine Sum Definition	$P = \sum_{i=0}^{m-1} a_i P_i$ where $\sum_{i=0}^{m-1} a_i = 1$

where the coefficients, which can be any real number (positive or negative), must add up to 1. One can interpret this sum as the center of mass for weights a_i located at the points P_i. A negative coefficient can be thought of as a negative mass, like a helium balloon. This center of mass is uniquely determined in absolute space regardless of what coordinate system and frame of reference are being used.

The affine sum has the property:

$$P + \mathbf{v} = (\textstyle\sum_i a_i P_i) + \mathbf{v} = \sum_i a_i (P_i + \mathbf{v}) \qquad \text{[Translation Invariance]}$$

Since the line equation is an affine sum, given equal weights $a_0 = a_1 = 1/2$, we get that $P(1/2) = \frac{1}{2}P_0 + \frac{1}{2}P_1 = (P_0 + P_1)/2$ is the midpoint of the line segment from P_0 to P_1. Further, every point on the line through P_0 and P_1 is uniquely represented by a pair of numbers (a_0, a_1) with $a_0 + a_1 = 1$. Putting $a_1 = t$ results in the ***parametric line equation***:

$$P(t) = (1 - t)P_0 + tP_1$$

where each point on the line is represented by a unique real number t.

Similarly, in 3D space, the affine sum of three non-collinear points $P_0,\ P_1, P_2$ defines a point in the plane going through these points. So, every point P in the plane of the triangle $\Delta P_0P_1P_2$ is uniquely represented by a triple (a_0, a_1, a_2) with $a_0 + a_1 + a_2 = 1$. This triple is called the ***barycentric coordinate*** relative to $\Delta P_0P_1P_2$. of its associated point $P = a_0P_0 + a_1P_1 + a_2\ P_2$ on the plane.

Setting $a_1 = s$ and $a_2 = t$, then $a_0 = 1 - s - t$, and we get the following ***parametric plane equation***.

$$P(s, t) = (1 - s - t)P_0 + sP_1 + tP_2 = P_0 + s\mathbf{u} + t\mathbf{v}$$

where $\mathbf{u} = P_1 - P_0$ and $\mathbf{v} = P_2 - P_0$ are independent vectors spanning the plane. The pair (s, t) is called the ***parametric coordinate*** of a point P relative to $\mathbf{\Delta}P_0P_1P_2$, and there is a unique parametric coordinate for each point of the plane.

It is good to know that the parametric line and plane equations are valid in any n-dimensional space, and they can represent any linear or planar subspace. Similarly, any m-dimensional subspace $(0 < m < n)$ can be represented by a parametric equation with one base point P_0 and m independent vectors $\mathbf{v_i}$ that span the subspace.

Vector Length

The **length** of a vector $\mathbf{v}$ is denoted by $|\mathbf{v}|$, and is defined as

Vector Length Definition	$\|\mathbf{v}\|^2 = \sum_{i=1}^{n} {v_i}^2$

This gives the standard Euclidean geometry (Pythagorean) length for a line segment representing the vector. For a 2D vector $\mathbf{v} = (v_1, v_2)$, one has $|\mathbf{v}|^2 = {v_1}^2 + {v_2}^2$, which is the Pythagorean theorem for the diagonal of a rectangle.

Vector length has the following properties:

$\|a\mathbf{v}\| = \|a\|\|\mathbf{v}\|$	[Scalar magnification]
$\|\mathbf{v} + \mathbf{w}\| \leq \|\mathbf{v}\| + \|\mathbf{w}\|$	[Triangle Inequality]
$\|\mathbf{v} - \mathbf{w}\| \geq \|\mathbf{v}\| - \|\mathbf{w}\|$	[Reverse Triangle Inequality]
$\|\mathbf{v} \pm \mathbf{w}\|^2 = \|\mathbf{v}\|^2 \pm 2\|\mathbf{v}\|\|\mathbf{w}\|\cos(\theta) + \|\mathbf{w}\|^2$ where $\theta = \angle(\mathbf{v}, \mathbf{w})$	[The Cosine Law]

A **unit vector** is one whose length = 1. One can scale any vector $\mathbf{v}$ to get a unit vector $\mathbf{u}$ that points in the same direction as $\mathbf{v}$ by computing $\mathbf{u} = \mathbf{v}/|\mathbf{v}|$. The process of scaling $\mathbf{v}$ to a unit vector $\mathbf{u}$ is called ***normalization***, and one says that $\mathbf{v}$ has been ***normalized***. One can think of $\mathbf{u}$ as the direction of $\mathbf{v}$ since $\mathbf{v} = |\mathbf{v}|\mathbf{u}$ simply scales $\mathbf{v}$ to the magnitude $|\mathbf{v}|$.

Vector Operations

Determinants

In most linear algebra texts, determinants are introduced in the context of matrices. But historically determinants appeared long before matrices. Initially, they were used to solve simultaneous linear equations. Eventually, in the 18th and 19th centuries, they evolved into matrix theory. We do not use matrices explicitly in our algorithms, so only describe the notation for determinants and how to compute their value. A determinant is represented as a 2-dimensional square array of variables or numbers with *n* rows and *n* columns surrounded by vertical bars, from which a scalar quantity gets computed. In effect, the determinant is a function mapping the array to a single scalar value. Originally, the array elements were the coefficients for a collection of *n* linear equations in *n* variables, and the determinant was used in solving those equations. When matrices are introduced, an $n \times n$ matrix array M defines a linear transformation M(**v**) of *n*-dimensional vectors. Then, the determinant of M gives the amount by which M expands or contracts the volume of a transformed region.

We will only use the determinants for *n* = 2 or 3. They are defined as follows.

2x2 Determinant	$\begin{vmatrix} a & b \\ c & d \end{vmatrix} = \text{ad} - \text{bc}$

3x3 Determinant	$\begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = a\begin{vmatrix} e & f \\ h & i \end{vmatrix} - b\begin{vmatrix} d & f \\ g & i \end{vmatrix} + c\begin{vmatrix} d & e \\ g & h \end{vmatrix}$ $= aei + bfg + cdh - ceg - bdi - afh$

The Dot Product

The **dot product (**aka **inner product** or **scalar product**) of two vectors $\mathbf{v} = (v_i)$ and $\mathbf{w} = (w_i)$, is defined as the (scalar) number given by the sum of the products of their corresponding coordinates. This operation is denoted by a dot • and defined by

Dot Product Definition	$\mathbf{v} \bullet \mathbf{w} = \sum_{i=1}^{n} v_i \, w_i$

For example, if $\mathbf{v} = (v_1, v_2)$ and $\mathbf{w} = (w_1, w_2)$ are 2D vectors, then $\mathbf{v} \bullet \mathbf{w} = v_1 w_1 + v_2 w_2$.

The dot product has the following properties.

$\mathbf{v} \bullet \mathbf{v} = \lvert\mathbf{v}\rvert^2$	[Vector Length]
$(a\mathbf{v}) \bullet (b\mathbf{w}) = (ab)(\mathbf{v} \bullet \mathbf{w})$	[Scalar Association]
$\mathbf{v} \bullet \mathbf{w} = \mathbf{w} \bullet \mathbf{v}$	[Commutation]
$\mathbf{u} \bullet (\mathbf{v} + \mathbf{w}) = (\mathbf{u} \bullet \mathbf{v}) + (\mathbf{u} \bullet \mathbf{w})$	[Additive Distribution]
$\lvert\mathbf{v} \bullet \mathbf{w}\rvert \leq \lvert\mathbf{v}\rvert\lvert\mathbf{w}\rvert$	[Cauchy-Schwarz Inequality]

An amazing mathematical formula for the dot product is:

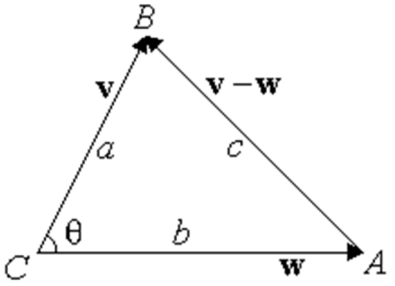
$$\mathbf{v} \bullet \mathbf{w} = \lvert\mathbf{v}\rvert\lvert\mathbf{w}\rvert \cos(\theta)$$

where $\theta = \angle(\mathbf{v}, \mathbf{w})$ is the angle between the vectors $\mathbf{v}$ and $\mathbf{w}$. This formula is used extensively in computer graphics since it speeds up computation in many situations by avoiding direct usage of an inefficient trigonometric function. Further, to compute $\cos(\theta)$ of the angle between two vectors, the following product of two normalized unit vectors is convenient:

$$\cos(\theta) = \frac{\mathbf{v}}{\lvert\mathbf{v}\rvert} \bullet \frac{\mathbf{w}}{\lvert\mathbf{w}\rvert}$$

It is useful to note that when $\lvert\mathbf{v}\rvert = \lvert\mathbf{w}\rvert = 1$, then $\cos(\theta) = \mathbf{v} \bullet \mathbf{w}$.

Interestingly, the dot product cosine formula is equivalent to the well-known trigonometric identity known as "The Cosine Law". Say we have a triangle with vertices *A*, *B*, *C*, and the side opposite each vertex has length a, b, c respectively. If we know a, b, and the angle θ between them, as shown in the diagram, then we can calculate c. Define the vectors $\mathbf{v} = B - C$ and $\mathbf{w} = A - C$, corresponding to *a* and *b*. Then, the side *c* corresponds to the vector $\mathbf{v} - \mathbf{w} = B - A$, and we can calculate:

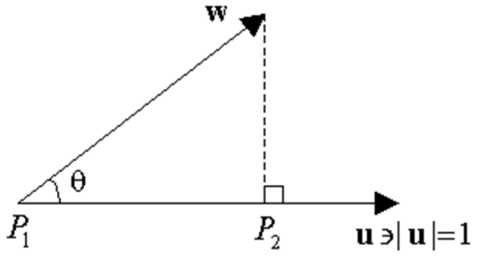


$$\begin{aligned} c^2 &= \lvert\mathbf{v} - \mathbf{w}\rvert^2 = \mathbf{v} \bullet \mathbf{v} - 2(\mathbf{v} \bullet \mathbf{w}) + \mathbf{w} \bullet \mathbf{w} \\ &= \lvert\mathbf{v}\rvert^2 + \lvert\mathbf{w}\rvert^2 - 2\lvert\mathbf{v}\rvert\lvert\mathbf{w}\rvert \cos(\theta) \\ &= a^2 + b^2 - 2ab\cos(\theta) \end{aligned}$$

Similarly, given the Cosine Law, one can derive the vector dot product cosine formula. Amazingly, the early Greeks knew the Cosine Law [Euclid's Elements, Book 2, Props 12 and 13], even though they did not have algebra and trigonometry at the time. For the acute angle case, it was stated in purely geometric terms by

Proposition 13. In acute-angled triangles, the square on the side opposite the acute angle is less than the sum of the squares on the sides containing the acute angle by twice the rectangle contained by one of the sides about the acute angle, namely that on which the perpendicular falls, and the straight line cut off within by the perpendicular towards the acute angle.

Another useful fact about the dot product cosine formula (implicitly used in Euclid's Prop 2-13) is that it can be interpreted geometrically as the projection of one vector onto the other. So, if $\mathbf{u}$ is a unit vector, then $\mathbf{u} \bullet \mathbf{w}$ is the length of the perpendicular projection of $\mathbf{w}$ onto $\mathbf{u}$, as shown in this diagram.

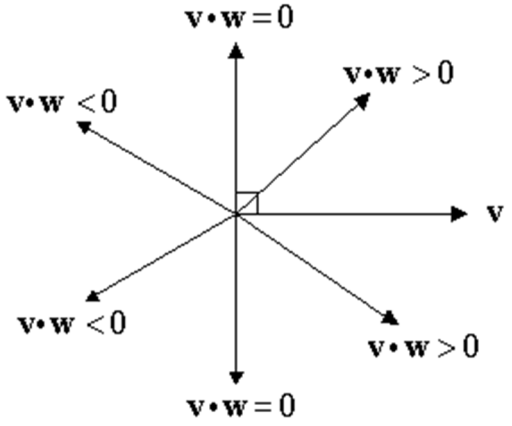


$$|P_2 - P_1| = |\mathbf{w}| \cos(\theta) = \mathbf{u} \bullet \mathbf{w}$$

Additionally, when two vectors $\mathbf{v}$ and $\mathbf{w}$ are perpendicular, they are said to be ***normal*** to each other, and this is equivalent to their dot product being zero, that is, $\mathbf{v} \bullet \mathbf{w} = 0$. This is a remarkably simple and efficient perpendicularity test. Because of this, for any vector $\mathbf{v} = (v_1, v_2, \ldots, v_n)$ one can easily construct perpendicular vectors by zeroing all components except 2, flipping those two, and reversing the sign of one of them, for example, $(-v_2, v_1, 0, \ldots 0)$, $(0, -v_3, v_2, 0, \ldots, 0)$, etc. The dot product of any of these with the original vector $\mathbf{v}$ is always $= 0$, and when these vectors are nonzero, they are all perpendicular to $\mathbf{v}$. For example, in 3D space with $\mathbf{v} = (v_1, v_2, v_3)$, the two vectors $\mathbf{w_1} = (-v_2, v_1, 0)$, and $\mathbf{w_2} = (0, -v_3, v_2)$, when nonzero, are a basis for the unique plane through the origin and perpendicular to $\mathbf{v}$.

Another important and useful consequence of the dot product formula is that, for $|\theta| \leq 180°$, we know the following.

$\mathbf{v} \bullet \mathbf{w} = 0 \Leftrightarrow \lvert\theta\rvert = 90°$	They are perpendicular
$\mathbf{v} \bullet \mathbf{w} > 0 \Leftrightarrow \lvert\theta\rvert < 90°$	θ is an acute angle
$\mathbf{v} \bullet \mathbf{w} < 0 \Leftrightarrow \lvert\theta\rvert > 90°$	θ is an obtuse angle

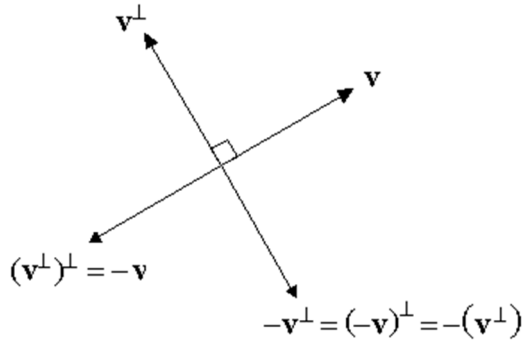


The 2D Perp Operator

We can define an operator on the 2D plane that gives a counterclockwise (ccw) normal (i.e., perpendicular) vector of **v** to be

2D Perp Operator Definition	$\mathbf{v}^{\perp} = (v_1, v_2)^{\perp} = (-v_2, v_1)$

This operator is called the **perp operator.** The **perp vector** $\mathbf{v}^{\perp}$ is the normal vector pointing to the left (ccw) side of the vector **v** as shown in the diagram.



The **perp operator** has the following properties.

$\mathbf{v}^{\perp} \bullet \mathbf{v} = 0$	[Perpendicular]
$\lvert\mathbf{v}^{\perp}\rvert = \lvert\mathbf{v}\rvert$	[Preserves length]

$(a\mathbf{v})^{\perp} = a(\mathbf{v}^{\perp}) = a\mathbf{v}^{\perp}$ [Scalar Association]

$(a\mathbf{v} + b\mathbf{w})^{\perp} = a\mathbf{v}^{\perp} + b\mathbf{w}^{\perp}$ [Linear]

$\mathbf{v}^{\perp\perp} = (\mathbf{v}^{\perp})^{\perp} = -\mathbf{v}$ [Anti-potent]

The 2D Perp Product

Also, in 2D space, there is another useful scalar product of two vectors **v** and **w**, the **perp product** (aka the **2D exterior product** or **outer product)**, which is denoted with a $\perp$, and defined by

2D Perp Product Definition	$\mathbf{v} \perp \mathbf{w} = \mathbf{v}^{\perp} \bullet \mathbf{w} = v_1 w_2 - v_2 w_1 = \begin{vmatrix} v_1 & v_2 \\ w_1 & w_2 \end{vmatrix}$

The perp product has the following properties.

$\mathbf{v} \perp \mathbf{v} = 0$ [Nilpotent]

$(a\mathbf{v}) \perp (b\mathbf{w}) = (ab)(\mathbf{v} \perp \mathbf{w})$ [Scalar Association]

$\mathbf{v} \perp \mathbf{w} = -(\mathbf{w} \perp \mathbf{v})$ [Antisymmetric]

$\mathbf{u} \perp (\mathbf{v} + \mathbf{w}) = (\mathbf{u} \perp \mathbf{v}) + (\mathbf{u} \perp \mathbf{w})$ [Additive Distribution]

$(\mathbf{v} \perp \mathbf{w})^2 + (\mathbf{v} \bullet \mathbf{w})^2 = (|\mathbf{v}||\mathbf{w}|)^2$ [Lagrange Identity]

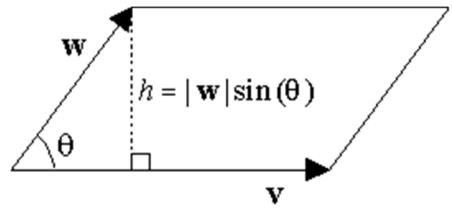
Also, for the 2D perp product, we have another useful formula:

$$\mathbf{v} \perp \mathbf{w} = |\mathbf{v}||\mathbf{w}| \sin(\theta)$$

which can be used to compute $\sin(\theta)$ from **v** and **w**. And if $|\mathbf{v}| = |\mathbf{w}| = \mathbf{1}$, then $\sin\theta = \mathbf{v} \perp \mathbf{w}$.

Moreover, geometrically the perp product gives the (signed) area of the 2D parallelogram spanned by **v** and **w**, as shown in the diagram:

$$\begin{aligned} \text{Area}(\square_{2D}) &= bh \\ &= |\mathbf{v}||\mathbf{w}||\sin(\theta)| \\ &= \mathbf{v} \perp \mathbf{w} \end{aligned}$$



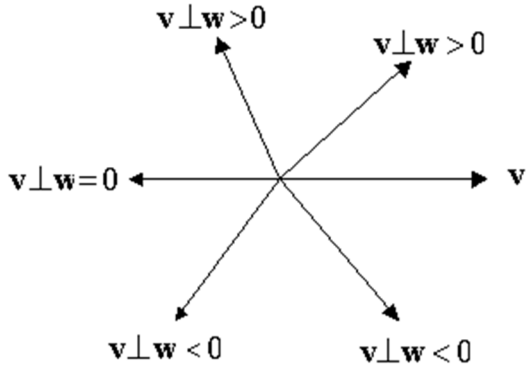
So, to compute the area of a 2D triangle with vertices $P_0P_1P_2$, define the edge vectors at P_0 as $\mathbf{v} = P_1 - P_0$ and $\mathbf{w} = P_2 - P_0$. Then, since a triangle is half of a parallelogram, we get the (signed) area of $\mathbf{\Delta}P_0P_1P_2$ as

$$\text{Area}(\mathbf{\Delta}P_0P_1P_2) = (\mathbf{v} \perp \mathbf{w})/2$$

which is a very efficient formula for area. This signed area is positive when the vertices $P_0P_1P_2$ are oriented counterclockwise and it is negative when they are oriented clockwise, so it can be used to test for the orientation of a triangle. This can also be used to test for which side of the directed line through P_0P_1 the point P_2 lies on. It is left side of P_0P_1 when the area is positive; it is on the line when the area = 0, and it is on the right side when the area is negative. If one only wants to test the sign of the area, dividing by 2 is not necessary.

Similarly, the 2D perp product can be used to determine which side (left or right) of one vector another vector is pointing, since for $|\theta| \leq 180°$, we know the following.

$\mathbf{v} \perp \mathbf{w} = 0 \Leftrightarrow \theta = 0 \text{ or } 180°$	They are collinear
$\mathbf{v} \perp \mathbf{w} > 0 \Leftrightarrow 0 < \theta < 180°$	$\mathbf{w}$ is left of $\mathbf{v}$
$\mathbf{v} \perp \mathbf{w} < 0 \Leftrightarrow 0 > \theta > -180°$	$\mathbf{w}$ is right of $\mathbf{v}$



The 3D Cross Product

The **3D cross product** (aka **3D outer product** or **vector product**) is defined for two 3D vectors, say $\mathbf{v} = (v_1, v_2, v_3)$ and $\mathbf{w} = (w_1, w_2, w_3)$, and it is another 3D vector.

3D Cross Product Definition	$\mathbf{v} \times \mathbf{w} = \left(\begin{vmatrix} v_2 & v_3 \\ w_2 & w_3 \end{vmatrix}, \begin{vmatrix} v_3 & v_1 \\ w_3 & w_1 \end{vmatrix}, \begin{vmatrix} v_1 & v_2 \\ w_1 & w_2 \end{vmatrix}\right)$

The cross product has the following properties.

$\mathbf{v} \times \mathbf{v} = \mathbf{0} = (0,0,0)$	[Nilpotent]
$(a\mathbf{v}) \times (b\mathbf{w}) = (ab)(\mathbf{v} \times \mathbf{w})$	[Scalar Association]
$\mathbf{v} \times \mathbf{w} = -(\mathbf{w} \times \mathbf{v})$	[Antisymmetric]
$\mathbf{u} \times (\mathbf{v} + \mathbf{w}) = (\mathbf{u} \times \mathbf{v}) + (\mathbf{u} \times \mathbf{w})$	[Additive Distribution]
$(\mathbf{v} \times \mathbf{w}) \bullet \mathbf{v} = (\mathbf{v} \times \mathbf{w}) \bullet \mathbf{w} = 0$	[Normality]
$\lvert\mathbf{v} \times \mathbf{w}\rvert^2 + (\mathbf{v} \bullet \mathbf{w})^2 = (\lvert\mathbf{v}\rvert\lvert\mathbf{w}\rvert)^2$	[Lagrange Identity]

However, the cross product is not associative with itself, and it is not distributive with the dot product. Instead, one has the following formulas. These are not often used in computer graphics but sometimes can streamline computations since dot products are easier to compute than cross products. Note that the formulas for left and right association are different.

$(\mathbf{u} \times \mathbf{v}) \times \mathbf{w} = (\mathbf{u} \bullet \mathbf{w})\mathbf{v} - (\mathbf{v} \bullet \mathbf{w})\mathbf{u}$	[Left Association]
$\mathbf{u} \times (\mathbf{v} \times \mathbf{w}) = (\mathbf{u} \bullet \mathbf{w})\mathbf{v} - (\mathbf{u} \bullet \mathbf{v})\mathbf{w}$	[Right Association]
$\mathbf{u} \bullet (\mathbf{v} \times \mathbf{w}) = (\mathbf{u} \times \mathbf{v}) \bullet \mathbf{w}$	[•× Association]
$(\mathbf{a} \times \mathbf{b}) \bullet (\mathbf{v} \times \mathbf{w}) = (\mathbf{a} \bullet \mathbf{v})(\mathbf{b} \bullet \mathbf{w}) - (\mathbf{a} \bullet \mathbf{w})(\mathbf{b} \bullet \mathbf{v})$	[Binet-Cauchy Identity]
$\mathbf{u} \times (\mathbf{v} \times \mathbf{w}) + \mathbf{v} \times (\mathbf{w} \times \mathbf{u}) + \mathbf{w} \times (\mathbf{u} \times \mathbf{v}) = \mathbf{0}$	[Jacobi Identity]

Using the Lagrange Identity, we can compute that:

$$\begin{aligned} \lvert\mathbf{v} \times \mathbf{w}\rvert^2 &= (\lvert\mathbf{v}\rvert\lvert\mathbf{w}\rvert)^2 - (\mathbf{v} \bullet \mathbf{w})^2 \\ &= \lvert\mathbf{v}\rvert^2\lvert\mathbf{w}\rvert^2(1 - \cos^2(\theta)) \\ &= \lvert\mathbf{v}\rvert^2\lvert\mathbf{w}\rvert^2 \sin^2(\theta) \end{aligned}$$

which demonstrates the cross product formulas:

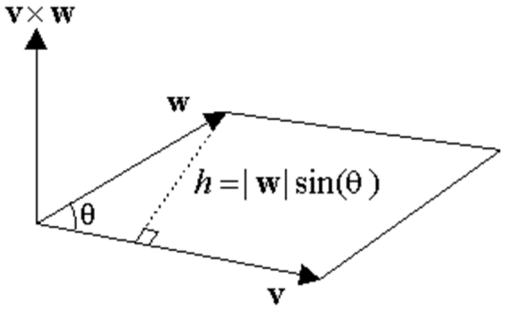
$$\lvert\mathbf{v} \times \mathbf{w}\rvert = \lvert\mathbf{v}\rvert\lvert\mathbf{w}\rvert\lvert\sin(\theta)\rvert$$

$$\mathbf{v} \times \mathbf{w} = \lvert\mathbf{v}\rvert\lvert\mathbf{w}\rvert \sin(\theta)\, \mathbf{u}, \quad \text{with } \lvert\mathbf{u}\rvert = 1 \text{ and } \mathbf{u} \bullet \mathbf{v} = \mathbf{u} \bullet \mathbf{w} = 0$$

where $\theta = \angle(\mathbf{v}, \mathbf{w})$. The vector $\mathbf{u}$ is perpendicular to both $\mathbf{v}$ and $\mathbf{w}$, and geometrically points outward from the $\mathbf{vw}$-plane using a right-hand rule.

Further, the magnitude $|\mathbf{v} \times \mathbf{w}|$ is the area of the parallelogram spanned by $\mathbf{v}$ and $\mathbf{w}$ as shown in the diagram.

$$\begin{aligned}\text{Area}(\square_{2D}) &= bh \\ &= |\mathbf{v}||\mathbf{w}||\sin(\theta)| \\ &= |\mathbf{v} \times \mathbf{w}|\end{aligned}$$



This fact makes the cross product useful for doing 3D area computations. For example, for a 3D triangle $\mathbf{\Delta} P_0 P_1 P_2$, with edge vectors $\mathbf{v} = P_1 - P_0$ and $\mathbf{w} = P_2 - P_0$, one can compute its area as

$$\begin{aligned}\text{Area}(\mathbf{\Delta} P_0 P_1 P_2) &= \tfrac{1}{2}|\mathbf{v} \times \mathbf{w}| \\ &= \tfrac{1}{2}|(P_1 - P_0) \times (P_2 - P_0)|\end{aligned}$$

Another important consequence of the cross product formula is that if $\mathbf{v}$ and $\mathbf{w}$ are perpendicular unit vectors, then $|\mathbf{v} \times \mathbf{w}|$ is also a unit vector. And the three vectors $\mathbf{v}$, $\mathbf{w}$, and $\mathbf{v} \times \mathbf{w}$ form an orthogonal coordinate frame of reference (or basis) for 3D space. This is used in 3D graphics to simplify perspective calculations from different observer viewpoints.

Further, in 2D space, there is a relationship between the embedded cross product and the 2D perp product. One can embed a 2D vector $\mathbf{v} = (v_1, v_2)$in 3D space by appending a third coordinate equal to 0, namely $(\mathbf{v}, 0) = (v_1, v_2, 0)$. Then, for two 2D vectors $\mathbf{v}$ and $\mathbf{w}$, the embedded 3D cross product is

$$(\mathbf{v}, 0) \times (\mathbf{w}, 0) = (0,0, v_1 w_2 - v_2 w_1) = (0,0, \mathbf{v} \perp \mathbf{w})$$

whose only non-zero component is equal to the perp product $\mathbf{v} \perp \mathbf{w}$.

The 3D Triple Product

Another useful geometric computation is the 3D (**scalar**) **triple product**, which is defined as

3D Triple Product Definition	$[\mathbf{u}\,\mathbf{v}\,\mathbf{w}] = \mathbf{u} \bullet (\mathbf{v} \times \mathbf{w}) = \begin{vmatrix} u_1 & u_2 & u_3 \\ v_1 & v_2 & v_3 \\ w_1 & w_2 & w_3 \end{vmatrix}$

This product has the properties:

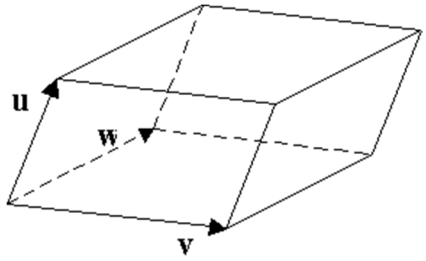
$[\mathbf{u}\,\mathbf{v}\,\mathbf{w}] = [\mathbf{v}\,\mathbf{w}\,\mathbf{u}] = [\mathbf{w}\,\mathbf{u}\,\mathbf{v}]$ [Even-Parity Invariant]

$[\mathbf{u}\,\mathbf{v}\,\mathbf{w}] = -[\mathbf{u}\,\mathbf{w}\,\mathbf{v}]$ [Anti-Symmetric]

$[\mathbf{u}\,\mathbf{v}\,\mathbf{w}]\mathbf{u} = (\mathbf{u} \times \mathbf{v}) \times (\mathbf{u} \times \mathbf{w})$

Geometrically, the triple product is equal to the volume of the parallelepiped (the 3D analog of a parallelogram) defined by the three vectors **u, v** and **w** starting from the same corner point as shown.

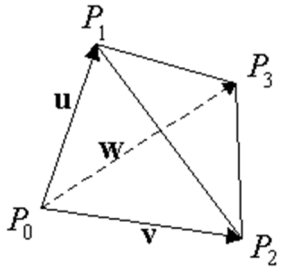
$\text{Volume}(\square_{3D}) = [\mathbf{u}\,\mathbf{v}\,\mathbf{w}]$



To understand this, recall that $|\mathbf{v} \times \mathbf{w}|$ is the area of the base (parallelogram), and the projection of **u** onto the normal vector $\mathbf{v} \times \mathbf{w}$ gives a perpendicular altitude with height $= |\mathbf{u}| \cos(\theta)$. The volume is then the product of the base area with this height, which yields the formula. Further, this is a signed volume, whose sign depends on the orientation of the "coordinate frame" basis vectors **u, v** and **w**.

Using this formula, we can also get the volume of a 3D tetrahedron $\mathbf{\Delta}^4 P_0 P_1 P_2 P_3$ with 4 vertices $P_i = (x_i, y_i, z_i)$ for i=0,3. The volume of this tetrahedron is 1/6 that of the parallelepiped spanned by the vectors $\mathbf{u} = P_1 - P_0$, $\mathbf{v} = P_2 - P_0$, and $\mathbf{w} = P_3 - P_0$. And then

$$\text{Vol}(\,\mathbf{\Delta}^4 P_0 P_1 P_2 P_3\,) = \tfrac{1}{6}[\mathbf{u}\,\mathbf{v}\,\mathbf{w}]$$



Area

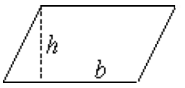
Computing the area of a planar polygon is a basic geometry calculation and can be found in many introductory texts. However, there are several different methods for computing planar areas depending on the information available.

Triangles

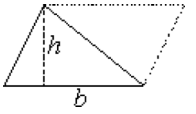
Ancient Triangles

Before Pythagoras, the area of the parallelogram (including the rectangle and the square) had been known to equal the product of its base times its height. Further, two copies of the same triangle paste together to form a parallelogram, and thus the area of a triangle is half of its base b times its height h. So, for these simple but commonly occurring cases, we have:

Parallelogram: $\mathrm{A}(\square) = bh$



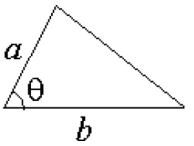
Triangle: $\mathrm{A}(\Delta) = \frac{1}{2}bh$



However, except in special situations, finding the height of a triangle at an arbitrary orientation usually requires also computing the perpendicular distance of the top vertex from the base.

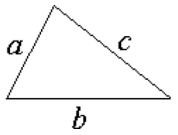
For example, if one knows the lengths of two sides, a and b, of a triangle and also the angle θ between them, then this is enough to determine the triangle and its area. Using trigonometry, the height of the triangle over the base b is given by $h = a\sin\theta$, and thus the area is:

$$\mathrm{A}(\Delta) = \tfrac{1}{2}ab\sin\theta$$



Another frequently used computation is derived from the fact that triangles with equal sides are congruent, and thus have the same area. This observation from Euclid (~300 BC) culminated in Heron's formula (~50 AD) for area as a function of the lengths of its three sides [Note: some historians attribute this result to Archimedes (~250 BC)]; namely:

$$\mathrm{A}(\Delta) = \sqrt{s(s-a)(s-b)(s-c)} \qquad \text{where } s = \tfrac{1}{2}(a+b+c)$$



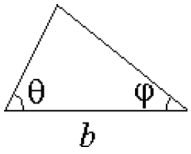
where *a*,*b*,*c* are the lengths of the sides, and *s* is the semiperimeter. There are interesting algebraic variations of this formula; such as:

$$\mathrm{A}(\Delta) = \tfrac{1}{4}\sqrt{4a^2b^2 - \left(a^2+b^2-c^2\right)^2}$$

which efficiently avoids calculating the 3 square roots to explicitly get the lengths *a*,*b*,*c* from the triangle's vertex coordinates.

The remaining classical triangle congruence is when two angles and one side are known. Knowing two angles gives all three, so we can assume the angles θ and φ are both adjacent to the known base *b*. Then the formula for area is:

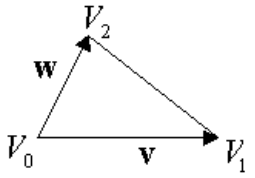
$$\mathrm{A}(\Delta) = \frac{b^2}{2(\cot\theta + \cot\varphi)}$$



Modern Triangles

More recently, starting in the 17-th century with Descartes and Fermat, linear algebra produced new simple formulas for area. In 3 dimensional space (3D), the area of a planar parallelogram or triangle can be expressed by the magnitude of the cross-product of two edge vectors, since $|\mathbf{v}\times\mathbf{w}| = |\mathbf{v}||\mathbf{w}||\sin(\theta)|$ where θ is the angle between the two vectors **v** and **w**. Thus for a 3D triangle with vertices $V_0V_1V_2$ putting $\mathbf{v} = V_1 - V_0$ and $\mathbf{w} = V_2 - V_0$, one gets:

$$\mathrm{A}(\Delta) = \tfrac{1}{2}|\mathbf{v}\times\mathbf{w}| = \tfrac{1}{2}\left|(V_1 - V_0)\times(V_2 - V_0)\right|$$



A 2D vector (*x*, *y*) can be viewed as embedded in 3D by adding a third z component set = 0. This lets one take the cross-product of 2D vectors, and use it to compute area. Given a triangle with vertices $V_i = (x_i, y_i) = (x_i, y_i, 0)$ for *i* = 0,1,2, we can compute that:

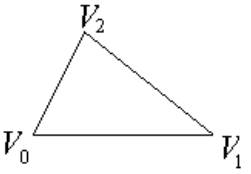
$$(V_1 - V_0)\times(V_2 - V_0) = \left(0, 0, \begin{vmatrix}(x_1 - x_0) & (x_2 - x_0)\\ (y_1 - y_0) & (y_2 - y_0)\end{vmatrix}\right)$$

And the absolute value of the third *z*-component is twice the absolute area of the triangle. However, it is useful to not take the absolute value here, and instead let the area be a signed quantity.

$$2\,\mathrm{A}(\Delta) = \begin{vmatrix} (x_1 - x_0) & (x_2 - x_0) \\ (y_1 - y_0) & (y_2 - y_0) \end{vmatrix} = \begin{vmatrix} x_0 & y_0 & 1 \\ x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \end{vmatrix}$$

$$= (x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0)$$

where $V_i = (x_i, y_i)$



This formula for area is a very efficient computation with no roots and no trigonometric functions involved - just 2 multiplications and 5 additions, and possibly 1 division by 2 (which can sometimes be avoided).

Note that the signed area will be positive if the vertices $V_0V_1V_2$ are oriented counterclockwise around the triangle, and will be negative if the triangle is oriented clockwise; and so, this area computation can be used to test for a triangle's orientation. This signed area can also be used to test whether the point V_2 is to the left (positive) or the right (negative) of the directed line segment V_0V_1 . So this value is a very useful primitive, and it's great to have such an efficient formula for it.

Quadrilaterals

The Greeks singled out certain quadrilaterals (also called quadrangles) for special treatment, including the square, the rectangle, the parallelogram, and the trapezium. Then, given an arbitrary quadrilateral, they showed how to construct a parallelogram [Euclid, Book I, Prop 45] or square [Euclid, Book II, Prop 14] with an equal area. And the area of the parallelogram was equal to its base times its height. But there was no general formula for the quadrilateral's area.

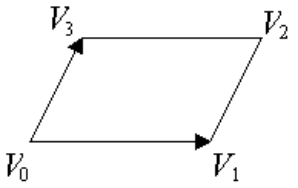
An extension of Heron's triangle area formula to quadrilaterals was discovered by the Hindu geometer Brahmagupta (620 AD) [Coxeter,1967, Section 3.2]. However, it only works for ***cyclic quadrilaterals***where all four vertices lie on the same circle. For a cyclic quadrilateral Θ, let the lengths of the four sides be *a*, *b*, *c*, *d*, and the semiperimeter be *s* = (*a*+*b*+*c*+*d*)/2. Then, the area of Θ is given by:

$$\mathrm{A}(\Theta) = \sqrt{(s-a)(s-b)(s-c)(s-d)}$$

which is an amazing symmetric formula. If one side is zero length, say *d* = 0, then we have a triangle (which is always cyclic) and this formula reduces to Heron's one.

In modern linear algebra, as already noted, the area of a planar parallelogram is the magnitude of the cross product of two adjacent edge vectors. So, for any 3D planar parallelogram $\square = V_0V_1V_2V_3$, we have:

$$\mathrm{A}(\square) = \left|(V_1 - V_0) \times (V_3 - V_0)\right|$$

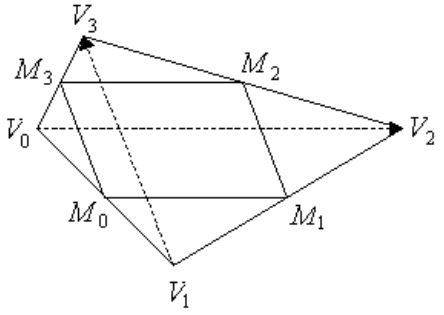


In 2D, with vertices $V_i = (x_i, y_i) = (x_i, y_i, 0)$ for $i = 0,3$, this becomes:

$$\begin{aligned}\mathrm{A}(\square) &= \begin{vmatrix}(x_1 - x_0) & (y_1 - y_0)\\ (x_3 - x_0) & (y_3 - y_0)\end{vmatrix}\\ &= (x_1 - x_0)(y_3 - y_0) - (x_3 - x_0)(y_1 - y_0)\end{aligned}$$

which is again a *signed* area, just as we had for triangles. It also indicates orientation.

Next, for an ***arbitrary quadrilateral***, one can compute its area using a parallelogram discovered by Pierre Varignon (first published in 1731). It is amazing that the Greeks missed Varignon's simple result which was discovered 2000 years after Euclid! Given any quadrilateral, one can take the midpoints of its 4 edges to get 4 vertices which form a new quadrilateral. It is then easy to show that this midpoint quadrilateral is always a parallelogram, called the "Varignon parallelogram", and that its area is exactly one-half the area of the original quadrilateral [Coxeter,1967, Section 3.1]. To see this, for any quadrilateral $\Theta = V_0V_1V_2V_3$, let the midpoint vertices be $M_0M_1M_2M_3$ as shown in the diagram:



From elementary geometry, we know that in triangle $V_0V_1V_2$ the midpoint line M_0M_1 is parallel to the base V_0V_2. In triangle $V_0V_2V_3$, the line M_2M_3 is parallel to that same base V_0V_2. Thus, M_0M_1 and M_2M_3 are parallel to each other. Similarly, M_0M_3 and M_1M_2 are parallel, which shows that $M_0M_1M_2M_3$ is a parallelogram. The area relation is also easy to demonstrate. And we can then compute the area as:

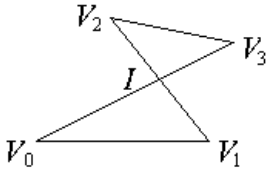
$$\begin{aligned}\mathrm{A}(\Theta) &= 2\,\mathrm{A}(M_0M_1M_2M_3)\\ &= 2\left|(M_1-M_0)\times(M_3-M_0)\right|\\ &= 2\left|\left(\frac{V_1+V_2}{2}-\frac{V_0+V_1}{2}\right)\times\left(\frac{V_3+V_0}{2}-\frac{V_0+V_1}{2}\right)\right|\\ &= \tfrac{1}{2}\left|(V_2-V_0)\times(V_3-V_1)\right|\end{aligned}$$

which is one-half the magnitude of the cross-product of the two diagonals of the quadrilateral. This result was noted by [Van Gelder,1995] who used a different proof. This formula holds for any 3D planar quadrilateral. When restricted to 2D with $V_i=(x_i,y_i)$, this becomes a formula for any quadrilateral's signed area:

$$\begin{aligned}2\,\mathrm{A}(\Theta) &= \begin{vmatrix}(x_2-x_0) & (y_2-y_0)\\ (x_3-x_1) & (y_3-y_1)\end{vmatrix}\\ &= (x_2-x_0)(y_3-y_1)-(x_3-x_1)(y_2-y_0)\end{aligned}$$

This formula for an arbitrary quadrilateral is just as efficient as the one for an arbitrary triangle, using only 2 multiplications and 5 additions. For simple quadrilaterals, the area is positive when the vertices are oriented counterclockwise, and negative when they are clockwise. However, it also works for nonsimple quadrilaterals and is equal to the difference in area of the two regions the quadrilateral bounds. For example, in the following diagram where I is the self-intersection point of a nonsimple quadrilateral $\Theta=V_0V_1V_2V_3$, we have the following.

$$\begin{aligned}\mathrm{A}(\Theta) &= \mathrm{A}(\Delta V_0V_1I)+\mathrm{A}(\Delta IV_2V_3)\\ &= \mathrm{A}(\Delta V_0V_1I)-\mathrm{A}(\Delta IV_3V_2)\end{aligned}$$

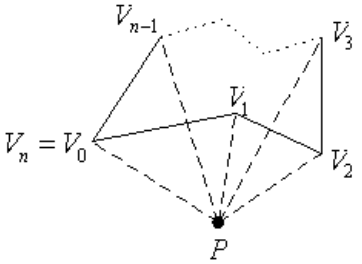


Polygons

2D Polygons

A 2D polygon can be decomposed into triangles. For computing area, there is a very easy decomposition method for ***simple polygons*** (i.e. ones without self intersections). Let a polygon Ω be defined by its vertices $V_i=(x_i,y_i)$ for $i=0,n$ with $V_n=V_0$. Also, let P be any point; and for each edge $\mathbf{e}_i=V_iV_{i+1}$ of Ω, form the triangle $\Delta_i=\Delta PV_iV_{i+1}$. Then, the area of Ω is equal to the sum of the signed areas of all the triangles Δ_i for $i=0,n-1$; and we have:

$$A(\Omega) = \sum_{i=0}^{n-1} A(\Delta_i)$$
$$\text{where } \Delta_i = \Delta P V_i V_{i+1}$$

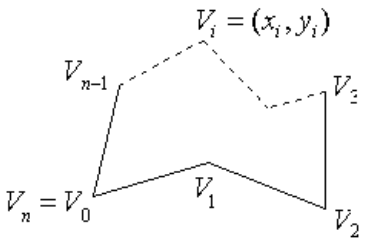


Notice that, for a counterclockwise oriented polygon, when the point P is on the "inside" left side of an edge V_iV_{i+1}, then the area of Δ_i is positive; whereas, when P is on the "outside" right side of an edge V_iV_{i+1}, then Δ_i has a negative area. If instead the polygon is oriented clockwise, then the signs are reversed, and inside triangles become negative.

For example, in the above diagram, the triangles Δ_2 and Δ_{n-1} have positive area, and contribute positively to the total area of polygon Ω. However, as one can see, only part of Δ_2 and Δ_{n-1} are actually inside Ω and there is a part of each triangle that is also exterior. On the other hand, the triangles Δ_0 and Δ_1 have negative area, and this cancels out the exterior excesses of positive area triangles. In the final analysis, the exterior areas all get canceled, and one is left with exactly the area of the polygon Ω.

One can make the formula more explicit by picking a specific point P and expanding the terms. By selecting $P = (0, 0)$, the area formula of each triangle reduces to $2A(\Delta_i) = (x_i y_{i+1} - x_{i+1} y_i)$. This yields:

$$\begin{aligned} 2A(\Omega) &= \sum_{i=0}^{n-1} \left(x_i y_{i+1} - x_{i+1} y_i\right) \\ &= \sum_{i=0}^{n-1} \left(x_i + x_{i+1}\right)\left(y_{i+1} - y_i\right) \\ &= \sum_{i=1}^{n} x_i \left(y_{i+1} - y_{i-1}\right) \end{aligned}$$
$$\text{where } V_i = \left(x_i, y_i\right), \text{ with } i \left(\text{mod} n\right)$$



A little algebra shows that the more efficient second and third summations are equal to the first [Sunday, 2002]. For a polygon with n vertices, the first summation uses $2n$ multiplications and $(2n–1)$ additions; the second uses n multiplications and $(3n–1)$ additions; and the third uses only n multiplications and $(2n–1)$ additions. So, the third is preferred for efficiency. And, to avoid any overhead from computing the index i(mod n), one can either extend the polygon array up to $V_{n+1} = V_1$, or simply put the final term outside of the summation loop. We give an efficient implementation below in the routine area2D_Polygon().

This computation gives a *signed* area for a polygon; and, similar to the signed area of a triangle, is *positive*when the vertices are oriented counterclockwise around the polygon, and *negative* when oriented clockwise. So, this computation can be used to test for a

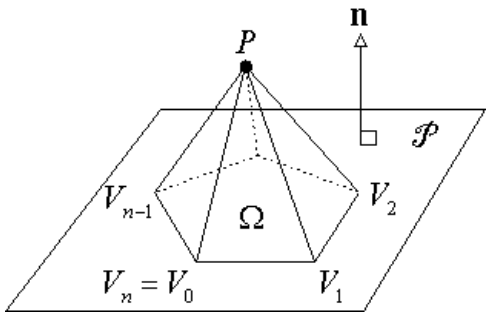
polygon's global orientation. However, there are other more efficient algorithms for determining polygon orientation. The easiest is to find the rightmost lowest vertex of the polygon, and then test the orientation of the entering and leaving edges at this vertex. This test can be made by checking if the end vertex of the leaving edge is to the left of the entering edge, which means that the orientation is counterclockwise, otherwise it is clockwise.

3D Planar Polygons

An important generalization is for planar polygons embedded in 3D space [Goldman, 1994]. We have already shown that the area of a 3D triangle $\Delta V_0V_1V_2$ is given by half the magnitude of the cross product of two edge vectors; namely, $|(V_1-V_0)\times(V_2-V_0)|/2$.

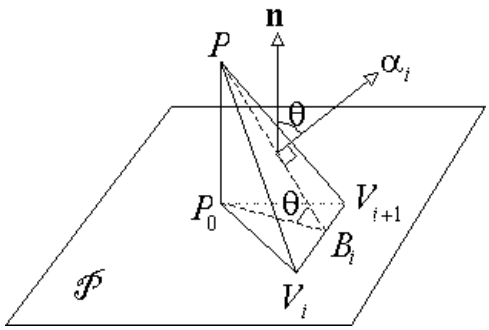
The Standard Formula

There is a classic standard formula for the area of a 3D polygon [Goldman, 1994] that extends the cross-product formula for a triangle. It can be derived from Stokes Theorem. However, we show here how to derive it from a 3D triangular decomposition that is geometrically more intuitive.



A general 3D planar polygon Ω has vertices $V_i = (x_i, y_i\, z_i)$ for $i = 0,n$ with $V_n = V_0$, where all the vertices lie on the same 3D plane $\mathcal{P}$ which has a ***unit normal*** vector **n**. Now, as in the 2D case, let P be any 3D point (not generally on the plane $\mathcal{P}$); and for each edge $\mathbf{e}_i = V_iV_{i+1}$ of Ω, form the 3D triangle $\Delta_i = \Delta PV_iV_{i+1}$. We would like to relate the sum of the areas of all these triangles to the area of the polygon Ω in the plane $\mathcal{P}$. But what we have is a pyramidal cone with P as an apex over the polygon Ω as a base. We are going to project the triangular sides of this cone onto the plane $\mathcal{P}$ of the base polygon, and compute signed areas of the projected triangles. Then the sum of the projected areas will equal the total area of the planar polygon.

To achieve this, start by associating to each triangle $\Delta_i = \Delta PV_iV_{i+1}$ an area vector $\boldsymbol{\alpha}_i = [(V_i - P)\times(V_{i+1} - P)]/2$, which is perpendicular to Δ_i, and whose magnitude we know is equal to that triangle's area. Next, drop a perpendicular from $P$to a point P_0 on $\mathcal{P}$, and consider the projected triangle $\mathbf{T}_i = \Delta P_0V_iV_{i+1}$. Then drop a



perpendicular P_0B_i from P_0 to $B_i$on the edge $\mathbf{e}_i = V_iV_{i+1}$. Since PP_0 is also perpendicular to $\mathbf{e}_i$, the three points PP_0B_i define a plane that is perpendicular to $\mathbf{e}_i$, and thus PB_i is a perpendicular from P to $\mathbf{e}_i$. Thus $|PB_i|$ is the height of Δ_i, and $|P_0B_i|$ is the height of $\mathbf{T}_i$. Further, the angle between these two altitudes = θ = the angle between $\mathbf{n}$ and $\boldsymbol{\alpha}_i$ since a 90° rotation (in the PP_0B_i plane) results in congruence. This gives:

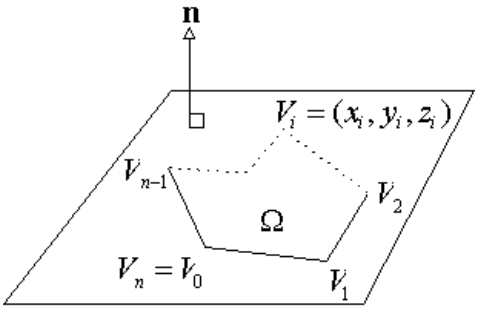
$$\mathrm{A}(\mathbf{T}_i) = \tfrac{1}{2}\left|V_iV_{i+1}\right|\left|P_0B_i\right| = \tfrac{1}{2}\left|V_iV_{i+1}\right|\left|PB_i\right|\cos\theta = \mathrm{A}(\Delta_i)\cos\theta = |\mathbf{n}||\boldsymbol{\alpha}_i|\cos\theta = \mathbf{n}\bullet\boldsymbol{\alpha}_i$$

This signed area computation is positive if the vertices of $\mathbf{T}_i$ are oriented counterclockwise when we look at the plane $\mathcal{P}$ from the side pointed to by $\mathbf{n}$. As in the 2D case, we can now add together the signed areas of all the triangles $\mathbf{T}_i$ to get the area of the polygon Ω. Writing this down, we have:

$$\mathrm{A}(\Omega) = \sum_{i=0}^{n-1}\mathrm{A}(\mathbf{T}_i) = \sum_{i=0}^{n-1}\mathbf{n}\bullet\boldsymbol{\alpha}_i = \frac{\mathbf{n}}{2}\bullet\sum_{i=0}^{n-1}\left(PV_i\times PV_{i+1}\right)$$

Finally, by selecting P = (0,0,0), we have $PV_i = V_i$ and this produces the concise formula:

$$2\,\mathrm{A}(\Omega) = \mathbf{n}\bullet\sum_{i=0}^{n-1}\left(V_i\times V_{i+1}\right)$$



which uses $6n$+3 multiplications and $4n$+2 additions

Similar to the 2D case, this is a *signed* area which is positive when the vertices are oriented counterclockwise around the polygon when viewed from the side of $\mathcal{P}$ pointed to by $\mathbf{n}$.

Quadrilateral Decomposition

[Van Gelder, 1995] has shown how to significantly speed up this computation by using a decomposition into quadrilaterals instead of triangles. As we have already shown, the area of a 3D planar quadrilateral $\Theta = V_0V_1V_2V_3$ can be computed in terms of the cross-product of its diagonals; namely as: $2\,\mathrm{A}(\Theta) = \mathbf{n}\bullet[(V_2 - V_0)\times(V_3 - V_1)]$, which reduces four expensive cross-product computations to just one!

Then, any polygon Ω (with $n > 4$ vertices) can be decomposed into quadrilaterals formed by V_0 and three other sequential vertices V_{2i-1}, V_{2i}, and V_{2i+1} for $i = 1,h$ where h = the greatest integer $\leq (n-1)/2$. When n is odd, the decomposition ends with a triangle. This gives:

$$2\,\mathrm{A}(\Omega) = \mathbf{n} \bullet \left(\sum_{i=1}^{h-1} \left(V_{2i} - V_0\right) \times \left(V_{2i+1} - V_{2i-1}\right) + \left(V_{2h} - V_0\right) \times \left(V_k - V_{2h-1}\right) \right)$$

where $k = 0$ for n odd, and $k = n-1$ for n even. This formula reduces the number of expensive cross-products by a factor of two (replacing them with vector subtractions). In total there are $3n+3$ multiplications and $5n+1$ additions making this formula roughly twice as fast as the classical one.

[Van Gelder, 1995] also states that this method can be applied to 2D polygons, but he does not write down the details. Working this out produces a formula that uses n multiplications and $3n-1$ additions, which is not as fast as the prior 2D formula we have given that use only n multiplications and $(2n-1)$ additions. We simply note this here, and do not pursue it further.

Projection to 2D

However, one can further significantly speed up the computation of 3D planar polygon area by projecting the polygon onto a 2D plane [Snyder & Barr, 1987]. Then, the area can be computed in 2D using our fastest formula, and the 3D area is recovered by using an area scaling factor. This method projects the 3D-embedded polygon onto an axis-aligned 2D subplane, by ignoring one of the three coordinates. Further, to get the correct area sign for the orientation of the projected polygons, the basis vectors of the 2D subplanes have to be ordered so that their orientation is compatible with the 3D basis vectors. In particular, if (x,y,z) are the 3D coordinates for the standard xyz basis, then the projections that ignore x, y, and z are onto the yz, zx, and xy subplanes respectively. That is, the projections are given by: $\mathrm{Proj}_x(x,y,z) = (y,z)$, $\mathrm{Proj}_y(x,y,z) = (z,x)$, and $\mathrm{Proj}_z(x,y,z) = (x,y)$. Then, these projections are "orientation-preserving", in that the sign of the projected polygon's area matches it's orientation in the projection subplane. And because of this, the scaling factor uses the sign of the component that is ignored.

Next, to select the projection that best avoids degeneracy and optimizes robustness, we look at the polygon's normal vector **n**, and choose the component with the greatest absolute value as the one to ignore. Let $\mathrm{Proj}_c()$ be the projection that ignores the selected coordinate $c = x$, y, or z. Then, the ratio of areas for the projected polygon $\mathrm{Proj}_c(\Omega)$ and original polygon Ω with normal $\mathbf{n} = (n_x, n_y, n_z)$ is given by:

$$\frac{\mathrm{A}\left(\mathrm{Proj}_c(\Omega)\right)}{\mathrm{A}(\Omega)} = \frac{n_c}{|\mathbf{n}|} \quad \text{where } c = x, y, \text{ or } z$$

An interesting consequence of this formula, is that the "area vector" **a** of Ω is now given by:

$$\mathbf{a}(\Omega) = \left(A(\mathrm{Proj}_x(\Omega)),\ A(\mathrm{Proj}_y(\Omega)),\ A(\mathrm{Proj}_z(\Omega))\right)$$

This area vector **a** is normal to the plane of the 3D polygon, and it's length is the area of the 3D polygon. So conversely, if the 3D polygon area $A(\Omega)$ is already known, and **n** is the unit normal vector for Ω's plane, then $A(\Omega)\mathbf{n}$ is the area vector, whose components are thus the areas of Ω's projections. In effect, by computing the area of one polygon projection, one can quickly get the areas of the other projections.

As a result, the 3D planar area can be recovered by a single extra multiplication in addition to the 2D area computation, and in total this algorithm uses n+5 multiplications, 2n+1 additions, 1 square root (when **n** is not a unit normal), plus a small overhead choosing the coordinate to ignore. This is a very significant improvement over the classical 3D formula, achieving a 6 times speed-up! We give an efficient implementation below in the routine area3D_Polygon().

Implementations

Here are some sample "C++" implementations of these formulas as algorithms. We just give the 2D case with integer coordinates, and use the simplest structures for a point, a triangle, and a polygon which may differ in your application. We represent a polygon as an array of points, but it is often more convenient to have it as a linked list of vertices (to allow insertion or deletion during drawing operations), and the polygon routines can be easily modified to scan through the linked list (see [O'Rourke, 1998] for an example of this approach).

```
// Copyright 2001, 2012, 2021 Dan Sunday
// This code may be freely used and modified for any purpose
// providing that this copyright notice is included with it.
// There is no warranty for this code, and the author of it cannot
// be held liable for any real or imagined damage from its use.
// Users of this code must verify correctness for their application.


// a Point (or vector) is defined by its coordinates
typedef struct {int x, y, z;} Point;  // set z=0 for a 2D Point

// a Triangle is given by three points: Point V0, V1, V2

// a Polygon is given by:
//       int n = number of vertex points
//       Point* V[] = an array of n+1 vertex points with V[n]=V[0]


// Note: for efficiency low-level functions are declared inline.
```

```
// isLeft(): test if a point is Left|On|Right of an infinite 2D line.
//    Input:  three points P0, P1, and P2
//    Return: >0 for P2 left of the line through P0 to P1
//          =0 for P2 on the line
//          <0 for P2 right of the line
inline int
isLeft( Point P0, Point P1, Point P2 )
{
    return ( (P1.x - P0.x) * (P2.y - P0.y)
           - (P2.x - P0.x) * (P1.y - P0.y) );
}
//===================================================================


// orientation2D_Triangle(): test the orientation of a 2D triangle
//  Input:  three vertex points V0, V1, V2
//  Return: >0 for counterclockwise
//          =0 for none (degenerate)
//          <0 for clockwise
inline int
orientation2D_Triangle( Point V0, Point V1, Point V2 )
{
    return isLeft(V0, V1, V2);
}
//===================================================================


// area2D_Triangle(): compute the area of a 2D triangle
//  Input:  three vertex points V0, V1, V2
//  Return: the (float) area of triangle T
inline float
area2D_Triangle( Point V0, Point V1, Point V2 )
{
    return (float)isLeft(V0, V1, V2) / 2.0;
}
//===================================================================


// orientation2D_Polygon(): get the orientation of a simple polygon
//  Input:  int n = the number of vertices in the polygon
//          Point* V = an array of n+1 vertex points with V[n]=V[0]
//  Return: >0 for counterclockwise
//          =0 for none (degenerate)
//          <0 for clockwise
//  Note: this algorithm is faster than computing the signed area.
int
orientation2D_Polygon( int n, Point* V )
{
    // first find rightmost lowest vertex of the polygon
    int rmin = 0;
    int xmin = V[0].x;
    int ymin = V[0].y;

    for (int i=1; i<n; i++) {
```

```
        if (V[i].y > ymin)
            continue;
        if (V[i].y == ymin) {   // just as low
            if (V[i].x < xmin)  // and to left
                continue;
        }
        rmin = i;       // a new rightmost lowest vertex
        xmin = V[i].x;
        ymin = V[i].y;
    }

    // test orientation at the rmin vertex
    // ccw <=> the edge leaving V[rmin] is left of the entering edge
    if (rmin == 0)
        return isLeft( V[n-1], V[0], V[1] );
    else
        return isLeft( V[rmin-1], V[rmin], V[rmin+1] );
}
 //===================================================================


// area2D_Polygon(): compute the area of a 2D polygon
//  Input:  int n = the number of vertices in the polygon
//          Point* V = an array of n+1 points with V[n]=V[0]
//  Return: the (float) area of the polygon
float
area2D_Polygon( int n, Point* V )
{
    float area = 0;
    int  i, j, k;   // indices

    if (n < 3) return 0;  // a degenerate polygon

    for (i=1, j=2, k=0; i<n; i++, j++, k++) {
        area += V[i].x * (V[j].y - V[k].y);
    }
    area += V[n].x * (V[1].y - V[n-1].y);  // wrap-around term
    return area / 2.0;
}
//===================================================================


// area3D_Polygon(): compute the area of a 3D planar polygon
//  Input:  int n = the number of vertices in the polygon
//          Point* V = an array of n+1 planar points with V[n]=V[0]
//          Point N = a normal vector of the polygon's plane
//  Return: the (float) area of the polygon
float
area3D_Polygon( int n, Point* V, Point N )
{
    float area = 0;
    float an, ax, ay, az; // abs value of normal and its coords
    int  coord;           // coord to ignore: 1=x, 2=y, 3=z
    int  i, j, k;         // loop indices
```

```
if (n < 3) return 0;  // a degenerate polygon

// select largest abs coordinate to ignore for projection
ax = (N.x>0 ? N.x : -N.x);    // abs x-coord
ay = (N.y>0 ? N.y : -N.y);    // abs y-coord
az = (N.z>0 ? N.z : -N.z);    // abs z-coord

coord = 3;                    // ignore z-coord
if (ax > ay) {
    if (ax > az) coord = 1;   // ignore x-coord
}
else if (ay > az) coord = 2;  // ignore y-coord

// compute area of the 2D projection
switch (coord) {
  case 1:
    for (i=1, j=2, k=0; i<n; i++, j++, k++)
        area += (V[i].y * (V[j].z - V[k].z));
    break;
  case 2:
    for (i=1, j=2, k=0; i<n; i++, j++, k++)
        area += (V[i].z * (V[j].x - V[k].x));
    break;
  case 3:
    for (i=1, j=2, k=0; i<n; i++, j++, k++)
        area += (V[i].x * (V[j].y - V[k].y));
    break;
}
switch (coord) {    // wrap-around term
  case 1:
    area += (V[n].y * (V[1].z - V[n-1].z));
    break;
  case 2:
    area += (V[n].z * (V[1].x - V[n-1].x));
    break;
  case 3:
    area += (V[n].x * (V[1].y - V[n-1].y));
    break;
}

// scale to get area before projection
an = sqrt( ax*ax + ay*ay + az*az); // length of normal vector
switch (coord) {
  case 1:
    area *= (an / (2 * N.x));
    break;
  case 2:
    area *= (an / (2 * N.y));
    break;
  case 3:
    area *= (an / (2 * N.z));
}
return area;
```

```
}
//===================================================================
```

References

Donald Coxeter & Samuel Greitzer, Geometry Revisited (1967)

Ronald Goldman, "Area of Planar Polygons and Volume of Polyhedra" in Graphics Gems II (1994)

Joseph O'Rourke, Computational Geometry in C (2nd Edition), Section 1.3 "Area of a Polygon" (1998)

J.P. Snyder and A.H. Barr, "Ray Tracing Complex Models Containing Surface Tessellations", ACM Comp Graphics 21, (1987)

Dan Sunday, "Fast Polygon Area and Newell Normal Computation", journal of graphics tools (***jgt***) Vol 7(2), (2002)

Allen Van Gelder, "Efficient Computation of Polygon Area and Polyhedron Volume" in Graphics Gems V (1995)

Lines

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:

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

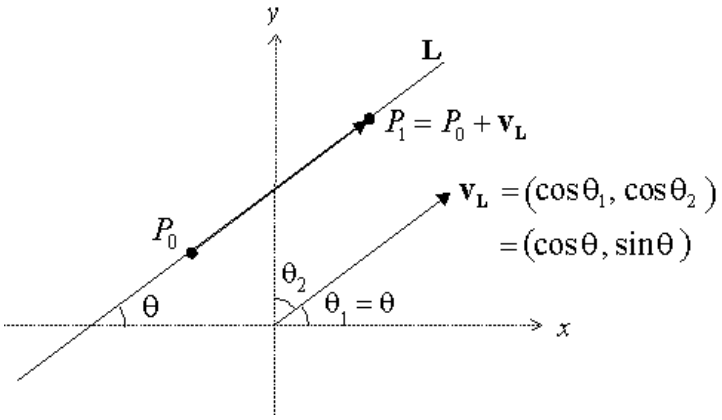
Lines

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, 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:

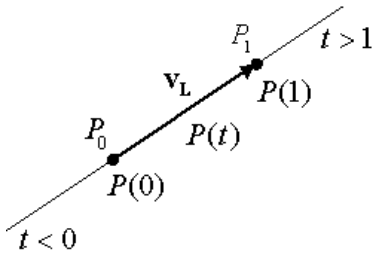
$$\mathbf{n_L} \bullet (P - P_0) = ax + by - \mathbf{n_L} \bullet P_0 = 0$$

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. This, of course, is useful in its own right, but it is not our interest here.

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.

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



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:

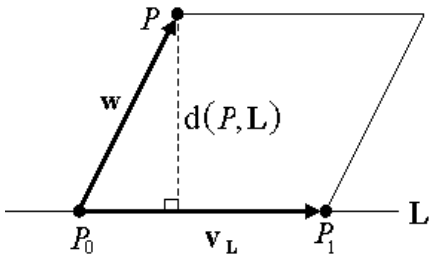
$$P(t) = (x_0 + t\cos\theta, y_0 + t\sin\theta)$$

Distance from a Point to a 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:

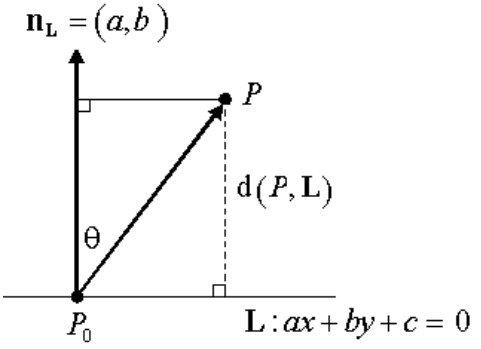
$$\mathrm{d}(P,\mathbf{L}) = \frac{(y_0 - y_1)x + (x_1 - x_0)y + (x_0 y_1 - x_1 y_0)}{\sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2}}$$

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 $\mathrm{f}(x, y) = a\,x + b\,y + c = 0$. 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: $\mathbf{n_L} \bullet P_0P = |\mathbf{n_L}||P_0P|\cos\theta = |\mathbf{n_L}|\,\mathrm{d}(P,\mathbf{L})$

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

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

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

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:

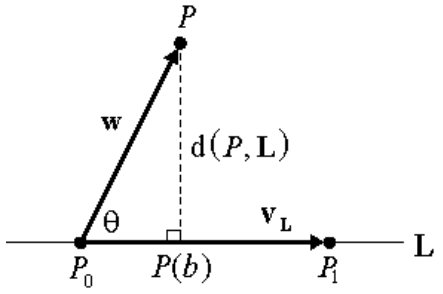
$$\mathrm{d}(P,\mathbf{L}) = \mathrm{f}(P) = ax + by + c \quad \text{when} \quad a^2 + b^2 = 1$$

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 (any dimension)

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

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

and thus

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

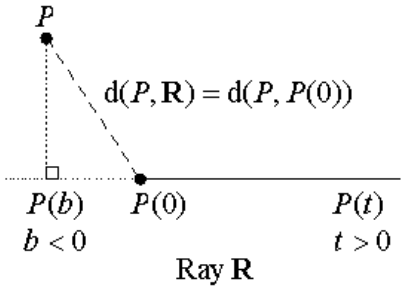
where $\mathbf{u}_\mathbf{L}$ is our friend the unit direction vector of **L**.

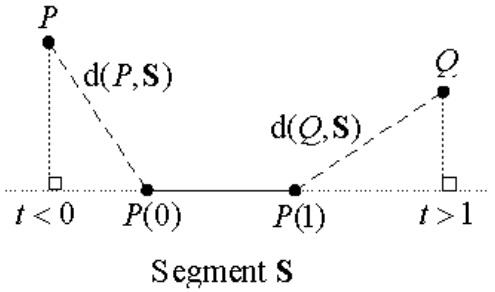
This computation has the advantage of working for any dimension *n* and of also computing the base point $P(b)$ which is sometimes useful. In 3D, it is just as efficient as the cross product formula. But in 2D, when $P(b)$ is not needed, the implicit method is better, especially if one is computing the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

Distance from a Point to a Ray or Segment (any dimension)

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 on a line that are between 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.

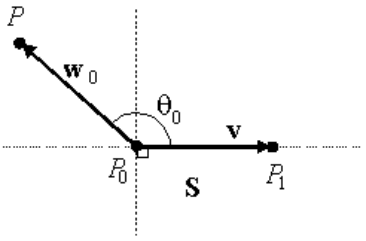




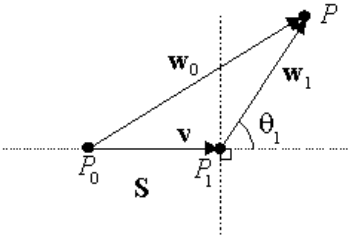
For a ray there is only one choice, but for a segment one must determine which end of the segment is closest to P. One could just compute both distances and use the shortest, but this is not very efficient. Also, one must first determine that P's perpendicular base is actually outside the segment's range. 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 following diagrams:

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



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



Since $\mathbf{w}_0 = \mathbf{v} + \mathbf{w}_1$, the two tests can be done just using the two dot products $\mathbf{w}_0 \cdot \mathbf{v}$ and $\mathbf{v} \cdot \mathbf{v}$ 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, 2012, 2021 Dan Sunday
// This code may be freely used and modified for any purpose
// providing that this copyright notice is included with it.
// There is no warranty for this code, and the author of it cannot
// be held liable for any real or imagined damage 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)

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

Point in Polygon

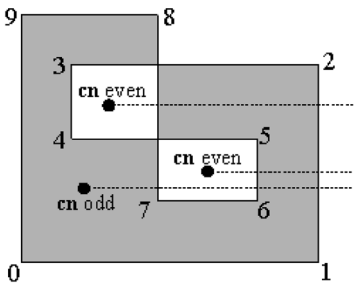
Determining the inclusion of a point P in a 2D planar polygon is a geometric problem that results in interesting algorithms. Two commonly used methods are:

1. The ***Crossing Number*** **(cn)** method
   - which counts the number of times a ray starting from the point P crosses the polygon boundary edges. The point is outside when this "crossing number" is even; otherwise, when it is odd, the point is inside. This method is sometimes referred to as the "even-odd" test.

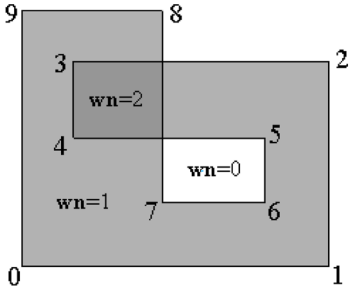
2. The ***Winding Number*** **(wn)** method
   - which counts the number of times the polygon winds around the point P. The point is outside only when this "winding number" wn = 0; otherwise, the point is inside.

If a polygon is simple (i.e., it has no self intersections), then both methods give the same result for all points. But for non-simple polygons, the two methods can give different answers for some points. For example, when a polygon overlaps with itself, then points in the region of overlap are found to be outside using the crossing number, but are inside using the winding number, as shown in the diagrams:

Crossing Number Method



Winding Number Method



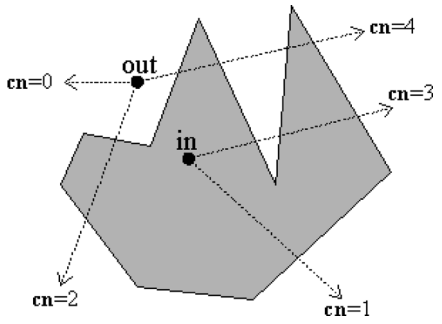
Vertices are numbered: 0 1 2 3 4 5 6 7 8 9

In this example, points inside the overlap region have **wn** = 2, implying that they are inside the polygon twice. Clearly, the winding number gives a better intuitive answer than the crossing number does.

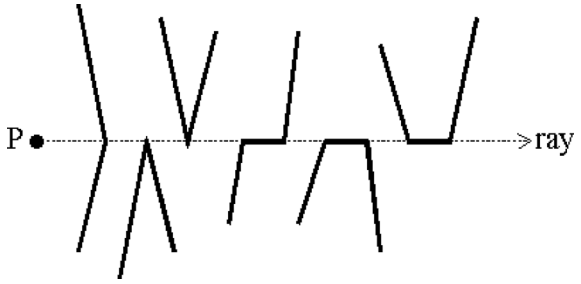
Despite this, the crossing number method is more commonly used since **cn** is erroneously thought to be significantly more efficient to compute than **wn** [O'Rourke, 1998]. But this is not the case, and **wn** can be computed with the same efficiency as **cn** by counting signed crossings. In fact, our implementation for wn_PnPoly() is faster than the code for **cn** given by [Franklin, 2000], [Haines, 1994] or [O'Rourke, 1998], although the **cn** "PointInPolygon()" routine of [Moller & Haines, 1999] is comparable to our **wn** algorithm. But the bottom line is that for both geometric correctness and efficiency reasons, the **wn** algorithm should always be preferred for determining the inclusion of a point in a polygon.

The Crossing Number

This method counts the number of times a ray starting from a point P crosses a polygon boundary edge separating it's inside and outside. If this number is even, then the point is outside; otherwise, when the crossing number is odd, the point is inside. This is easy to understand intuitively. Each time the ray crosses a polygon edge, its in-out parity changes (since a boundary always separates inside from outside, right?). Eventually, any ray must end up beyond and outside the bounded polygon. So, if the point is inside, the sequence of crossings ">" must be: in > out > ... > in > out, and there are an odd number of them. Similarly, if the point is outside, there are an even number of crossings in the sequence: out > in > ... > in > out.



In implementing an algorithm for the **cn** method, one must insure that only crossings that change the in-out parity are counted. In particular, special cases where the ray passes through a vertex must be handled properly. These include the following types of ray crossings:



Further, one must decide whether a point on the polygon's boundary is inside or outside. A standard convention is to say that a point on a left or bottom edge is inside, and a point on a right or top edge is outside. This way, if two distinct polygons share a common boundary segment, then a point on that segment will be in one polygon or the other, but not both at the same time. This avoids a number of problems that might occur, especially in computer graphics displays.

A straightforward "crossing number" algorithm selects a horizontal ray extending to the right of P and parallel to the positive x-axis. Using this specific ray, it is easy to compute the intersection of a polygon edge with it. It is even easier to determine when no such intersection is possible. To count the total crossings, **cn**, the algorithm simply loops through all edges of the polygon, tests each for a crossing, and increments **cn**when one occurs. Additionally, the crossing tests must handle the special cases and points on an edge. This is accomplished by the

Edge Crossing Rules

1. an upward edge includes its starting endpoint, and excludes its final endpoint;
2. a downward edge excludes its starting endpoint, and includes its final endpoint;
3. horizontal edges are excluded
4. the edge-ray intersection point must be strictly right of the point P.

One can apply these rules to the preceding special cases, and see that they correctly determine valid crossings. Note that Rule #4 results in points on a right-side boundary edge being outside, and ones on a left-side edge being inside.

Pseudo-Code: Crossing # Inclusion

Code for this algorithm is well-known, and the edge crossing rules are easily expressed. For a polygon represented as an array $V[n+1]$ of vertex points with $V[n]=V[0]$, popular implementation logic ([Franklin, 2000], [O'Rourke, 1998]) is as follows:

```
typedef struct {int x, y;} Point;

cn_PnPoly( Point P, Point V[], int n )
{
    int    cn = 0;    // the  crossing number counter

    // loop through all edges of the polygon
    for (each edge E[i]:V[i]V[i+1] of the polygon) {
        if (E[i] crosses upward ala Rule #1
         || E[i] crosses downward ala  Rule #2) {
            if (P.x <  x_intersect of E[i] with y=P.y)   // Rule #4
                 ++cn;   // a valid crossing to the right of P.x
        }
    }
    return (cn&1);    // 0 if even (out), and 1 if  odd (in)

}
```

Note that the tests for upward and downward crossing satisfying Rules #1 and #2 also exclude horizontal edges (Rule #3). All-in-all, a lot of work is done by just a few tests which makes this an elegant algorithm.

However, the validity of the crossing number method is based on the "Jordan Curve Theorem" which says that a simple closed curve divides the 2D plane into exactly 2 connected components: a bounded "inside" one and an unbounded "outside" one. The catch is that the curve must be simple (without self intersections), otherwise there can be more than 2 components and then there is no guarantee that crossing a boundary changes the in-out parity. For a closed (and thus bounded) curve, there is exactly one unbounded "outside" component; but bounded components can be either inside or outside. And two of the bounded components that have a shared boundary may both be inside, and crossing over their shared boundary would not change the in-out parity.

The Winding Number

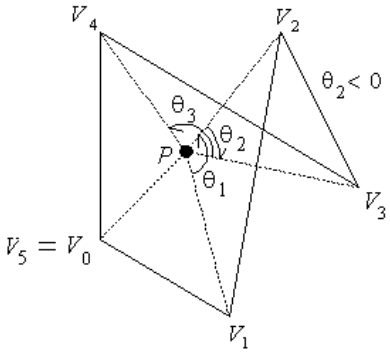
On the other hand, the winding number accurately determines if a point is inside a nonsimple closed polygon. It does this by computing how many times the polygon winds around the point. A point is outside only when the polygon doesn't wind around the point at all which is when the winding number **wn** = 0. More generally, one can define the winding number $\mathbf{wn}(P,\mathbf{C})$ of any closed continuous curve **C** around a point P in the 2D plane. Let the continuous 2D curve **C** be defined by the points $C(u)=C(x(u),y(u))$, for $0\le u\le 1$ with $C(0)=C(1)$. And let P be a point not on **C**. Then, define the vector $\mathbf{c}(P,u) = C(u) - P$ from P to $C(u)$, and the unit vector $\mathbf{w}(P,u) = \mathbf{c}(P,u) / |\mathbf{c}(P,u)|$ which gives a continuous function $W(P):\mathbf{C}\to\mathbf{S}^1$ mapping the point $C(u)$ on **C** to the point $\mathbf{w}(P,u)$ on the unit circle $\mathbf{S}^1=\{(x,y)|\,x^2+y^2=1\}$. This map can be represented in polar coordinates as $W(P)(u)=(\cos\theta(u),\sin\theta(u))$ where $\theta(u)$ is a positive counterclockwise angle in radians. The winding number $\mathbf{wn}(P,\mathbf{C})$ is then equal to the integer number of times that $W(P)$ wraps **C**around $\mathbf{S}^1$. This corresponds to a homotopy class of $\mathbf{S}^1$, and can be computed by the integral:

$$\mathbf{wn}(P,\mathbf{C})=\frac{1}{2\pi}\oint_{W(P)} d\theta=\frac{1}{2\pi}\int_{u=0}^{1}\theta(u)du$$

When the curve **C** is a polygon with vertices $V_0,V_1,...,V_n = V_0$, this integral reduces to the sum of the (signed) angles that each edge V_iV_{i+1} subtends with the point P. So, if $\theta_i=\measuredangle(PV_i,PV_{i+1})$, we have:

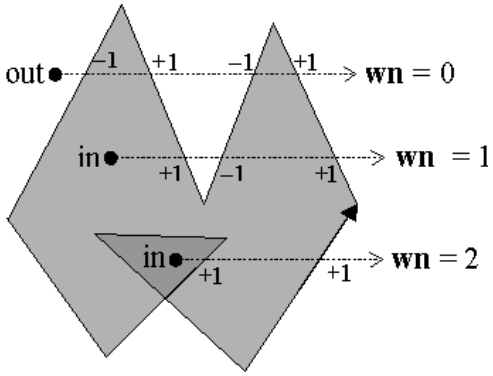
$$\mathbf{wn}(P,\mathbf{C})=\frac{1}{2\pi}\sum_{i=0}^{n-1}\theta_i$$

$$=\frac{1}{2\pi}\sum_{i=0}^{n-1}\arccos\left(\frac{(V_i-P)\bullet(V_{i+1}-P)}{|(V_i-P)||(V_{i+1}-P)|}\right)$$

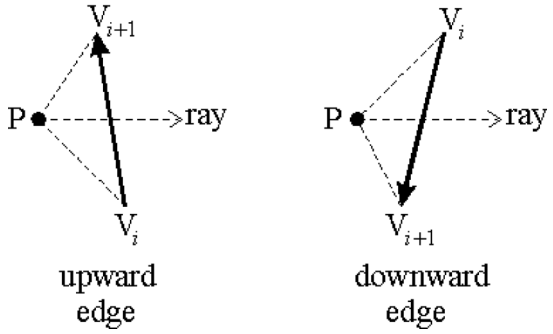


This formula is clearly not very efficient since it uses a computationally expensive arccos() trig function. But, a simple observation lets us replace this formula by a more efficient one. Pick any point Q on $\mathbf{S}^1$. Then, as the curve $W(P)$ wraps around $\mathbf{S}^1$, it passes Q a certain number of times. If we count (+1) when it passes Qcounterclockwise, and (–1) when it passes clockwise, then the accumulated sum is exactly the total number of times that $W(P)$ wraps around $\mathbf{S}^1$, and is equal to the winding number $\mathbf{wn}(P,\mathbf{C})$. Further, if we take an infinite ray **R** starting at P and extending in the direction of the vector Q, then intersections where **R** crosses the curve **C** correspond to the points where $W(P)$ passes Q. To do the math, we have to distinguish between positive and negative crossings where **C** crosses **R** from right-to-left or left-to-right. This can be determined by the sign of the dot product between a normal vector to **C** and the

direction vector **q** = Q, and when the curve **C** is a polygon, one just needs to make this determination once for each edge. For a horizontal ray **R**from P, testing whether an edge's endpoints are above and below the ray suffices. If the edge crosses the positive ray from below to above, the crossing is positive (+1); but if it crosses from above to below, the crossing is negative (–1). One then simply adds all crossing values to get $\mathbf{wn}(P,\mathbf{C})$. For example:



Additionally, one can avoid computing the actual edge-ray intersection point by using the isLeft() attribute; however, it needs to be applied differently for ascending and descending edges. If an upward edge crosses the ray to the right of P, then P is on the left side of the edge since the triangle $V_iV_{i+1}P$ is oriented counterclockwise. On the other hand, a downward edge crossing the positive ray would have P on the right side since the triangle $V_iV_{i+1}P$ would then be oriented clockwise.



Pseudo-Code: Winding Number Inclusion

This results in the following **wn** algorithm which is an adaptation of the **cn** algorithm and uses the same edge crossing rules as before to handle special cases.

```
typedef struct {int x, y;} Point;

wn_PnPoly( Point P, Point V[], int n )
{
    int    wn = 0;    // the  winding number counter
```

```
    // loop through all edges of the polygon
    for (each edge E[i]:V[i]V[i+1] of the polygon) {
        if (E[i] crosses upward ala Rule #1)  {
            if (P is  strictly left of E[i])    // Rule #4
                 ++wn;   // a valid up intersect right of P.x
        }
        else
        if (E[i] crosses downward ala Rule  #2) {
            if (P is  strictly right of E[i])   // Rule #4
                 --wn;   // a valid down intersect right of P.x
        }
    }
    return wn;    // =0 <=> P is outside the polygon

}
```

Clearly, this winding number algorithm has the same efficiency as the analogous crossing number algorithm. Thus, since it is more accurate in general, the winding number algorithm should always be the preferred method to determine inclusion of a point in an arbitrary polygon.

The **wn** algorithm's efficiency can be improved further by rearranging the crossing comparison tests. This is shown in the detailed implementation of wn_PnPoly() given below. In that code, all edges that are totally above or totally below P get rejected after only two (2) inequality tests. However, currently popular implementations of the **cn** algorithm ([Franklin, 2000], [Haines, 1994], [O'Rourke, 1998]) use *at least* three (3) inequality tests for each rejected edge. Since most of the edges in a large polygon get rejected in practical applications, there is about a 33% (or more) reduction in the number of comparisons done. In runtime tests using very large (1,000,000 edge) random polygons (with edge length < 1/10 the polygon diameter) and 1000 random test points (inside the polygon's bounding box), we measured a 20% increase in efficiency overall.

Enhancements

There are some enhancements to point in polygon algorithms [Haines, 1994] that software developers should be aware of. We mention a few that pertain to ray crossing algorithms. However, there are other techniques that give better performance in special cases such as testing inclusion in small convex polygons like triangles. These are discussed in [Haines, 1994].

Bounding Box or Ball

It is efficient to first test that a point P is inside the bounding box or ball of a large polygon before testing all edges for ray crossings. If a point is outside the bounding box or ball, it is also outside the polygon, and no further testing is needed. But, one must precompute the bounding box (the max and min for vertex x and $y$coordinates) or the bounding ball (center and minimum radius) and store it for future use. This is worth doing if more than a few points are going to be tested for inclusion, which is generally the case. Further information about computing bounding containers can be found in another chapter.

3D Planar Polygons

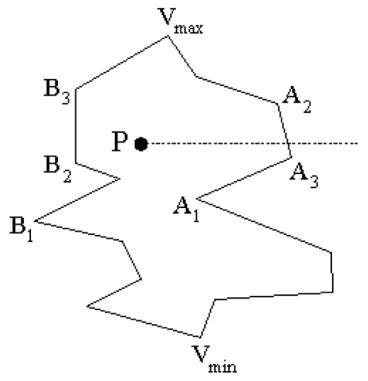
In 3D applications, one sometimes wants to test a point and polygon that are in the same plane. For example, one may have the intersection point of a ray with the plane of a polyhedron's face, and want to test if it is inside the face. Or one may want to know if the base of a 3D perpendicular dropped from a point is inside a planar polygon.

3D inclusion is easily determined by projecting the point and polygon into 2D. To do this, one simply ignores one of the 3D coordinates and uses the other two. To optimally select the coordinate to ignore, compute a normal vector to the plane, and select the coordinate with the largest absolute value [Snyder & Barr, 1987]. This gives the projection of the polygon with maximum area, and results in robust computations.

Convex or Monotone Polygons

When a polygon is known to be convex, many computations can be speeded up. For example, the bounding box can be computed in O(log *n*) time [O'Rourke, 1998] instead of O(*n*) as for nonconvex polygons. Also, point inclusion can be tested in O(log *n*) time. More generally, the following algorithm works for convex or monotone polygons.

A convex or *y*-monotone polygon can be split into two polylines, one with edges increasing in *y*, and one with edges decreasing in *y*. The split occurs at two vertices with max *y* and min *y* coordinates. Note that these vertices are at the top and bottom of the polygon's bounding box, and if they are both above or both below *P*'s y-coordinate, then the test point is outside the polygon. Otherwise, each of these two polylines intersects a ray paralell to the x-axis once, and each potential crossing edge can be found by doing a binary search on the vertices of each polyline. This results in a practical O(log *n*) (preprocessing and runtime) algorithm for convex and montone polygon inclusion testing. For example, in the following diagram, only three binary search vertices have to be tested on each polyline (*A*1, *A*2, *A*3 ascending; and *B*1, *B*2, *B*3 descending):



This method also works for polygons that are monotone in an arbitrary direction. One then uses a ray from *P*that is perpendicular to the direction of monotonicity, and the algorithm is easily

adapted. A little more computation is needed to determine which side of the ray a vertex is on, but for a large enough polygon, the O(log *n*) performance more than makes up for the overhead.

Implementations

Here is a "C++" implementation of the winding number algorithm for the inclusion of a point in polygon. We just give the 2D case, and use the simplest structures for a point and a polygon which may differ in your application.

```
// Copyright 2001, 2012, 2021 Dan Sunday
// This code may be freely used and modified for any purpose
// providing that this copyright notice is included with it.
// There is no warranty for this code, and the author of it cannot
// be held liable for any real or imagined damage from its use.
// Users of this code must verify correctness for their application.


// a Point is defined by its coordinates {int x, y;}
//===================================================================


// isLeft(): tests if a point is Left|On|Right of an infinite line.
//    Input:  three points P0, P1, and P2
//    Return: >0 for P2 left of the line through P0 and P1
//            =0 for P2  on the line
//            <0 for P2  right of the line
inline int
isLeft( Point P0, Point P1, Point P2 )
{
    return ( (P1.x - P0.x) * (P2.y - P0.y)
            - (P2.x -  P0.x) * (P1.y - P0.y) );
}
//===================================================================


// cn_PnPoly(): crossing number test for a point in a polygon
//      Input:   P = a point,
//               V[] = vertex points of a polygon V[n+1] with V[n]=V[0]
//      Return:  0 = outside, 1 = inside
// This code is patterned after [Franklin, 2000]
int
cn_PnPoly( Point P, Point* V, int n )
{
    int    cn = 0;    // the  crossing number counter

    // loop through all edges of the polygon
    for (int i=0; i<n; i++) {    // edge from V[i]  to V[i+1]
       if (((V[i].y <= P.y) && (V[i+1].y > P.y))     // an upward crossing
        || ((V[i].y > P.y) && (V[i+1].y <=  P.y))) { // a downward crossing
            // compute  the actual edge-ray intersect x-coordinate
            float vt = (float)(P.y  - V[i].y) / (V[i+1].y - V[i].y);
            float x_intersect = V[i].x + vt * (V[i+1].x - V[i].x);
            if (P.x < x_intersect)
                 ++cn;   // a valid crossing of y=P.y right of P.x
        }
```

```
    }
    return (cn&1);    // 0 if even (out), and 1 if  odd (in)

}
//===================================================================


// wn_PnPoly(): winding number test for a point in a polygon
//      Input:   P = a point,
//               V[] = vertex points of a polygon V[n+1] with V[n]=V[0]
//      Return:  wn = the winding number (=0 only when P is outside)
int
wn_PnPoly( Point P, Point* V, int n )
{
    int    wn = 0;    // the  winding number counter

    // loop through all edges of the polygon
    for (int i=0; i<n; i++) {   // edge from V[i] to  V[i+1]
        if (V[i].y <= P.y) {          // start y <= P.y
            if (V[i+1].y  > P.y)      // an upward crossing
                 if (isLeft( V[i], V[i+1], P) > 0)  // P left of  edge
                     ++wn;            // have  a valid up intersect
        }
        else {                        // start y > P.y (no test needed)
            if (V[i+1].y  <= P.y)     // a downward crossing
                 if (isLeft( V[i], V[i+1], P) < 0)  // P right of  edge
                     --wn;            // have  a valid down intersect
        }
    }
    return wn;
}
//===================================================================
```

References

Wm. Randolph Franklin, "PNPOLY - Point Inclusion in Polygon Test" Web Page (2000)

Eric Haines, "Point in Polygon Strategies" in Graphics Gems IV (1994)

Tomas Moller & Eric Haines, "Ray/Polygon Intersection" in Real-Time Rendering (3rd Edition) (2008)

Joseph O'Rourke, "Point in Polygon" in Computational Geometry in C (2nd Edition) (1998)

John M. Snyder & Alan H. Barr, "Ray Tracing Complex Models Containing Surface Tessellations", Computer Graphics 21(4), 119-126 (1987) [also in the Proceedings of SIGGRAPH 1987]

Planes

Here we present several basic methods for representing planes in 3D space, and how to compute the distance of a point to a plane.

Planes

A surface is that which has length and breadth only. [Book I, Definition 5]
The extremities of a surface are lines. [Book I, Definition 6]
A plane surface is a surface which lies evenly with the straight lines on itself. [Book I, Definition 7]

If two straight lines cut one another, they are in one plane, and every triangle is in one plane. [Book XI, Proposition 2]

If two planes cut one another, their common section is a straight line. [Book XI, Proposition 3]

From the same point two straight lines cannot be set up at right angles to the same plane on the same side. [Book XI, Proposition 13]
[Euclid, 300 BC]

Although Euclid defined a plane in Book I as the third primitive after the point and line, he did not prove anything about planes until much later in Book XI. And even then, he did not appeal to the definition from Book I in his proofs of Propositions in Book XI [Heath, 1956]. In modern times, [Coxeter, 1989a] has given a more exact definition of a plane as:

> **Definition**. If A, B, C are three non-collinear points, the ***plane*** ABC is the set of all points collinear with pairs of points on one or two sides of the triangle $\mathbf{\Delta}ABC$.

which is used to prove properties of incidence in a plane. Nevertheless, Euclid's Propositions XI.2 and XI.13 make it clear the Greeks knew that a plane is uniquely determined by any of the following data:

1. by three non-collinear points,

2. by two straight lines meeting one another,

3. by a straight line and a point not on that line, and

4. a point and a line perpendicular to the plane.

Plane Equations

Implicit Equation

Thus, there are many ways to represent a plane $\mathcal{P}$. Some methods work in any dimension, and some work only in 3D. In any dimension, one can always specify ***3 non-collinear points*** $V_0 = (x_0, y_0, z_0)$, $V_1 = (x_1, y_1, z_1)$, $V_2 = (x_2, y_2, z_2)$ as the vertices of a triangle, the most primitive planar object. In 3D, this uniquely defines the plane of points $P = (x, y, z)$ satisfying the ***implicit equation***:

$$\begin{vmatrix} x - x_0 & y - y_0 & z - z_0 \\ x_1 - x_0 & y_1 - y_0 & z_1 - z_0 \\ x_2 - x_0 & y_2 - y_0 & z_2 - z_0 \end{vmatrix} = 0$$

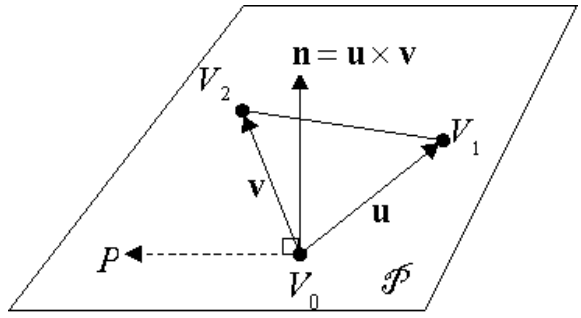
represented as a determinant.

Normal Implicit Equation

In 3 dimensions, another popular and useful representation for a plane $\mathcal{P}$is the ***normal form*** that specifies an ***origin*** point V_0 on $\mathcal{P}$and a "***normal***" vector **n** which is perpendicular to $\mathcal{P}$. This representation is useful for computing intersections, resulting in compact and efficient formulas. But, this representation only works in 3D space. In a higher *n*-dimensional space, this representation defines an (*n*–1)-dimensional linear subspace. We will not pursue this further here except to say that many of the results for planes in 3D carry over to hyperplanes of *n*-D space (see [Hanson, 1994] for further information).

In 3D, a normal vector **n** for $\mathcal{P}$can be computed from any triangle $\Delta V_0 V_1 V_2$ of points on $\mathcal{P}$ as the cross-product $\mathbf{n} = \mathbf{u} \times \mathbf{v} = (V_1 - V_0) \times (V_2 - V_0)$. Then, any point P on the plane satisfies the ***normal implicit equation***:

$$\mathbf{n} \bullet (P - V_0) = 0$$



For $\mathbf{n} = (a,b,c)$, $P = (x, y, z)$ and $d = -(\mathbf{n} \cdot V_0)$, the equation for the plane is:

$$\mathbf{n} \bullet \left(P - V_0 \right) = ax + by + cz + d = 0$$

So, the *xyz*-coefficients (a,b,c) of any linear equation for a plane $\mathcal{P}$ always give a vector which is perpendicular to the plane. Also, when $d = 0$, the plane passes through the origin $\mathbf{0} = (0,0,0)$.

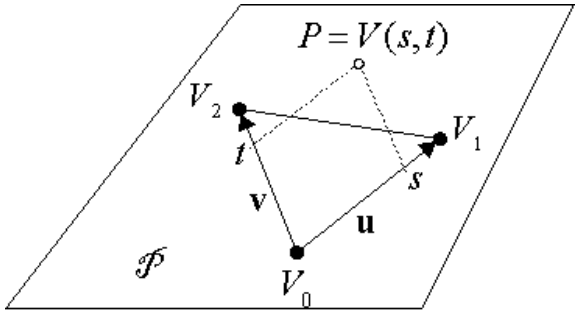
It is often useful to have a ***unit normal*** vector for the plane which simplifies some formulas. This is easily done by dividing **n** by |**n**|. Then, the associated implicit equation $ax + by + cz + d = 0$ is said to be "***normalized***". Further, when |**n**| = 1, the coordinates (a,b,c) of **n** are also the direction cosines of the angles that the vector **n** makes with the *xyz*-axes. Additionally, *d* is then the perpendicular distance from the origin **0**to the plane, as we will show in the next section.

Parametric Equation

Also in any dimension, similar to the parametric line equation, one can replace either or both of the two specified points V_1 and V_2 by direction vectors $\mathbf{u} = V_1 - V_0$ and $\mathbf{v} = V_2 - V_0$. Then, given one point V_0 on $\mathcal{P}$ and two nonparallel line direction vectors **u** and **v**, there is a natural ***parametric equation*** for ponts on the plane $\mathcal{P}$; namely:

$$V(s,t) = V_0 + s\mathbf{u} + t\mathbf{v} = (1 - s - t)V_0 + sV_1 + tV_2$$

where *s* and *t* are real numbers which are coordinates within the plane relative to the ***origin*** V_0 and the ***basis vectors*** **u** and **v**. And, when $s \geq 0$, $t \geq 0$, and $s + t \leq 1$, then $P = V(s,t)$ is inside the triangle $\mathbf{T} = \Delta V_0 V_1 V_2$.



The pair of parameters (s,t) is referred to as the ***parametric coordinate*** of $P = V(s,t)$ relative to **T**. The point *P* is on an edge of **T** whenever one of the conditions $s = 0$, $t = 0$, or $s + t = 1$ is true (each condition corresponds to one edge). And, the vertices of **T** are given by: $V_0 = V(0,0)$, $V_1 = V(1,0)$, and $V_2 = V(0,1)$.

Barycentric Coordinates

A similar representation is given by the ***barycentric coordinates*** [Coxeter, 1989b] of a point P relative to the triangle $\mathbf{T} = \Delta V_0 V_1 V_2$. This method associates a triple (b_0,b_1,b_2) with the centroid of three masses b_0, b_1, and b_2 located at the triangle's three vertices. Negative numbers are treated as negative masses (like helium balloons). This gives:

$$P(b_0,b_1,b_2) = a_0 V_0 + a_1 V_1 + a_2 V_2 \quad \text{where} \quad a_i = \frac{b_i}{b_0 + b_1 + b_2}$$

The triple (b_0,b_1,b_2) is called a "***homogeneous barycentric coordinate***" for points P relative to the triangle **T**. The normalized triple (a_0,a_1,a_2) satisfying $a_0 + a_1 + a_2 = 1$ is called the "***areal barycentric coordinate***" of P. Clearly $P(a_0,a_1,a_2) = a_0 V_0 + a_1 V_1 + a_2 V_2$ when $a_0 + a_1 + a_2 = 1$, and thus any triple (a_0,a_1,a_2) which sums to 1 is a valid areal coordinate. Further, there is a one-to-one correspondence between areal coordinates and all points on the plane $\mathcal{P}$. This can be seen by noting the equivalence of areal barycentric and parametric coordinate representations gotten by setting: $a_0 = (1–s–t)$, $a_1 = s$, and $a_2 = t$.

Areal coordinates have a number of useful properties with respect to the triangle **T**. For example, the point Plies inside **T** only when all components of its areal barycentric coordinate are nonnegative, that is $a_0 \geq 0$, $a_1 \geq 0$, and $a_2 \geq 0$. Further interesting properties, especially ones related to area ratios, are given by [Coxeter, 1989b].

Representation Conversions

One can convert from any of these representations to another when convenient. For example, given two direction vectors for lines on a plane, their cross-product gives a normal vector. Conversely, given a normal vector, one can easily find two other independent vectors perpendicular to it. For example, given $\mathbf{n} = (a,b,c)$ with $a \neq 0$, then $\mathbf{u}$ = (–b, a, 0) and $\mathbf{v}$ = (–c, 0, a) are perpendicular to $\mathbf{n}$ since both $\mathbf{n} \cdot \mathbf{u} = 0$ and $\mathbf{n} \cdot \mathbf{v} = 0$. Further, three non-collinear points on the plane are: V_0, $V_0 + \mathbf{u}$, and $V_0 + \mathbf{v}$. Most other conversions are at least as easy as these. But, one conversion is a bit more difficult; namely, finding the parametric or barycentric coordinates relative to a given triangle.

Barycentric Coordinate Computation

We want to find the parametric or barycentric coordinates (defined above) of a given 3D point $P = (x, y, z)$ relative to a triangle $\mathbf{T} = \Delta V_0 V_1 V_2$ in the plane. We start by putting $\mathbf{u} = V_1 - V_0$ and $\mathbf{v} = V_2 - V_0$ as before, as well as $\mathbf{w} = P - V_0$. Then, we find the parametric coordinates (s, t) of P as the solution of the equation: $\mathbf{w} = s\mathbf{u} + t\mathbf{v}$. This solution exists and is unique whenever P lies in the plane of **T**. Further, the areal

barycentric coordinates of $P = a_0V_0 + a_1V_1 + a_2V_2$ are: $a_0 = (1–s–t)$, $a_1 = s$, and $a_2 = t$ which satisfy: $a_0 + a_1 + a_2 = 1$.

To solve the equation $\mathbf{w} = s\mathbf{u} + t\mathbf{v}$, we first define a 3D generalization of Hill's "***perp operator"*** [Hill, 1994].

Definition. Given a 2D plane $\mathcal{P}$ embedded in 3D space with a unit normal vector **n**, and given any vector **v** in the plane (that is, **v** satisfies $\mathbf{n} \cdot \mathbf{v} = 0$), define the "***generalized perp operator***" $^\perp$ on $\mathcal{P}$ by: $\mathbf{v}^\perp = \mathbf{n} \times \mathbf{v}$.

Then, $\mathbf{v}^\perp$ is another vector in the plane $\mathcal{P}$ (since $\mathbf{n} \bullet \mathbf{v}^\perp = 0$), and it is also perpendicular to **v** (since $\mathbf{v} \bullet \mathbf{v}^\perp = 0$). Additionally, this embedded perp operator is linear for vectors in $\mathcal{P}$; that is, $(a\,\mathbf{v} + b\,\mathbf{w})^\perp = a\,\mathbf{v}^\perp + b\,\mathbf{w}^\perp$ where **v**and **w** are vectors in $\mathcal{P}$, and *a* and *b* are scalar numbers. Also, if $\mathcal{P}$ is the 2D *xy*-plane ($z = 0$) with $\mathbf{n} = (0,0,1)$, then our 3D perp operator is exactly the same 2D perp operator given by [Hill, 1994]; since we have: $(x, y, 0)^\perp = (0,0,1) \times (x, y, 0) = (-y, x, 0)$.

We will now solve the equation: $\mathbf{w} = s\mathbf{u} + t\mathbf{v}$ for *s* and *t*. First, take the dot product of both sides with $\mathbf{v}^\perp$ to get $\mathbf{w} \bullet \mathbf{v}^\perp = s\,\mathbf{u} \bullet \mathbf{v}^\perp + t\,\mathbf{v} \bullet \mathbf{v}^\perp = s\,\mathbf{u} \bullet \mathbf{v}^\perp$, and solve for *s*. Similarly, taking the dot product with $\mathbf{u}^\perp$, we get: $\mathbf{w} \bullet \mathbf{u}^\perp = s\,\mathbf{u} \bullet \mathbf{u}^\perp + t\,\mathbf{v} \bullet \mathbf{u}^\perp = t\,\mathbf{v} \bullet \mathbf{u}^\perp$, and solve for *t*. Then we have:

$$s = \frac{\mathbf{w} \bullet \mathbf{v}^\perp}{\mathbf{u} \bullet \mathbf{v}^\perp} = \frac{\mathbf{w} \bullet (\mathbf{n} \times \mathbf{v})}{\mathbf{u} \bullet (\mathbf{n} \times \mathbf{v})} \qquad \text{and} \qquad t = \frac{\mathbf{w} \bullet \mathbf{u}^\perp}{\mathbf{v} \bullet \mathbf{u}^\perp} = \frac{\mathbf{w} \bullet (\mathbf{n} \times \mathbf{u})}{\mathbf{v} \bullet (\mathbf{n} \times \mathbf{u})}$$

The denominators are nonzero whenever the triangle **T** is nondegenerate (that is, has a nonzero area). When **T**is degenerate, it is either a segment or a point, and in either case does not uniquely define a plane.

Altogether we have used 3 cross products (one to compute $\mathbf{n} = \mathbf{u} \times \mathbf{v}$) which is a lot of computation. But, we can simplify this with the formula for left association of the cross product; namely, for any three 3D vectors **a**, **b**, and **c**, then $(\mathbf{a} \times \mathbf{b}) \times \mathbf{c} = (\mathbf{a} \bullet \mathbf{c})\mathbf{b} - (\mathbf{b} \bullet \mathbf{c})\mathbf{a}$. [Note: the cross product is not associative, and so there is a different (but similar) formula for right association]. Applying this formula results in the simplifications:

$$\mathbf{u}^\perp = \mathbf{n} \times \mathbf{u} = (\mathbf{u} \times \mathbf{v}) \times \mathbf{u} = (\mathbf{u} \cdot \mathbf{u})\mathbf{v} - (\mathbf{u} \cdot \mathbf{v})\mathbf{u}$$
$$\mathbf{v}^\perp = \mathbf{n} \times \mathbf{v} = (\mathbf{u} \times \mathbf{v}) \times \mathbf{v} = (\mathbf{u} \cdot \mathbf{v})\mathbf{v} - (\mathbf{v} \cdot \mathbf{v})\mathbf{u}$$

We can now compute the solutions for *s* and *t* using only dot products as:

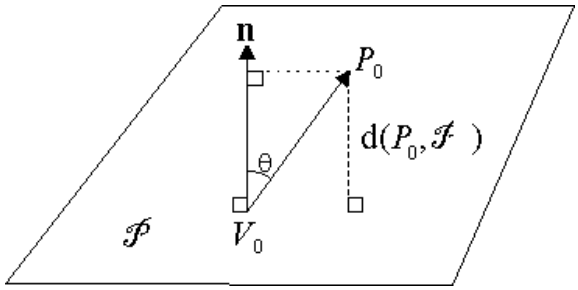
$$s = \frac{(\mathbf{u} \bullet \mathbf{v})(\mathbf{w} \bullet \mathbf{v}) - (\mathbf{v} \bullet \mathbf{v})(\mathbf{w} \bullet \mathbf{u})}{(\mathbf{u} \bullet \mathbf{v})^2 - (\mathbf{u} \bullet \mathbf{u})(\mathbf{v} \bullet \mathbf{v})}$$

$$t = \frac{(\mathbf{u}\cdot\mathbf{v})(\mathbf{w}\cdot\mathbf{u}) - (\mathbf{u}\cdot\mathbf{u})(\mathbf{w}\cdot\mathbf{v})}{(\mathbf{u}\cdot\mathbf{v})^2 - (\mathbf{u}\cdot\mathbf{u})(\mathbf{v}\cdot\mathbf{v})}$$

with 5 distinct dot products. The two denominators are the same and only need to be calculated once.

Distance of a Point to a Plane

The distance d(P_0,$\mathcal{P}$) from an arbitrary 3D point $P_0 = (x_0, y_0, z_0)$ to the plane $\mathcal{P}$ given by $ax + by + cz + d = 0$, can be computed by using the dot product to get the projection of the vector $(P_0 - V_0)$ onto **n** as shown in the diagram:



which results in the formula:

$$\mathrm{d}(P_0, P) = |P_0 - V_0|\cos\theta = \frac{\mathbf{n}\cdot(P_0 - V_0)}{|\mathbf{n}|} = \frac{ax_0 + by_0 + cz_0 + d}{\sqrt{a^2 + b^2 + c^2}}$$

When |**n**| = 1, this formula simplifies to:

$$\mathrm{d}(P_0, \mathcal{P}) = ax_0 + by_0 + cz_0 + d$$

showing that *d* is the distance from the origin **0** = (0,0,0) to the plane $\mathcal{P}$.

This formula gives a *signed* distance which is positive on one side of the plane and negative on the other. So, one has to take the absolute value to get an absolute distance. Otherwise, the distance is positive for points on the side pointed to by the normal vector **n**. Because of this, the sign of d(P_0,$\mathcal{P}$) can be used to simply test which side of the plane a point is on. For example, if P_0P_1 is a finite line segment, then it intersects $\mathcal{P}$ only when the two endpoints are on opposite sides of the plane; that is, if $\mathrm{d}(P_0, \mathcal{P})\,\mathrm{d}(P_1, \mathcal{P}) < 0$. Conversely, when $\mathrm{d}(P_0, \mathcal{P})\,\mathrm{d}(P_1, \mathcal{P}) > 0$, there cannot be an intersection. Also, if $\mathrm{d}(P_0, \mathcal{P})\,\mathrm{d}(P_1, \mathcal{P}) = 0$, then at least one of the endpoints is on $\mathcal{P}$. When both points are on $\mathcal{P}$, the whole segment P_0P_1 lies in the plane.

To compute the distance to a plane $\mathcal{P}$, we did not calculate the base point of the perpendicular from the point P_0 to $\mathcal{P}$, which some authors do. If one just wants the distance, then directly computing it without going through an intermediate calculation is fastest.

Nevertheless, there are situations where one wants to know the orthogonal (perpendicular) projection of P_0onto $\mathcal{P}$. It can be computed by taking a line through P_0 that is perpendicular to $\mathcal{P}$ (that is, one which is parallel to **n**), and computing it's intersection with the plane. The simplest such line is given by: $P(s) = P_0 + s\mathbf{n}$. This line intersects $\mathcal{P}$ when $P(s)$ satisfies the equation of the plane; namely, $\mathbf{n} \bullet (P(s) - V_0) = 0$. Solving this for s at the intersection point, we get:

$$s_{\mathcal{P}} = \frac{-\mathbf{n} \cdot (P_0 - V_0)}{\mathbf{n} \cdot \mathbf{n}} = \frac{-(ax_0 + by_0 + cz_0 + d)}{a^2 + b^2 + c^2} = \frac{-\mathrm{d}(P_0, \mathcal{P})}{|\mathbf{n}|}$$

And the base of the perpendicular is the intersection point:

$$P_{\mathcal{P}} = P_0 - \frac{(ax_0 + by_0 + cz_0 + d)}{a^2 + b^2 + c^2}\mathbf{n}$$

For the special case when $P_0 = \mathbf{0} = (0,0,0)$, one has $P_{\mathcal{P}} = -d\,\mathbf{n} / |\mathbf{n}|^2$ as the orthogonal projection of the origin onto the plane. Then $\mathrm{d}(\mathbf{0}, \mathcal{P}) = \mathrm{d}(\mathbf{0}, P_{\mathcal{P}}) = |\,\mathbf{0} - P_{\mathcal{P}}\,| = d\,/\,|\mathbf{n}|$, and when **n** is a unit normal $\mathrm{d}(\mathbf{0}, \mathcal{P}) = d$.

Implementations

Here are some sample "C++" implementations of these algorithms.

```
// Copyright 2001, 2012, 2021 Dan Sunday
// This code may be freely used and modified for any purpose
// providing that this copyright notice is included with it.
// There is no warranty for this code, and the author of it cannot
// be held liable for any real or imagined damage 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;}
//        operators for:
//            Point  = Point ± Vector
//            Vector = Point - Point
//            Vector = Scalar * Vector    (scalar product)
//    Plane with a point and a normal vector {Point V0; Vector  n;}
```

```
//===================================================================


// 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(P,Q)      norm(P-Q)        // distance = norm of difference


// dist_Point_to_Plane(): get the distance from a point to a plane
//    Input:  P  = a 3D point
//            PL = a  plane with point V0 and normal n
//    Output: *B = base point on PL of perpendicular from P
//    Return: the distance from P to the plane PL
float
dist_Point_to_Plane( Point P, Plane PL, Point* B)
{
    float    sb, sn, sd;

    sn = -dot( PL.n, (P - PL.V0));
    sd = dot(PL.n, PL.n);
    sb = sn / sd;

    *B = P + sb * PL.n;
    return d(P, *B);
}
//===================================================================
```

References

Donald Coxeter, "Planes and Hyperplanes" in Introduction to Geometry (2nd Edition) (1989)

Donald Coxeter, "Barycentric Coordinates" in Introduction to Geometry (2nd Edition) (1989)

Euclid, The Elements (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)

Thomas Heath, The Thirteen Books of Euclid's Elements, Vol 3 (Books X-XIII) (1956)

Francis Hill, "The Pleasures of 'Perp Dot' Products" in Graphics Gems IV (1994)

Line and Plane Intersections

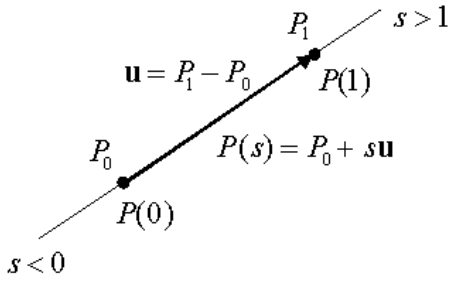
The intersection of geometric primitives is a fundamental construct in many computer graphics and modeling applications ([Foley et al, 1996], [O'Rourke, 1998]). Here we look at the algorithms for the simplest 2D and 3D linear primitives: lines, segments and planes.

Line and Segment Intersections

For computing intersections of lines and segments in 2D and 3D, it is best to use the parametric equation representation for lines. In any dimension, the parametric equation of a line defined by two points P_0 and P_1 can be represented as:

$$P(s) = P_0 + s(P_1 - P_0) = P_0 + s\mathbf{u},$$

where the parameter s is a real number and $\mathbf{u} = P_1 - P_0$ is a line direction vector. Using this representation $P(0) = P_0$, $P(1) = P_1$, and when $0 \le s \le 1$, $P(s)$ is a point on the finite segment P_0P_1 where s is the fraction of $P(s)$'s distance along the segment. That is, s = d(P_0,$P(s)$) / d(P_0,P_1). Further, if $s < 0$ then $P(s)$ is outside the segment on the P_0 side, and if $s > 1$ then $P(s)$ is outside the segment on the P_1 side.



Let two lines be given by: $P(s) = P_0 + s(P_1 - P_0) = P_0 + s\mathbf{u}$ and $Q(t) = Q_0 + t(Q_1 - Q_0) = Q_0 + t\mathbf{v}$, either or both of which could be infinite, a ray, or a finite segment.

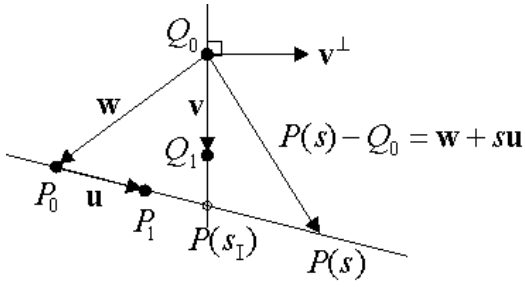
Parallel Lines

These lines are parallel when and only when their directions are collinear, namely when the two vectors $\mathbf{u} = P_1 - P_0$ and $\mathbf{v} = Q_1 - Q_0$ are linearly related as $\mathbf{u} = a\mathbf{v}$ for some real number a. For $\mathbf{u} = (u_i)$ and $\mathbf{v} = (v_i)$, this means that all ratios u_i / v_i have the value a, or that $u_1 / v_1 = u_i / v_i$ for all i. This is equivalent to the conditions that all $u_1 v_i - u_i v_1 = 0$. In 2D, with $\mathbf{u} = (u_1, u_2)$ and $\mathbf{v} = (v_1, v_2)$, this is the ***perp product*** [Hill, 1994] condition that $\mathbf{u}^\perp \bullet \mathbf{v} = u_1 v_2 - u_2 v_1 = 0$ where $\mathbf{u}^\perp = (-u_2, u_1)$, the ***perp operator***, is perpendicular to $\mathbf{u}$. This condition says that two vectors in the Euclidean plane are parallel when they are both perpendicular to the same direction vector $\mathbf{u}^\perp$. When true, the two associated lines are either coincident or do not intersect at all.

Coincidence is easily checked by testing if a point on one line, say P_0, also lies on the other line $Q(t)$. That is, there exists a number t_0 such that: $P_0 = Q(t_0) = Q_0 + t_0\mathbf{v}$. If $\mathbf{w} = (w_i) = P_0 - Q_0$, then this is equivalent to the condition that $\mathbf{w} = t_0\mathbf{v}$ for some t_0 which is the same as $w_1 v_i - w_i v_1 = 0$ for all i. In 2D, this is another perp product condition: $\mathbf{w}^{\perp} \bullet \mathbf{v} = w_1 v_2 - w_2 v_1 = 0$. If this condition holds, one has $t_0 = w_1 / v_1$, and the infinite lines are coincident. And if one line (but not the other) is a finite segment, then it is the coincident intersection. However, if both lines are finite segments, then they may (or may not) overlap. In this case, solve for t_0 and $t_1$such that $P_0 = Q(t_0)$ and $P_1 = Q(t_1)$. If the segment intervals $[t_0,t_1]$ and $[0,1]$ are disjoint, there is no intersection. Otherwise, intersect the intervals (using max and min operations) to get $[r_0, r_1] = [t_0, t_1] \cap [0,1]$. Then the intersection segment is $Q(r_0)Q(r_1) = P_0P_1 \cap Q_0Q_1$. This works in any dimension.

Non-Parallel Lines

When the two lines or segments are not parallel, they might intersect in a unique point. In 2D Euclidean space, infinite lines always intersect. In higher dimensions they usually miss each other and do not intersect. But if they intersect, then their linear projections onto a 2D plane will also intersect. So, one can simply restrict to two coordinates, for which **u** and **v** are not parallel, compute the 2D intersection point I at $P(s_I)$ and $Q(t_I)$ for those two coordinates, and then test if $P(s_I) = Q(t_I)$ for all coordinates. To compute the 2D intersection point, consider the two lines and the associated vectors in the diagram:



To determine s_I, we have the vector equality $P(s) - Q_0 = \mathbf{w} + s\mathbf{u}$ where $\mathbf{w} = P_0 - Q_0$. At the intersection, the vector $P(s_I) - Q_0$ is perpendicular to $\mathbf{v}^{\perp}$, and this is equivalent to the perp product condition that $\mathbf{v}^{\perp} \bullet (\mathbf{w} + s_I\mathbf{u}) = 0$. Solving this equation, we get:

$$s_I = \frac{-\mathbf{v}^{\perp} \cdot \mathbf{w}}{\mathbf{v}^{\perp} \cdot \mathbf{u}} = \frac{v_2 w_1 - v_1 w_2}{v_1 u_2 - v_2 u_1}$$

Note that the denominator $\mathbf{v}^{\perp} \bullet \mathbf{u} = 0$ only when the lines are parallel as previously discussed. Similarly, solving for $Q(t_I)$, we get:

$$t_I = \frac{\mathbf{u}^{\perp} \cdot \mathbf{w}}{\mathbf{u}^{\perp} \cdot \mathbf{v}} = \frac{u_1 w_2 - u_2 w_1}{u_1 v_2 - u_2 v_1}$$

The denominators are the same up to sign, since $\mathbf{u}^{\perp} \bullet \mathbf{v} = -\mathbf{v}^{\perp} \bullet \mathbf{u}$, and should only be computed once if we want to know both s_I and t_I. However, knowing either is enough to get the intersection point $I = P(s_I) = Q(t_I)$.

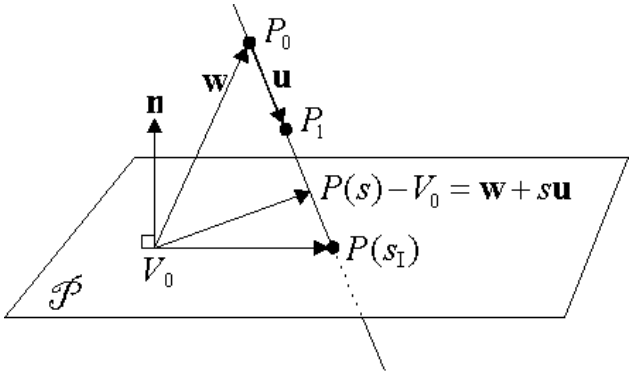
Further, if one of the two lines is a finite segment (or a ray), say P_0P_1, then the intersect point is in the segment only when $0 \le s_I \le 1$ (or $s_I \ge 0$ for a ray). If both lines are segments, then both solution parameters, $s_I$and t_I, must be in the [0,1] interval for the segments to intersect. Although this sounds simple enough, the code for the intersection of two segments is a bit delicate since many special cases need to be checked (see our implementation intersect2D_2Segments()).

Plane Intersections

Line-Plane Intersection

In 3D, a line **L** is either parallel to a plane $\mathcal{P}$ or intersects it in a single point. Let **L** be given by the parametric equation: $P(s) = P_0 + s(P_1 - P_0) = P_0 + s\mathbf{u}$, and the plane $\mathcal{P}$ be given by a point V_0 on it and a normal vector $\mathbf{n} = (a,b,c)$. We first check if **L** is parallel to $\mathcal{P}$ by testing if $\mathbf{n} \bullet \mathbf{u} = 0$, which means that the line direction vector **u** is perpendicular to the plane normal **n**. If this is true, then **L** and $\mathcal{P}$ are parallel and either never intersect or else **L** lies totally in the plane $\mathcal{P}$. Disjointness or coincidence can be determined by testing whether any one specific point of **L**, say P_0, is contained in $\mathcal{P}$, that is whether it satisfies the implicit line equation: $\mathbf{n} \bullet (P_0 - V_0) = 0$.

If the line and plane are not parallel, then **L** and $\mathcal{P}$ intersect in a unique point $P(s_I)$ which is computed using a method similar to the one for the intersection of two lines in 2D. Consider the diagram:



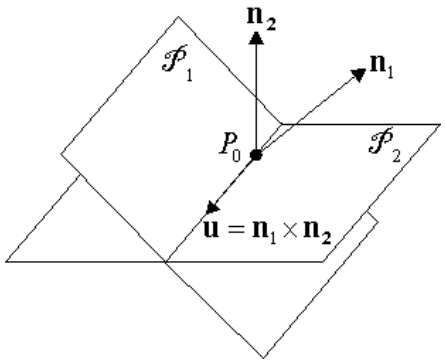
At the intersect point, the vector $P(s) - V_0 = \mathbf{w} + s\mathbf{u}$ is perpendicular to **n,** where $\mathbf{w} = P_0 - V_0$. This is equivalent to the dot product condition: $\mathbf{n} \bullet (\mathbf{w} + s\mathbf{u}) = 0$. Solving we get:

$$s_I = \frac{-\mathbf{n} \cdot \mathbf{w}}{\mathbf{n} \cdot \mathbf{u}} = \frac{\mathbf{n} \cdot (\mathrm{V}_0 - \mathrm{P}_0)}{\mathbf{n} \cdot (\mathrm{P}_1 - \mathrm{P}_0)} = \frac{-(ax_0 + by_0 + cz_0 + d)}{\mathbf{n} \cdot \mathbf{u}}$$

If the line **L** is a finite segment from P_0 to P_1, then one just has to check that $0 \le s_I \le 1$ to verify that there is an intersection between the segment and the plane. For a positive ray, there is an intersection with the plane when $s_I \ge 0$.

Intersection of 2 Planes

In 3D, two planes $\mathcal{P}_1$ and $\mathcal{P}_2$ are either parallel or they intersect in a single straight line **L**. Let $\mathcal{P}_i$ (i = 1,2) be given by a point V_i and a normal vector $\mathbf{n}_i$, and have an implicit equation: $\mathbf{n}_i \cdot P + d_i = 0$, where $P = (x, y, z)$. The planes $\mathcal{P}_1$ and $\mathcal{P}_2$ are parallel whenever their normal vectors $\mathbf{n}_1$ and $\mathbf{n}_2$ are parallel, and this is equivalent to the condition that: $\mathbf{n}_1 \times \mathbf{n}_2 = 0$. In software, one would test if $|\mathbf{n}_1 \times \mathbf{n}_2| \le \delta$ where division by $\delta > 0$ would cause overflow, and uses this as the robust condition for $\mathbf{n}_1$ and $\mathbf{n}_2$ to be parallel in practice. When not parallel, $\mathbf{u} = \mathbf{n}_1 \times \mathbf{n}_2$ is a direction vector for the intersection line **L** since **u** is perpendicular to both $\mathbf{n}_1$ and $\mathbf{n}_2$, and thus is parallel to both planes as shown in the following diagram. If |**u**| is small, it is best to scale it so that |**u**| = 1, making **u** a unit direction vector.



After computing $\mathbf{n}_1 \times \mathbf{n}_2$ (3 adds + 6 multiplies), to fully determine the intersection line, we still need to find a specific point on it. That is, we need to find a point $P_0 = (x_0, y_0, z_0)$ that lies in both planes. We can do this by finding a common solution of the implicit equations for $\mathcal{P}_1$ and $\mathcal{P}_2$. But there are only two equations in the 3 unknowns since the point P_0 can lie anywhere on the 1-dimensional line **L**. So we need another constraint to solve for a specific P_0. There are a number of ways this could be done:

(A) Direct Linear Equation. One could set one coordinate to zero, say $z_0 = 0$, and then solve for the other two. But this will only work when **L** intersects the plane $z_0 = 0$. This is will be true when the z-coordinate $u_z$of $\mathbf{u} = (u_x, u_y, u_z)$ is nonzero. So, one must first select a nonzero coordinate of **u**, and then set the corresponding coordinate of P_0 to 0. Further, one should choose the coordinate with the largest absolute value, as this will give the most robust computations. Suppose that $u_z \ne 0$, then $P_0 = (x_0, y_0, 0)$ lies on **L**. Solving the two equations: $a_1 x_0 + b_1 y_0 + d_1 = 0$ and $a_2 x_0 + b_2 y_0 + d_2 = 0$, one gets:

$$x_0 = \frac{b_1 d_2 - b_2 d_1}{a_1 b_2 - a_2 b_1} \quad \textbf{and} \quad y_0 = \frac{a_2 d_1 - a_1 d_2}{a_1 b_2 - a_2 b_1}$$

and the parametric equation of **L** is:

$$P(s)=\frac{\left(\begin{vmatrix} b_1 & b_2 \\ d_1 & d_2 \end{vmatrix}, \begin{vmatrix} d_1 & d_2 \\ a_1 & a_2 \end{vmatrix}, 0\right)}{\begin{vmatrix} a_1 & a_2 \\ b_1 & b_2 \end{vmatrix}}+s\left(\mathbf{n}_1\times\mathbf{n}_2\right)$$

The denominator here is equal to the non-zero 3rd coordinate of **u**. So, ignoring the test for a large nonzero coordinate, and counting division as a multiplication, the total number of operations for this solution = 5 adds + 13 multiplies.

(B) Line Intersect Point. If one knows a specific line in one plane (for example, two points in the plane), and this line intersects the other plane, then its point of intersection, I, will lie in both planes. Thus, it is on the line of intersection for the two planes, and the parametric equation of **L** is: $P(s) = I + s\ (\mathbf{n}_1 \text{ x } \mathbf{n}_2)$. To compute $\mathbf{n}_1\times\mathbf{n}_2$ and the intersection point (given the line), the total number of operations = 11 adds + 19 multiplies.

One way of constructing a line in one plane that must intersect the other plane is to project one plane's normal vector onto the other plane. This gives a line that must always be orthogonal to the line of the planes' intersection. So, the projection of $\mathbf{n}_2$ on $\mathscr{P}_1$ defines a line that intersects $\mathscr{P}_2$ in the sought for point P_0 on **L**. More specifically, project the two points **0** = (0,0,0) and $\mathbf{n}_2 = (nx_2, ny_2, nz_2)$ to $\mathscr{P}_1(\mathbf{0})$ and $\mathscr{P}_1(\mathbf{n}_2)$ respectively. Then the projected line in $\mathscr{P}_1$ is $\mathbf{L}_1$: $Q(t) = \mathscr{P}_1(\mathbf{0}) + t\ (\mathscr{P}_1(\mathbf{n}_2) - \mathscr{P}_1(\mathbf{0}))$, and intersection of it with $\mathscr{P}_2$ can be computed. In the most efficient case, where both $\mathbf{n}_1$ and $\mathbf{n}_2$ are unit normal vectors and the constant $\mathscr{P}_1(\mathbf{0})$ is pre-stored, the total operations = 17 adds + 22 multiplies.

(C) 3 Plane Intersect Point. Another method selects a third plane $\mathscr{P}_3$ with an implicit equation $\mathbf{n}_3 \cdot P = 0$ where $\mathbf{n}_3 = \mathbf{n}_1 \text{ x } \mathbf{n}_2$ and $d_3 = 0$ (meaning it passes through the origin). This always works since: (1) **L** is perpendicular to $\mathscr{P}_3$ and thus intersects it, and (2) the vectors $\mathbf{n}_1$, $\mathbf{n}_2$, and $\mathbf{n}_3$ are linearly independent. Thus the planes $\mathscr{P}_1$, $\mathscr{P}_2$ and $\mathscr{P}_3$ intersect in a unique point P_0 which must be on **L**. Using the formula for the intersection of 3 planes (see the next section), where $d_3 = 0$ for $\mathscr{P}_3$, we get:

$$P_0=\frac{-d_1(\mathbf{n}_2\times\mathbf{n}_3)-d_2(\mathbf{n}_3\times\mathbf{n}_1)}{\mathbf{n}_1\cdot(\mathbf{n}_2\times\mathbf{n}_3)}=\frac{(d_2\mathbf{n}_1-d_1\mathbf{n}_2)\times\mathbf{n}_3}{(\mathbf{n}_1\times\mathbf{n}_2)\cdot\mathbf{n}_3}$$
$$=\frac{(d_2\mathbf{n}_1-d_1\mathbf{n}_2)\times(\mathbf{n}_1\times\mathbf{n}_2)}{|\mathbf{n}_1\times\mathbf{n}_2|^2}$$

and the parametric equation of **L** is:

$$P(s)=\frac{(d_2\mathbf{n}_1-d_1\mathbf{n}_2)\times\mathbf{u}}{|\mathbf{u}|^2}+s\mathbf{u}\ ,\ \text{where}\ \ \mathbf{u}=\mathbf{n}_1\times\mathbf{n}_2$$

The number of operations for this solution = 11 adds + 23 multiplies.

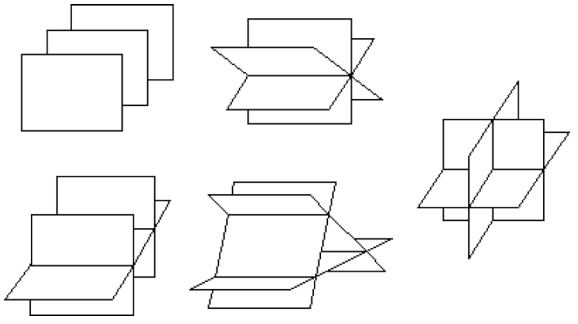
The bottom line is that ***the most efficient method is the direct solution*** **(A)** that uses only 5 adds + 13 multiplies to compute the equation of the intersection line.

Intersection of 3 Planes

In 3D, three planes $\mathcal{P}_1$, $\mathcal{P}_2$ and $\mathcal{P}_3$ can intersect (or not) in the following ways:

Geometric Relation	Intersection	Algebraic Condition
All 3 planes are parallel		$\mathbf{n}_j \times \mathbf{n}_k = 0$ for all $j \neq k$
They coincide	A plane	$\mathbf{n}_1 \bullet V_1 = \mathbf{n}_1 \bullet V_2 = \mathbf{n}_1 \bullet V_3$
They are disjoint	None	$\mathbf{n}_1 \bullet V_1 \neq \mathbf{n}_1 \bullet V_2 \neq \mathbf{n}_1 \bullet V_3 \neq \mathbf{n}_1 \bullet V_1$
Only two planes are parallel, and the 3rd plane cuts both of them [Note: the 2 parallel planes may coincide]	2 parallel lines [if the two planes coincide => 1 line]	Only one $\mathbf{n}_j \times \mathbf{n}_k = 0$ for $j \neq k$
No two planes are parallel, so pairwise they intersect in 3 lines		$\mathbf{n}_j \times \mathbf{n}_k \neq 0$ for all $j \neq k$
The intersect lines are parallel		$\mathbf{n}_1 \bullet (\mathbf{n}_2 \times \mathbf{n}_3) = 0$
They all coincide	1 line	Test a point of one line with another line
They are disjoint	3 parallel lines	Same test fails => do not coincide
No intersect lines are parallel	A unique point	$\mathbf{n}_1 \bullet (\mathbf{n}_2 \times \mathbf{n}_3) \neq 0$

As shown in the illustrations:



One should first test for the most frequent case of a unique intersect point, namely that $\mathbf{n}_1 \bullet (\mathbf{n}_2 \times \mathbf{n}_3) \neq 0$, since this excludes all the other cases. When the intersection is a unique point, it is given by the formula:

$$P_0 = \frac{-d_1(\mathbf{n}_2 \times \mathbf{n}_3) - d_2(\mathbf{n}_3 \times \mathbf{n}_1) - d_3(\mathbf{n}_1 \times \mathbf{n}_2)}{\mathbf{n}_1 \bullet (\mathbf{n}_2 \times \mathbf{n}_3)}$$

which can verified by showing that this P_0 satisfies the parametric equations for all planes $\mathcal{P}_1$, $\mathcal{P}_2$ and $\mathcal{P}_3$.

However, there can be a problem with the robustness of this computation when the denominator $\mathbf{n}_1 \bullet (\mathbf{n}_2 \times \mathbf{n}_3)$ is very small. In that case, it would be best to get a robust line of intersection for two of the planes, and then compute the point where this line intersects the third plane.

Implementations

Here are some sample "C++" implementations of these algorithms.

```
// Copyright 2001, 2012, 2021 Dan Sunday
// This code may be freely used and modified for any purpose
// providing that this copyright notice is included with it.
// There is no warranty for this code, and the author of it cannot
// be held liable for any real or imagined damage 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;}
//        operators for:
//            == to test  equality
//            != to test  inequality
//            Point   = Point ± Vector
//            Vector =  Point - Point
//            Vector =  Scalar * Vector    (scalar product)
//            Vector =  Vector * Vector    (3D cross product)
//    Line and Ray and Segment with defining  points {Point P0, P1;}
//        (a Line is infinite, Rays and  Segments start at P0)
//        (a Ray extends beyond P1, but a  Segment ends at P1)
//    Plane with a point and a normal {Point V0; Vector  n;}
//===================================================================


#define SMALL_NUM   0.00000001 // anything that avoids division overflow
// 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 perp(u,v)  ((u).x * (v).y - (u).y * (v).x)  // perp product  (2D


// intersect2D_2Segments(): find the 2D intersection of 2 finite segments
//    Input:  two finite segments S1 and S2
//    Output: *I0 = intersect point (when it exists)
//            *I1 =  endpoint of intersect segment [I0,I1] (when it exists)
//    Return: 0=disjoint (no intersect)
//            1=intersect  in unique point I0
//            2=overlap  in segment from I0 to I1
int
intersect2D_2Segments( Segment S1, Segment S2, Point* I0, Point* I1 )
```

```
{
    Vector    u = S1.P1 - S1.P0;
    Vector    v = S2.P1 - S2.P0;
    Vector    w = S1.P0 - S2.P0;
    float     D = perp(u,v);

    // test if  they are parallel (includes either being a point)
    if (fabs(D) < SMALL_NUM) {           // S1 and S2 are parallel
        if (perp(u,w) != 0 || perp(v,w) != 0)  {
            return 0;                    // they are NOT collinear
        }
        // they are collinear or degenerate
        // check if they are degenerate  points
        float du = dot(u,u);
        float dv = dot(v,v);
        if (du==0 && dv==0) {            // both segments are points
            if (S1.P0 !=  S2.P0)         // they are distinct  points
                 return 0;
            *I0 = S1.P0;                 // they are the same point
            return 1;
        }
        if (du==0) {                     // S1 is a single point
            if  (inSegment(S1.P0, S2) == 0)  // but is not in S2
                 return 0;
            *I0 = S1.P0;
            return 1;
        }
        if (dv==0) {                     // S2 a single point
            if  (inSegment(S2.P0, S1) == 0)  // but is not in S1
                 return 0;
            *I0 = S2.P0;
            return 1;
        }
        // they are collinear segments - get  overlap (or not)
        float t0, t1;                    // endpoints of S1 in eqn for S2
        Vector w2 = S1.P1 - S2.P0;
        if (v.x != 0) {
                 t0 = w.x / v.x;
                 t1 = w2.x / v.x;
        }
        else {
                 t0 = w.y / v.y;
                 t1 = w2.y / v.y;
        }
        if (t0 > t1) {                   // must have t0 smaller than t1
                 float t=t0; t0=t1; t1=t;    // swap if not
        }
        if (t0 > 1 || t1 < 0) {
            return 0;      // NO overlap
        }
        t0 = t0<0? 0 : t0;               // clip to min 0
        t1 = t1>1? 1 : t1;               // clip to max 1
        if (t0 == t1) {                  // intersect is a point
            *I0 = S2.P0 +  t0 * v;
            return 1;
        }

        // they overlap in a valid subsegment
```

```
        *I0 = S2.P0 + t0 * v;
        *I1 = S2.P0 + t1 * v;
        return 2;
    }

    // the segments are skew and may intersect in a point
    // get the intersect parameter for S1
    float     sI = perp(v,w) / D;
    if (sI < 0 || sI > 1)               // no intersect with S1
        return 0;

    // get the intersect parameter for S2
    float     tI = perp(u,w) / D;
    if (tI < 0 || tI > 1)               // no intersect with S2
        return 0;

    *I0 = S1.P0 + sI * u;               // compute S1 intersect point
    return 1;
}
//==================================================================



// inSegment(): determine if a point is inside a segment
//    Input:  a point P, and a collinear segment S
//    Return: 1 = P is inside S
//            0 = P is  not inside S
int
inSegment( Point P, Segment S)
{
    if (S.P0.x != S.P1.x) {    // S is not  vertical
        if (S.P0.x <= P.x && P.x <= S.P1.x)
            return 1;
        if (S.P0.x >= P.x && P.x >= S.P1.x)
            return 1;
    }
    else {    // S is vertical, so test y  coordinate
        if (S.P0.y <= P.y && P.y <= S.P1.y)
            return 1;
        if (S.P0.y >= P.y && P.y >= S.P1.y)
            return 1;
    }
    return 0;
}
//==================================================================



// intersect3D_SegmentPlane(): get the intersect of a segment and plane
//    Input:  S = a segment, and Pn = a plane = {Point V0;  Vector n;}
//    Output: *I0 = the intersect point (when it exists)
//    Return: 0 = disjoint (no intersection)
//            1 =  intersection in the unique point *I0
//            2 = the  segment lies in the plane
int
intersect3D_SegmentPlane( Segment S, Plane Pn, Point* I )
{
    Vector    u = S.P1 - S.P0;
```

```
    Vector    w = S.P0 - Pn.V0;

    float     D = dot(Pn.n, u);
    float     N = -dot(Pn.n, w);

    if (fabs(D) < SMALL_NUM) {           // segment is parallel to plane
        if (N == 0)                      // segment lies in plane
            return 2;
        else
            return 0;                    // no intersection
    }
    // they are not parallel
    // compute intersect param
    float sI = N / D;
    if (sI < 0 || sI > 1)
        return 0;                        // no intersection

    *I = S.P0 + sI * u;                  // compute segment intersect point
    return 1;
}
//==================================================================



// intersect3D_2Planes(): find the 3D intersection of two planes
//    Input:  two planes Pn1 and Pn2
//    Output: *L = the intersection line (when it exists)
//    Return: 0 = disjoint (no intersection)
//            1 = the two  planes coincide
//            2 =  intersection in the unique line *L
int
intersect3D_2Planes( Plane Pn1, Plane Pn2, Line* L )
{
    Vector   u = Pn1.n * Pn2.n;          // cross product
    float    ax = (u.x >= 0 ? u.x : -u.x);
    float    ay = (u.y >= 0 ? u.y : -u.y);
    float    az = (u.z >= 0 ? u.z : -u.z);

    // test if the two planes are parallel
    if ((ax+ay+az) < SMALL_NUM) {        // Pn1 and Pn2 are near parallel
        // test if disjoint or coincide
        Vector   v = Pn2.V0 -  Pn1.V0;
        if (dot(Pn1.n, v) == 0)          // Pn2.V0 lies in Pn1
            return 1;                    // Pn1 and Pn2 coincide
        else
            return 0;                    // Pn1 and Pn2 are disjoint
    }

    // Pn1 and Pn2 intersect in a line
    // first determine max abs coordinate of cross product
    int      maxc;                       // max coordinate
    if (ax > ay) {
        if (ax > az)
             maxc =  1;
        else maxc = 3;
    }
    else {
        if (ay > az)
```

```
            maxc =  2;
        else maxc = 3;
    }

    // next, to get a point on the intersect line
    // zero the max coord, and solve for the other two
    Point    iP;                // intersect point
    float    d1, d2;            // the constants in the 2 plane equations
    d1 = -dot(Pn1.n, Pn1.V0);  // note: could be pre-stored  with plane
    d2 = -dot(Pn2.n, Pn2.V0);  // ditto

    switch (maxc) {             // select max coordinate
    case 1:                     // intersect with x=0
        iP.x = 0;
        iP.y = (d2*Pn1.n.z - d1*Pn2.n.z) /  u.x;
        iP.z = (d1*Pn2.n.y - d2*Pn1.n.y) /  u.x;
        break;
    case 2:                     // intersect with y=0
        iP.x = (d1*Pn2.n.z - d2*Pn1.n.z) /  u.y;
        iP.y = 0;
        iP.z = (d2*Pn1.n.x - d1*Pn2.n.x) /  u.y;
        break;
    case 3:                     // intersect with z=0
        iP.x = (d2*Pn1.n.y - d1*Pn2.n.y) /  u.z;
        iP.y = (d1*Pn2.n.x - d2*Pn1.n.x) /  u.z;
        iP.z = 0;
    }
    L->P0 = iP;
    L->P1 = iP + u;
    return 2;
}
//===================================================================
```

References

James Foley, Andries van Dam, Steven Feiner & John Hughes, "Clipping Lines" in Computer Graphics (3rd Edition) (2013)

Joseph O'Rourke, "Search and Intersection" in Computational Geometry in C (2nd Edition) (1998)

Ray and Triangle Intersections

We will now extend the line and plane intersection algorithms to include 3D triangles which are common elements of 3D surface and polyhedron models. We only consider transversal intersections where the two intersecting objects do not lie in the same plane. Ray and triangle intersection computation is perhaps the most frequent nontrivial operation in computer graphics rendering using ray tracing. Because of its importance, there are several published algorithms for this problem (see: [Badouel, 1990], [Moller & Trumbore, 1997], [O'Rourke, 1998], [Moller & Haines, 1999]). We present an improvement of these algorithms for ray (or segment) and triangle intersection. We also give algorithms for triangle-plane and triangle-triangle intersection.

Intersection of a Ray/Segment with a Plane

Assume we have a ray **R** (or segment **S**) from P_0 to P_1, and a plane $\mathcal{P}$through V_0 with normal **n**. The intersection of the parametric line **L**: $P(r) = P_0 + r(P_1 - P_0)$ and the plane $\mathcal{P}$ occurs at the point $P(r_I)$ with parameter value:

$$r_I = \frac{\mathbf{n} \bullet (V_0 - P_0)}{\mathbf{n} \bullet (P_1 - P_0)}$$

When the denominator $\mathbf{n} \bullet (P_1 - P_0) = 0$, the line **L** is parallel to the plane $\mathcal{P}$, and thus either does not intersect it or else lies completely in the plane (whenever either P_0 or P_1 is in $\mathcal{P}$). Otherwise, when the denominator is nonzero and r_I is a real number, then the ray **R** intersects the plane $\mathcal{P}$ only when $r_I \geq 0$. A segment **S** intersects $\mathcal{P}$ only if $0 \leq r_I \leq 1$. In all algorithms, the additional test $r_I \leq 1$ is the only difference for a segment instead of a ray.

Intersection of a Ray/Segment with a Triangle

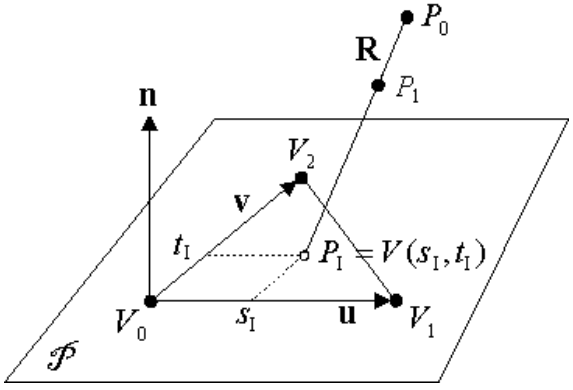
Consider a ray **R** (or a segment **S**) from P_0 to P_1, and a triangle **T** with vertices V_0, V_1 and V_2. The triangle **T**lies in the plane $\mathcal{P}$ through V_0 with normal vector $\mathbf{n} = (V_1 - V_0) \times (V_2 - V_0)$. To get the intersection of **R** (or **S**) with **T**, one first determines the intersection of **R** (or **S**) and $\mathcal{P}$. If it does not intersect, then it also does not intersect **T** and we are done. However, if they intersect in the point $P_I = P(r_I)$, we need to determine if this point is inside the triangle **T** for there to be a valid intersection.

There are a number of ways to test for the inclusion of a point inside a 3D planar triangle. The algorithms given by [Badouel, 1990] and [O'Rourke, 1998] project the point and triangle onto a 2D coordinate plane where inclusion is tested. To implement these algorithms, one must choose a projection coordinate plane which avoids a degenerate projection. This is done by excluding the coordinate which has the largest component in the plane normal vector **n** [Synder & Barr, 1987]. The intent is to reduce the 3D problem to a simpler 2D problem which has an efficient solution. However, there is a small overhead involved in selecting and applying the projection function. The algorithm of [Moller-Trumbore, 1997] (MT) does not project into 2D, and finds a solution using direct 3D computations. Testing with some complex models shows that the MT algorithm is faster than the one by Badouel.

We present here an alternate method that also uses direct 3D computations to determine inclusion, avoiding the projection onto a 2D coordinate plane. As a result, the code is cleaner and more compact. Like [Moller-Trumbore, 1997], we use the parametric equation of $\mathcal{P}$ relative to **T**, but derive a different method of solution which computes the ***parametric coordinates*** of the intersection point in the plane. The parametric plane equation is given by:

$$V(s,t) = V_0 + s(V_1 - V_0) + t(V_2 - V_0) = V_0 + s\mathbf{u} + t\mathbf{v}$$

where s and t are real numbers, and $\mathbf{u} = V_1 - V_0$ and $\mathbf{v} = V_2 - V_0$ are edge vectors of **T**. Then, $P = V(s,t)$ is in the triangle **T** when $s \geq 0$, $t \geq 0$, and $s + t \leq 1$. So, given P_I, one just has to find the (s_I, t_I) coordinate for it, and then check these inequalities to verify inclusion in **T**. Further, $P = V(s,t)$ is on an edge of **T** if one of the conditions $s = 0$, $t = 0$, or $s + t = 1$ is true (each condition corresponds to one edge). And, the three vertices are given by: $V_0 = V(0,0)$, $V_1 = V(1,0)$, and $V_2 = V(0,1)$.



To solve for s_I and t_I, we apply the method previously described for Barycentric Coordinate Computation using the "***generalized perp operator*** on $\mathcal{P}$" that we defined. Then, with $\mathbf{w} = P_I - V_0$, which is a vector in $\mathcal{P}$ (that is, $\mathbf{n} \bullet \mathbf{w} = 0$), we solve the equation: $\mathbf{w} = s\mathbf{u} + t\mathbf{v}$ for s and t. The final result is:

$$s_I = \frac{(\mathbf{u}\cdot\mathbf{v})(\mathbf{w}\cdot\mathbf{v}) - (\mathbf{v}\cdot\mathbf{v})(\mathbf{w}\cdot\mathbf{u})}{(\mathbf{u}\cdot\mathbf{v})^2 - (\mathbf{u}\cdot\mathbf{u})(\mathbf{v}\cdot\mathbf{v})}$$

$$t_I = \frac{(\mathbf{u}\cdot\mathbf{v})(\mathbf{w}\cdot\mathbf{u}) - (\mathbf{u}\cdot\mathbf{u})(\mathbf{w}\cdot\mathbf{v})}{(\mathbf{u}\cdot\mathbf{v})^2 - (\mathbf{u}\cdot\mathbf{u})(\mathbf{v}\cdot\mathbf{v})}$$

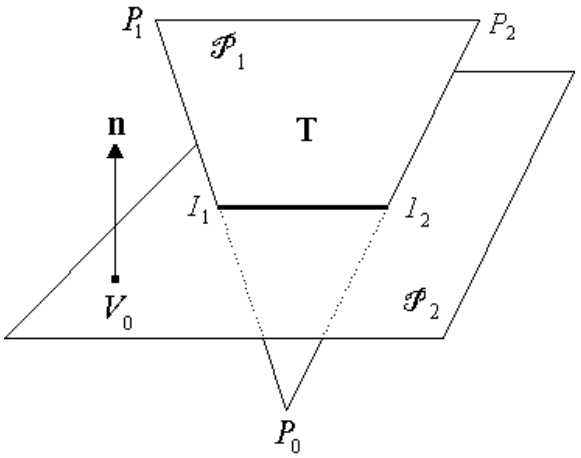
which has only 5 distinct dot products. We have arranged terms so that the two denominators are the same and only need to be calculated once.

This solution yields a straightforward ray/segment-triangle intersection algorithm (see our implementation: intersect3D_RayTriangle()). Based on a count of the operations done up to the first rejection test, this algorithm is a bit less efficient than the MT algorithm, although we have not done any runtime performance comparisons. However, the MT algorithm uses two cross products whereas our algorithm uses only one, and the one we use computes the normal vector of the triangle's plane, which is needed to compute the line parameter r_I. But, when the normal vectors have been precomputed and stored for all triangles in a scene (which is often the case), our algorithm would not have to compute this cross product at all. However, in this case, the MT algorithm would still compute two cross products, and be less efficient. So, the preferred algorithm depends on the application.

Intersection of a Triangle with a Plane

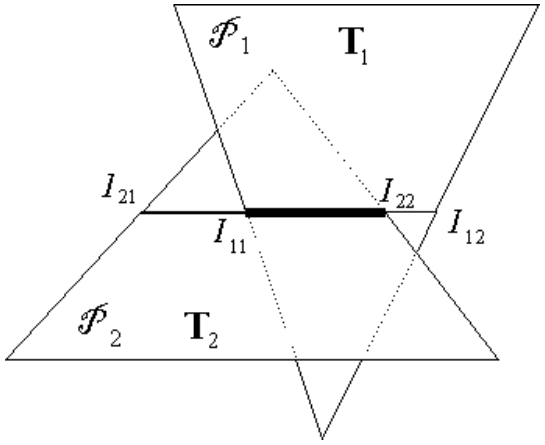
Consider a triangle **T** with vertices P_0, P_1 and P_2 lying in a plane $\mathcal{P}_1$ with normal $\mathbf{n}_1$. Let $\mathcal{P}_2$ be a second plane through the point V_0 with the normal vector $\mathbf{n}_2$. Unless they are parallel, the two planes $\mathcal{P}_1$ and $\mathcal{P}_2$intersect in a line **L**, and when **T** intersects $\mathcal{P}_2$ it will be a segment contained in **L**. When **T** does not intersect $\mathcal{P}_2$ all three of its vertices must strictgly lie on the same side of the $\mathcal{P}_2$ plane. On the other hand, when **T** does intersect $\mathcal{P}_2$, one point of **T** must be on one side of (or on) $\mathcal{P}_2$ and the other two be on the other side of (or on) $\mathcal{P}_2$. We previously gave a test for which side of a plane a point is on by using the signed distance from the point to the plane. In fact, to determine the sideness of a point P, one only has to find the sign of $\mathbf{n}\bullet(P - V_0)$.

Suppose that P_0 is on one side of $\mathcal{P}_2$ and that P_1 and P_2 are on the other side. Then the two segments $P_0P_1$and P_0P_2 intersect $\mathcal{P}_2$ in two points I_1 and I_2 which are on the intersection line of $\mathcal{P}_1$ and $\mathcal{P}_2$. Then, the segment I_1I_2 is the intersection of triangle **T** and the plane $\mathcal{P}_2$.



Intersection of a Triangle with a Triangle

Consider two triangles $\mathbf{T}_1$ and $\mathbf{T}_2$. They each lie in a plane, respectively $\mathscr{P}_1$ and $\mathscr{P}_2$, and their intersection must be on the line of intersection **L** for the two planes. Let the intersection of $\mathbf{T}_1$ and $\mathscr{P}_2$ be the segment $\mathbf{S}_1 = I_{11}I_{12}$, and the intersection of $\mathbf{T}_2$ and $\mathscr{P}_1$ be $\mathbf{S}_2 = I_{21}I_{22}$. If either $\mathbf{S}_1$ or $\mathbf{S}_2$ doesn't exist (that is, one triangle does not intersect the plane of the other), then $\mathbf{T}_1$ and $\mathbf{T}_2$ do not intersect. Otherwise their intersection is equal to the intersection of the two segments $\mathbf{S}_1$ and $\mathbf{S}_2$ on the line **L**. This can be easily computed by projecting them onto an appropriate coordinate axis, and determining their intersection on it.



Implementations

Here are some sample "C++" implementations of these algorithms.

```
// Copyright 2001, 2012, 2021 Dan Sunday
// This code may be freely used and modified for any purpose
// providing that this copyright notice is included with it.
// There is no warranty for this code, and the author of it cannot
// be held liable for any real or imagined damage 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;}
//        operators for:
//            == to test  equality
//            != to test  inequality
//            (Vector)0 =  (0,0,0)         (null vector)
//            Point  = Point ± Vector      (translation)
//            Vector = Point - Point
//            Vector = Scalar * Vector     (scalar product)
//            Vector = Vector * Vector     (cross product)
//    Line and Ray and Segment with defining  points {Point P0, P1;}
//        A Line is infinite, Rays and  Segments start at P0.
//        A Ray extends beyond P1, but a  Segment ends at P1.
//    Plane with a point and a normal {Point V0; Vector n;}
//    Triangle with defining vertices {Point V0, V1, V2;}
//    Polyline and Polygon with n vertices {int n;  Point *V;}
//        A Polygon has V[n]=V[0].
//===================================================================


#define SMALL_NUM   0.00000001 // to avoid division overflow
// 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)


// intersect3D_RayTriangle(): get a ray and triangle intersection
//    Input:  a ray R, and a triangle T
//    Output: *I = intersection point (when it exists)
//    Return: -1 = triangle is degenerate (a segment or point)
//             0 =  disjoint (no intersect)
//             1 =  intersect in unique point I1
//             2 =  are in the same plane
int
intersect3D_RayTriangle( Ray R, Triangle T, Point* I )
{
    Vector    u, v, n;          // triangle vectors
    Vector    dir, w0, w;       // ray vectors
    float     r, a, b;          // params to calc intersect
```

```
    // get triangle edge vectors and plane normal
    u = T.V1 - T.V0;
    v = T.V2 - T.V0;
    n = u * v;                  // cross product
    if (n == (Vector)0)         // triangle is degenerate
        return -1;              // do not deal with this case

    dir = R.P1 - R.P0;          // ray direction vector
    w0 = R.P0 - T.V0;
    a = -dot(n,w0);
    b = dot(n,dir);
    if (fabs(b) < SMALL_NUM) {  // ray is parallel to plane
        if (a == 0)             // ray lies in triangle plane
            return 2;
        else return 0;          // ray is disjoint from plane
    }

    // get intersect point of ray with triangle plane
    r = a / b;
    if (r < 0.0)                // ray goes away from triangle
        return 0;               // => no intersect
    // for a segment, also test if (r > 1.0) => no intersect

    *I = R.P0 + r * dir;        // intersect point of ray and plane

    // is I inside T?
    float    uu, uv, vv, wu, wv, D;
    uu = dot(u,u);
    uv = dot(u,v);
    vv = dot(v,v);
    w = *I - T.V0;
    wu = dot(w,u);
    wv = dot(w,v);
    D = uv * uv - uu * vv;

    // get and test parametric coords
    float s, t;
    s = (uv * wv - vv * wu) / D;
    if (s < 0.0 || s > 1.0)         // I is outside T
        return 0;
    t = (uv * wu - uu * wv) / D;
    if (t < 0.0 || (s + t) > 1.0)   // I is outside T
        return 0;

    return 1;                       // I is in T
}
```

References

Didier Badouel, "An Efficient Ray-Polygon Intersection" in Graphics Gems (1990)

Francis Hill, "The Pleasures of 'Perp Dot' Products" in Graphics Gems IV (1994)

Tomas Moller & Eric Haines, "Intersection Test Methods" in Real-Time Rendering (3rd Edition) (2008)

Tomas Moller & Ben Trumbore, "Fast Minimum Storage Ray-Triangle Intersection" in J. Graphics Tools (**jgt**) 2(1), 21-28 (1997)

Joseph O'Rourke, "Segment-Triangle Intersection" in Computational Geometry in C (2nd Edition) (1998)

J.P. Snyder and A.H. Barr, "Ray Tracing Complex Models Containing Surface Tessellations", ACM Comp Graphics 21, (1987)

Distance between Lines

We previously described methods to find the distance from a point to a line, to a line segment, and to a plane. We now consider the distance between both infinite lines and finite line segments.

One sometimes has to compute the minimum distance separating two geometric objects; for example, in collision avoidance algorithms. These objects could be polygons (in 2D) or polyhedra (in 3D). The Euclidean distance between any two geometric objects is defined as the minimum distance between any two of their points. That is, for two geometric sets $\mathbf{G}_1$ and $\mathbf{G}_2$ (in any n-dimensional space), the distance between them is defined as:

$$d(\mathbf{G}_1, \mathbf{G}_2) = \min_{P \in \mathbf{G}_1,\, Q \in \mathbf{G}_2} d(P, Q)$$

In this algorithm, we show how to efficiently compute this distance between lines, rays and segments, in any dimension. We do this by showing how to find two points, P_C in $\mathbf{G}_1$ and Q_C in $\mathbf{G}_2$, where this minimum occurs; that is, where $d(\mathbf{G}_1,\mathbf{G}_2) = d(P_C, Q_C)$. It is not true that these points always exist for arbitrary geometric sets. But, they exist for many well-behaved geometric objects such as lines, planes, polygons, polyhedra, and "compact" (i.e.: closed & bounded) objects.

Additionally, we show how to compute the closest point of approach (CPA) between two points that are dynamically moving in straight lines. This is an important computation in collision detection, for example to determine how close two planes or two ships, represented as point "tracks", will get as they pass each other.

We use the parametric equation to represent lines, rays and segments, as described in the chapter on Lines.

Distance between Lines

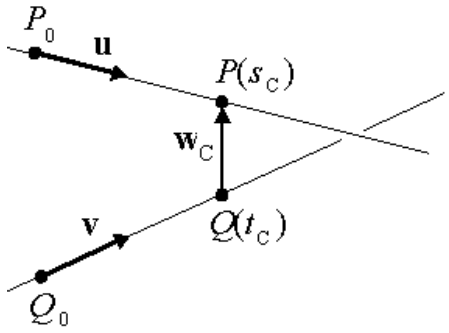
Consider two lines

$$\mathbf{L}_1\colon\ P(s) = P_0 + s(P_1 - P_0) = P_0 + s\mathbf{u}$$
$$\mathbf{L}_2\colon\ Q(t) = Q_0 + t(Q_1 - Q_0) = Q_0 + t\mathbf{v}.$$

Let $\mathbf{w}(s,t) = P(s) - Q(t)$ be a vector between points on the two lines. We want to find the $\mathbf{w}(s,t)$ that has a minimum length for all s and t. This can be computed using calculus [Eberly, 2001]. Here, we use a more geometric approach, and end up with the same result. A similar geometric approach was used by [Teller, 2000], but he used a

cross product which restricts his method to 3D space whereas our method works in any dimension. Also, the solution given here and the Eberly result are faster than Teller's which computes intermediate planes and gets their intersections with the two lines.

In any n-dimensional space, the two lines $\mathbf{L}_1$ and $\mathbf{L}_2$ are closest at unique points $P_C = P(s_C)$ and $Q_C = Q(t_C)$ for which $\mathbf{w}(s_C, t_C)$ is the unique minimum for $\mathbf{w}(s,t)$. Further, if $\mathbf{L}_1$ and $\mathbf{L}_2$ are not parallel and do not intersect each other, then the segment P_CQ_C joining these points is uniquely simultaneously perpendicular to both lines. No other segment between $\mathbf{L}_1$ and $\mathbf{L}_2$ has this property. That is, the vector $\mathbf{w}_C = \mathbf{w}(s_C, t_C)$ is uniquely perpendicular to the line direction vectors $\mathbf{u}$ and $\mathbf{v}$, and this is equivalent to it satisfying the two equations: $\mathbf{u} \bullet \mathbf{w}_C = 0$ and $\mathbf{v} \bullet \mathbf{w}_C = 0$.



We can solve these two equations by substituting $\mathbf{w}_C = P(s_C) - Q(t_C) = \mathbf{w}_0 + s_C\,\mathbf{u} - t_C\,\mathbf{v}$, where $\mathbf{w}_0 = P_0 - Q_0$, into each one to get two simultaneous linear equations:

$$(\mathbf{u}\cdot\mathbf{u})s_C - (\mathbf{u}\cdot\mathbf{v})t_C = -\mathbf{u}\cdot\mathbf{w}_0$$
$$(\mathbf{v}\cdot\mathbf{u})s_C - (\mathbf{v}\cdot\mathbf{v})t_C = -\mathbf{v}\cdot\mathbf{w}_0$$

Then, putting $a = \mathbf{u} \cdot \mathbf{u}$, $b = \mathbf{u} \cdot \mathbf{v}$, $c = \mathbf{v} \cdot \mathbf{v}$, $d = \mathbf{u} \cdot \mathbf{w}_0$, and $e = \mathbf{v} \cdot \mathbf{w}_0$, we solve for s_C and t_C as:

$$s_C = \frac{be - cd}{ac - b^2} \quad \text{and} \quad t_C = \frac{ae - bd}{ac - b^2}$$

whenever the denominator $ac–b^2$ is nonzero. Note that $ac - b^2 = |\mathbf{u}|^2 |\mathbf{v}|^2 - (|\mathbf{u}|\ |\mathbf{v}| \cos\theta)^2 = (|\mathbf{u}|\ |\mathbf{v}|\ \sin\theta)^2 \geq 0$ is always nonnegative. When $ac–b^2 = 0$, the two equations are dependant, the two lines are parallel, and the distance between the lines is constant. We can solve for this parallel distance of separation by fixing the value of one parameter and using either equation to solve for the other. Selecting $s_C = 0$, we get $t_C = d / b = e / c$.

Having solved for s_C and t_C, we have the points P_C and Q_C on the two lines $\mathbf{L}_1$ and $\mathbf{L}_2$ where they are closest to each other. Then the distance between them is given by:

$$\mathbf{d}(\mathbf{L}_1,\mathbf{L}_2) = \left|P(s_C) - Q(t_C)\right| = \left|(P_0 - Q_0) + \frac{(be-cd)\mathbf{u} - (ae-bd)\mathbf{v}}{ac-b^2}\right|$$

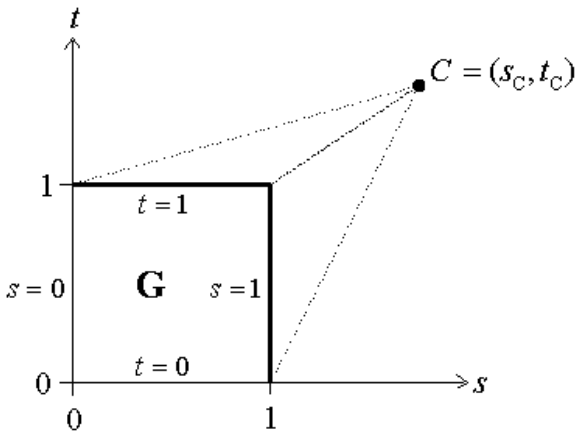
Distance between Segments and Rays

The distance between segments and rays may not be the same as the distance between their extended lines. The closest points on the extended infinite line may be outside the range of the segment or ray which is a restricted subset of the line. We represent the segment $\mathbf{S}_1 = [P_0, P_1]$ by $P(s) = P_0 + s(P_1 - P_0) = P_0 + s\mathbf{u}$ with $0 \le s \le 1$. And the positive ray $\mathbf{R}_1$ (starting from P_0) is given by the points $P(s)$ with $s \ge 0$. Similarly, the segment $\mathbf{S}_2 = [Q_0, Q_1]$ is given by the points $Q(t)$ with $0 \le t \le 1$, and the ray $\mathbf{R}_2$ is given by points with $t \ge 0$.

The first step in computing a distance involving segments and/or rays is to get the closest points for the infinite lines that they lie on. So, we first compute s_C and t_C for $\mathbf{L}_1$ and $\mathbf{L}_2$, and if these are both in the range of the respective segment or ray, then they are also give closest points. But if they lie outside the range of either, then they are not and we have to determine new points that minimize $\mathbf{w}(s,t) = P(s) - Q(t)$ over the ranges of interest. To do this, we first note that minimizing the length of $\mathbf{w}$ is the same as minimizing $|\mathbf{w}|^2 = \mathbf{w} \bullet \mathbf{w} = (\mathbf{w}_0 + s\mathbf{u} - t\mathbf{v}) \bullet (\mathbf{w}_0 + s\mathbf{u} - t\mathbf{v})$ which is a quadratic function of s and t. In fact, $|\mathbf{w}|^2$ defines a parabaloid over the (s,t)-plane with a minimum at $C = (s_C, t_C)$, and which is strictly increasing along rays in the (s,t)-plane that start from C and go in any direction. But, when segments and/or rays are involved, we need the minimum over a subregion $\mathbf{G}$ of the (s,t)-plane, and the global absolute minimum at C may lie outside of $\mathbf{G}$. However, in these cases, the minimum always occurs on the boundary of $\mathbf{G}$, and in particular, on the part of $\mathbf{G}$'s boundary that is visible to C. That is, there is a line from C to the boundary point which is exterior to $\mathbf{G}$, and we say that C can "see" points on this visible boundary of $\mathbf{G}$.

To be more specific, suppose that we want the minimum distance between two finite segments $\mathbf{S}_1$ and $\mathbf{S}_2$. Then, we have that $\mathbf{G} = \{(s,t) | 0 \le s \le 1 \text{ and } 0 \le t \le 1\} = [0,1] \times [0,1]$ is the unit square as shown in the diagram. The four edges of the square are given by $s = 0$, $s = 1$, $t = 0$, and $t = 1$. And, if $C = (s_C, t_C)$ is outside $\mathbf{G}$, then it can see at most two edges of $\mathbf{G}$. If $s_C < 0$, C can see the s = 0 edge; if $s_C > 1$, C can see the s = 1 edge; and similarly for t_C. Clearly, if C is not in $\mathbf{G}$, then at least 1 and at most 2 of these

inequalities are true, and they determine which edges of **G** are candidates for a minimum of $|\mathbf{w}|^2$.



For each candidate edge, we use basic calculus to compute where the minimum occurs on that edge, either in its interior or at an endpoint. Consider the edge s = 0, along which $|\mathbf{w}|^2 = (\mathbf{w}_0 - t\,\mathbf{v}) \bullet (\mathbf{w}_0 - t\,\mathbf{v})$. Taking the derivative with t we get a minimum when:

$$0 = \frac{d}{dt}|\mathbf{w}|^2 = -2\mathbf{v} \bullet (\mathbf{w}_0 - t\,\mathbf{v})$$

which gives a minimum on the edge at $(0, t_0)$ where:

$$t_0 = \frac{\mathbf{v} \bullet \mathbf{w}_0}{\mathbf{v} \bullet \mathbf{v}}$$

If $0 \le t_0 \le 1$, then this will be the minimum of $|\mathbf{w}|^2$ on **G**, and $P(0)$ and $Q(t_0)$ are the two closest points of the two segments. However, if t_0 is outside the edge, then an endpoint of the edge, (0,0) or (0,1), is the minimum along that edge; and further, we will have to check a second visible edge in case the true absolute minimum is on it. The other edges are treated in a similar manner.

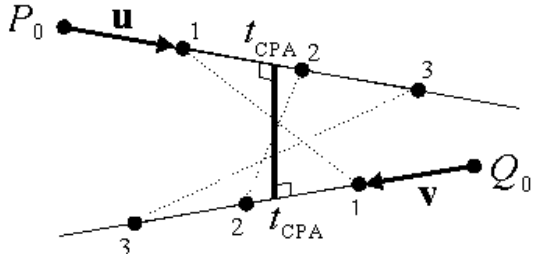
Putting it all together by testing all candidate edges, we end up with a relatively simple algorithm that only has a few cases to check. An analogous approach is given by [Eberly, 2001], but it has more cases which makes it more complicated to implement.

Other distance algorithms, such as line-to-ray or ray-to-segment, are similar in spirit, but have fewer boundary tests to make than the segment-to-segment distance algorithm. See our **`dist3D_Segment_to_Segment()`**.

Closest Point of Approach (CPA)

The "Closest Point of Approach" refers to the positions at which two dynamically moving objects reach their closest possible distance. This is an important calculation for collision avoidance. In many cases of interest, the objects, referred to as "tracks", are points moving in two fixed directions at fixed speeds. That means that the two points are moving along two lines in space. However, their closest distance is not the same as the closest distance between the lines since the distance between the points must be computed at the same moment in time. So, even in 2D with two lines that intersect, points moving along these lines may remain far apart. But if one of the tracks is stationary, then the CPA of another moving track is at the base of the perpendicular from the first track to the second's line of motion.

Consider two dynamically changing points whose positions at time t are $P(t)$ and $Q(t)$. Let their positions at time $t = 0$ be P_0 and Q_0; and let their velocity vectors per unit of time be **u** and **v**. Then, the equations of motion for these two points are $P(t) = P_0 + t\mathbf{u}$ and $Q(t) = Q_0 + t\mathbf{v}$, which are the familiar parametric equations for the lines. However, the two equations are coupled by having a common parameter t. So, at time t, the distance between them is $d(t) = |P(t) - Q(t)| = |\mathbf{w}(t)|$ where $\mathbf{w}(t) = \mathbf{w}_0 + t(\mathbf{u} - \mathbf{v})$ with $\mathbf{w}_0 = P_0 - Q_0$.



Now, since $d(t)$ is a minimum when $D(t) = d(t)^2$ is a minimum, we can compute:

$$D(t) = \mathbf{w}(t) \bullet \mathbf{w}(t) = (\mathbf{u} - \mathbf{v}) \bullet (\mathbf{u} - \mathbf{v})t^2 + 2\mathbf{w}_0 \bullet (\mathbf{u} - \mathbf{v})t + \mathbf{w}_0 \bullet \mathbf{w}_0$$

which has a minimum when

$$0 = \frac{d}{dt} D(t) = 2t[(\mathbf{u} - \mathbf{v}) \bullet (\mathbf{u} - \mathbf{v})] + 2\mathbf{w}_0 \bullet (\mathbf{u} - \mathbf{v})$$

which can be solved to get the time of CPA to be:

$$t_{CPA} = \frac{-\mathbf{w}_0 \bullet (\mathbf{u} - \mathbf{v})}{|\mathbf{u} - \mathbf{v}|^2}$$

whenever $|\mathbf{u} - \mathbf{v}|$ is nonzero. If $|\mathbf{u} - \mathbf{v}| = 0$, then the two point tracks are traveling in the same direction at the same speed, and will always remain the same distance apart, so one can use $t_{CPA} = 0$. In both cases we have that:

$$\mathbf{d}_{\mathrm{CPA}}\left(P(t),Q(t)\right) = \left|P(t_{\mathrm{CPA}}) - Q(t_{\mathrm{CPA}})\right|$$

Note that when $t_{CPA} < 0$, then the CPA has already occurred in the past, and the two tracks are getting further apart as they move on in time.

Implementations

Here are some sample "C++" implementations of these algorithms.

```
// Copyright 2001, 2012, 2021 Dan Sunday
// This code may be freely used and modified for any purpose
// providing that this copyright notice is included with it.
// There is no warranty for this code, and the author of it cannot
// be held liable for any real or imagined damage 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;}
//        operators for:
//            Point   = Point ± Vector
//            Vector =  Point - Point
//            Vector =  Vector ± Vector
//            Vector =  Scalar * Vector
//    Line and Segment with defining points {Point  P0, P1;}
//    Track with initial position and velocity vector
//            {Point P0;  Vector v;}
//===================================================================


#define SMALL_NUM   0.00000001 // to avoid division overflow
// 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
#define abs(x)     ((x) >= 0 ? (x) : -(x))   //  absolute value


// dist3D_Line_to_Line(): get the minimum distance between 2 lines
//    Input:  two 3D lines L1 and L2
//    Return: the shortest distance between L1 and L2
float
dist3D_Line_to_Line( Line L1, Line L2)
```

```
{
    Vector   u = L1.P1 - L1.P0;
    Vector   v = L2.P1 - L2.P0;
    Vector   w = L1.P0 - L2.P0;
    float    a = dot(u,u);        // always >= 0
    float    b = dot(u,v);
    float    c = dot(v,v);        // always >= 0
    float    d = dot(u,w);
    float    e = dot(v,w);
    float    D = a*c - b*b;       // always >= 0
    float    sc, tc;

    // compute the line parameters of the two closest points
    if (D < SMALL_NUM) {          // the lines are almost parallel
        sc = 0.0;
        tc = (b>c ? d/b : e/c);   // use the largest denominator
    }
    else {
        sc = (b*e - c*d) / D;
        tc = (a*e - b*d) / D;
    }

    // get the difference of the two closest points
    Vector   dP = w + (sc * u) - (tc * v);  // =  L1(sc) - L2(tc)

    return norm(dP);   // return the closest distance
}
//===================================================================


// dist3D_Segment_to_Segment(): get the distance between 2 segments
//    Input:  two 3D line segments S1 and S2
//    Return: the shortest distance between S1 and S2
float
dist3D_Segment_to_Segment( Segment S1, Segment S2)
{
    Vector   u = S1.P1 - S1.P0;
    Vector   v = S2.P1 - S2.P0;
    Vector   w = S1.P0 - S2.P0;
    float    a = dot(u,u);         // always >= 0
    float    b = dot(u,v);
    float    c = dot(v,v);         // always >= 0
    float    d = dot(u,w);
    float    e = dot(v,w);
    float    D = a*c - b*b;        // always >= 0
    float    sc, sN, sD = D;       // sc = sN/sD, default sD=D >= 0
    float    tc, tN, tD = D;       // tc = tN/tD, default tD=D >= 0

    // compute the line parameters of the two closest points
    if (D < SMALL_NUM) { // the lines are almost parallel
        sN = 0.0;         // force using point P0 on segment S1
        sD = 1.0;         // to prevent possible division by 0 later
        tN = e;
```

```
        tD = c;
    }
    else {          // get the closest points on the infinite lines
        sN = (b*e - c*d);
        tN = (a*e - b*d);
        if (sN < 0.0) {       // sc < 0 => the s=0 edge is visible
            sN = 0.0;
            tN = e;
            tD = c;
        }
        else if (sN > sD) {   // sc > 1  => the s=1 edge is visible
            sN = sD;
            tN = e + b;
            tD = c;
        }
    }

    if (tN < 0.0) {           // tc < 0 => the t=0 edge is visible
        tN = 0.0;
        // recompute sc for this edge
        if (-d < 0.0)
            sN = 0.0;
        else if (-d > a)
            sN = sD;
        else {
            sN = -d;
            sD = a;
        }
    }
    else if (tN > tD) {      // tc > 1  => the t=1 edge is visible
        tN = tD;
        // recompute sc for this edge
        if ((-d + b) < 0.0)
            sN = 0;
        else if ((-d + b) > a)
            sN = sD;
        else {
            sN = (-d +  b);
            sD = a;
        }
    }
    // finally do the division to get sc and tc
    sc = (abs(sN) < SMALL_NUM ? 0.0 : sN / sD);
    tc = (abs(tN) < SMALL_NUM ? 0.0 : tN / tD);

    // get the difference of the two closest points
    Vector   dP = w + (sc * u) - (tc * v);  // =  S1(sc) - S2(tc)

    return norm(dP);   // return the closest distance
}
//===================================================================
```

```
// cpa_time(): compute the time of CPA for two tracks
//    Input:  two tracks Tr1 and Tr2
//    Return: the time at which the two tracks are closest
float
cpa_time( Track Tr1, Track Tr2 )
{
    Vector   dv = Tr1.v - Tr2.v;

    float    dv2 = dot(dv,dv);
    if (dv2 < SMALL_NUM)       // the  tracks are almost parallel
        return 0.0;            // any time is ok.  Use time 0.

    Vector   w0 = Tr1.P0 - Tr2.P0;
    float    cpatime = -dot(w0,dv) / dv2;

    return cpatime;            // time of CPA
}
//===================================================================


// cpa_distance(): compute the distance at CPA for two tracks
//    Input:  two tracks Tr1 and Tr2
//    Return: the distance for which the two tracks are closest
float
cpa_distance( Track Tr1, Track Tr2 )
{
    float    ctime = cpa_time( Tr1, Tr2);
    Point    P1 = Tr1.P0 + (ctime * Tr1.v);
    Point    P2 = Tr2.P0 + (ctime * Tr2.v);

    return d(P1,P2);             // distance at CPA
}
//===================================================================
```

References

David Eberly, "Distance Methods" in 3D Game Engine Design (2006)

Seth Teller, line_line_closest_points3d() (2000) cited in the Graphics Algorithms FAQ (2001)

Intersections for a Set of Segments

Sometimes an application needs to find the set of intersection points for a collection of many line segments. Often these applications involve polygons which are just an ordered set of connected segments. Specific problems that might need an algorithmic solution are:

1. Compute the intersection (or union, or difference) of two simple polygons or planar graphs. To do this, one must determine all intersection points, and use them as new vertices to construct the intersection (or union, or difference).

2. Test if two polygons or planar graphs intersect. One has to determine the intersections of one object's edges with those of the other. As soon as any valid intersection is found, the test can stop, and it doesn't have to determine the complete set of intersections.

3. Test if a polyline or polygon is simple. That is, determine if any two nonsequential edges of a polyline intersect. This is an important property since many algorithms only work for simple polylines or polygons. Again, his test can stop as soon as any intersection is found.

4. Decompose a polygon into simple pieces. To do this, one needs to know the complete set of intersection points between the edges, and use each of them as a cut-point in the decomposition.

Algorithms solving these problems are used in many application areas such as computer graphics, CAD, circuit design, hidden line elimination, computer vision, and so on.

A Short Survey of Intersection Algorithms

In general, for a set of n line segments, there can be up to $O(n^2)$ intersection points, since if every segment intersected every other segment, there would be $n(n-1)/2 = O(n^2)$ intersection points. In the worst case, to compute them all would require a $O(n^2)$ algorithm. The "brute force" algorithm would simply consider all $O(n^2)$ pairs of line segments, test each pair for intersection, and record the ones it finds. This is a lot of computing. However, when there are only a few intersection points, or only one such point needs to be detected (or not), there are faster algorithms.

In fact, these problems can be solved by "output-sensitive" algorithms whose efficiency depends on both the input and the output sizes. Here the input is a set Ω of n segments, and the output is the set Λ of k computed intersections, where $k = n^2$ in the worst case, but is usually much smaller. An early algorithm [Shamos & Hoey, 1976] showed how to detect if at least one intersection exists in $O(n \log n)$ time and $O(n)$ space by "sweeping" over a linear ordering of Ω. Extending their idea, [Bentley & Ottmann, 1979] gave an algorithm to compute all k intersections in $O((n+k)\log n)$ time and $O(n+k)$ space. After many decades, the well-known "***Bentley-Ottmann Algorithm***" is still the most popular one to implement in practice ([Bartuschka, Mehlhorn & Naher, 1997], [de Berg et al, 2000], [Hobby, 1999], [O'Rourke, 1998], [Preparata & Shamos, 1985]) since it is relatively easy to both understand and

implement. However, their algorithm did not achieve the theoretical lower bound; and thus, was only the first of many output-sensitive algorithms for solving the segment intersection problem.

A decade later, [Chazelle & Edelsbrunner, 1988 and 1992] discovered an optimal $O(n \log n + k)$ time algorithm. But, their algorithm still needs $O(n+k)$ storage space, and it is difficult to implement. Subsequent work made further improvements, and [Balaban, 1995] found an $O(n \log n + k)$ time and $O(n)$ space deterministic algorithm. There have also been a number of "randomized" algorithms with *expected* $O(n \log n + k)$ running time. The earliest of these by [Myers, 1985] uses $O(n+k)$ space. However, the later one by [Clarkson & Shor, 1989] uses only $O(n)$ space.

Additionally, improved algorithms have been found for the more restrictive "red-blue intersection" problem. Here there are two separate sets of segments, the "red" set Ω_1 and the "blue" set Ω_2. One wants to find intersections between the sets, but not within the same set; that is, red-blue intersections, but not red-red or blue-blue ones. A simple deterministic $O(n \log n + k)$ time and $O(n)$ space "trapezoid sweep" algorithm was developed by [Chan, 1994] based on earlier work of [Mairson & Stolfi, 1988]. These algorithms can be used to perform boolean set operations, like intersections or unions, between two different simple polygons or planar subdivision graphs.

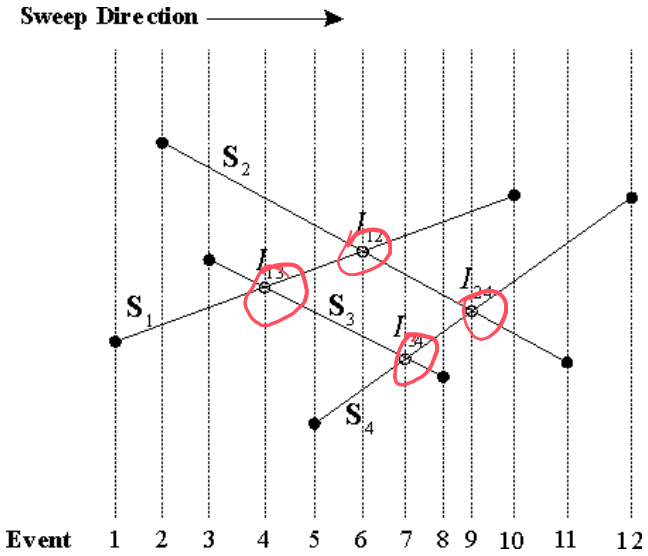
Nevertheless, the Shamos-Hoey and Bentley-Ottmann algorithms remain the landmarks of the field. Note, however, that when k is large of order $O(n^2)$, the Bentley-Ottmann algorithm takes $O(n^2 \log n)$ time which is worse than the $O(n^2)$ brute-force algorithm! Also, the more complicated optimal algorithms are $O(n^2)$ which is the same as the simple brute-force one. So, when k is expected to be much larger than $O(n)$, one might as well use the easy-to-implement brute-force algorithm. But, when k is expected to be less than or equal to $O(n)$, Bentley-Ottmann is the simplest expected $O(n \log n)$ time and $O(n)$ space algorithm.

The Bentley-Ottmann Algorithm

The input for the Bentley-Ottmann algorithm is a collection $\Omega = \{\mathbf{L}_i\}$ of line segments $\mathbf{L}_i$, and its output will be a set $\mathbf{A} = \{I_j\}$ of intersection points. This algorithm is referred to as a "***sweep line algorithm***" because it's operation can be visualized as having another line, a "sweep line" **SL**, sweeping over the collection Ω and collecting information as it passes over the individual segments $\mathbf{L}_i$. The information collected for each position of **SL** is basically an ordered list of all segments in Ω that are currently being intersected by **SL**. The data structure maintaining this information is often also called the "sweep line". This class structure also detects and outputs intersections as it discovers them. The process by which it discovers intersections is the heart of the algorithm and its efficiency.

To implement the sweep logic, we must first linearly order the segments of Ω to determine the sequence in which **SL** encounters them. That is, we need to order the endpoints $\{E_{i0}, E_{i1}\}_{i=1,n}$ of all the segments $\mathbf{L}_i$ so we can detect when **SL** starts and stops intersecting each segment of Ω. Traditionally, the endpoints are ordered by increasing x first and then increasing y-coordinate values, but any linear order will do (some authors prefer decreasing y first and then

increasing x). With the traditional ordering, the sweep line is vertical and moves from left to right as it encounters each segment, as shown in the diagram:



At any point in the algorithm, the sweep line **SL** intersects only those segments with one endpoint to the left of (or on) it and the other endpoint to the right of it. The **SL** data structure keeps a dynamic list of these segments by: (1) adding a segment when its leftmost endpoint is encountered, and (2) deleting a segment when its rightmost endpoint is encountered. Further, the **SL** orders the list of segments with an "above-below" relation. So, to add or delete a segment, its position in the list must be determined, which can be done by a worst-case $O(\log n)$ binary search of the current segments in the list. In addition, besides adding or deleting segments, there is another event that changes the list structure: namely, whenever two segments intersect, then their positions in the ordered list must be swapped. Given the two segments, which must be neighbors in the list, this swap is an $O(\log n)$ operation.

To organize all this, the algorithm maintains an ordered "***event queue***" **EQ** whose elements cause a change in the **SL** segment list. Initially, **EQ** is set to the sweep-ordered list of all segment endpoints. But as intersections between segments are found, then they are also added to **EQ** in the same sweep-order as used for the endpoints One must test, though, to avoid inserting duplicate intersections onto the event queue. The example in the above diagram shows how this can happen. At event 2, segments $\mathbf{S}_1$ and $\mathbf{S}_2$ cause intersection I_{12} to be computed and put on the queue. Then, at event 3, segment $\mathbf{S}_3$ comes between and separates $\mathbf{S}_1$ and $\mathbf{S}_2$. Next, at event 4, $\mathbf{S}_1$ and $\mathbf{S}_3$ swap places on the sweep line, and $\mathbf{S}_1$ is brought next to $\mathbf{S}_2$ again causing I_{12} to be computed again. But, there can only be one event for each intersection, and I_{12} cannot be put on the queue twice. So, when an intersection is being put on the queue, we must find its potential x-sorted location in the queue, and check that it is not already there. Since there is at most one intersect point for any two segments, labeling an intersection with identifiers for the segments is sufficient to uniquely identify it. As a result of all this, the maximum size of the event queue = $2n+k \le 2n+n^2$, and any insertion or deletion can be done with a $O(\log(2n+n^2)) = O(\log n)$ binary search.

But, what does all this have to do with efficiently finding the complete set of segment intersections? Well, as segments are sequentially added to the **SL** segment list, their possible intersections with other eligible segments are determined. When a valid intersection is found, then it is inserted into the event queue. Further, when an intersection-event on **EQ** is processed during the sweep, then it causes a re-ordering of the **SL** list, and the intersection is also added to the output list **A**. In the end, when all events have been processed, **A** will contain the complete ordered set of all intersections.

However, there is one critical detail, the heart of the algorithm, that we still need to describe; namely, how does one compute a valid intersection? Clearly, two segments can only intersect if they occur simultaneously on the sweep-line at some time. But this by itself is not enough to make the algorithm efficient. The important observation is that two intersecting segments must be immediate above-below neighbors on the sweep-line. Thus, there are only a few restricted cases for which possible intersections need to be computed:

1. When a segment is added to the **SL** list, determine if it intersects with its above and below neighbors.
2. When a segment is deleted from the **SL** list, its previous above and below neighbors are brought together as new neighbors. So, their possible intersection needs to be determined.
3. At an intersection event, two segments switch positions in the **SL** list, and their intersection with their new neighbors (one for each) must be determined.

This means that for the processing of any one event (endpoint or intersection) of **EQ**, there are at most two intersection determinations that need to be made.

One detail remains, namely the time needed to add, find, swap, and remove segments from the **SL** structure. To do this, the **SL** can be implemented as a balanced binary tree (such as an AVL, a 2-3, or a red-black tree) which guarantees that these operations will take at most $O(\log n)$ time since n is the maximum size of the **SL**list. Thus, each of the $(2n+k)$ events has at worst $O(\log n)$ processing to do. Adding up the initial sort and the event processing, the efficiency of the algorithm is: $O(n \log n)+O((2n+k)\log n)=O((n+k)\log n)$.

We do not provide the code for a balanced binary tree, which is needed for the sweepline data structure. We did this to concentrate only on the geometric aspect of the algorithm, and not recommend a specific type of balanced tree, such as AVL, or red-black, or 2-3. These can be found in standard libraries. Also, C++ code for an AVL balanced Tree has been developed by [Brad Appleton, 1997], is available online and can be downloaded from his website.

Pseudo-Code: Bentley-Ottmann Algorithm

Putting all of this together, the top-level logic for an implementation of the Bentley-Ottmann algorithm is given by the following pseudo-code:

```
Initialize event queue EQ = all segment endpoints;
Sort EQ by increasing x and y;
Initialize sweep line SL to be empty;
Initialize output intersection list IL to be empty;

While (EQ is nonempty) {
```

```
        Let E = the next event from EQ;
        If (E is a left endpoint) {
            Let segE = E's segment;
            Add segE to SL;
            Let segA = the segment Above segE in SL;
            Let segB = the segment Below segE in SL;
            If (I = Intersect( segE with segA) exists)
                Insert I into EQ;
            If (I = Intersect( segE with segB) exists)
                Insert I into EQ;
        }
        Else If (E is a right endpoint) {
            Let segE = E's segment;
            Let segA = the segment Above segE in SL;
            Let segB = the segment Below segE in SL;
            Delete segE from SL;
            If (I = Intersect( segA with segB) exists)
                If (I is not in EQ already)
                    Insert I into EQ;
        }
        Else {  // E is an intersection event
            Add E's intersect point to the output list IL;
            Let segE1 above segE2 be E's intersecting segments in SL;
            Swap their positions so that segE2 is now above segE1;
            Let segA = the segment above segE2 in SL;
            Let segB = the segment below segE1 in SL;
            If (I = Intersect(segE2 with segA) exists)
                If (I is not in EQ already)
                    Insert I into EQ;
            If (I = Intersect(segE1 with segB) exists)
                If (I is not in EQ already)
                    Insert I into EQ;
        }
        remove E from EQ;
    }
    return IL;
}
```

This routine outputs the complete ordered list of all intersection points.

The Shamos-Hoey Algorithm

If one only wants to know if an intersection exists, then as soon as any intersection is detected, the routine can terminate immediately. This results in a greatly simplified algorithm. Intersections don't ever have to be put on the event queue, and so its size is only $2n$ for the endpoints of all the segments. And, code for processing this non-existent event can be removed. Further, the event (priority) queue can be implemented as a simple ordered array since it never changes. Additionally, no output list needs to be built since the algorithm terminates as soon as any intersection is found. Consequently, this algorithm needs only $O(n)$ space and runs in $O(n \log n)$ time. This is the original algorithm of [Shamos & Hoey, 1976].

Pseudo-Code: Shamos-Hoey Algorithm

The simplified pseudo-code is:

```
        Initialize event queue EQ = all segment endpoints;
        Sort EQ by increasing x and y;
        Initialize sweep line SL to be empty;

        While (EQ is nonempty) {
            Let E = the next event from EQ;
            If (E is a left endpoint) {
                Let segE = E's segment;
                Add segE to SL;
                Let segA = the segment Above segE in SL;
                Let segB = the segment Below segE in SL;
                If (I = Intersect( segE with segA) exists)
                    return TRUE;   // an Intersect Exists
                If (I = Intersect( segE with segB) exists)
                    return TRUE;   // an Intersect Exists
            }
            Else {  // E is a right endpoint
                Let segE = E's segment;
                Let segA = the segment above segE in SL;
                Let segB = the segment below segE in SL;
                Delete segE from SL;
                If (I = Intersect( segA with segB) exists)
                    return TRUE;   // an Intersect Exists
            }
            remove E from EQ;
        }
        return FALSE;       // No  Intersections
    }
```

Applications

Simple Polygons

(A) **Test if Simple**. The Shamos-Hoey algorithm can be used to ***test if a polygon is simple or not***. We give a C++ implementation simple_Polygon() for this algorithm below. Note that the shared endpoint between sequential edges does not count as a non-simple intersection point, and the intersection test routine must check for that.

(B) **Decompose into Simple Pieces**. The Bentley-Ottmann algorithm can be used to ***decompose a non-simple polygon into simple pieces***. To do this, all intersection points are needed. One approach for a simple decomposition algorithm is to perform "surgery" at each intersection point. This proceedure can be incorporated into the sweepline algorithm as it discovers new intersection points, resulting in an $O(n \log n + k)$ decomposition algorithm. To do this, one must also output the edges that persist in a linked list, whose connected sublists will be the new simple polygons.

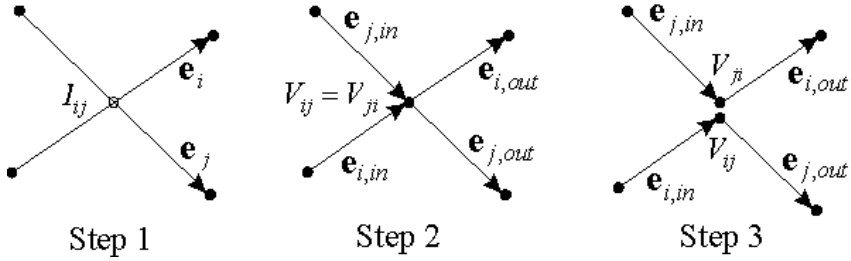
Let the original polygon be given by the set of segment endpoints $\Omega = \{E_i\}_{i=0,n}$, with $E_n = E_0$. Its directed edges are given by the vectors $\mathbf{e}_i = E_{i+1} - E_i$.

1. Compute all the intersection points of the edge segments using the Bentley-Ottmann algorithm. The following steps may be incorporated into this algorithm whenever the sweepline finds a new intersection.

2. For an intersection point I_{ij} between $\mathbf{e}_i$ and $\mathbf{e}_j$, ($j > i+1 > 0$), add 2 new vertices V_{ij} and V_{ji} (one on each edge $\mathbf{e}_i$ and $\mathbf{e}_j$). Split each edge $\mathbf{e}_k$ into two new edges $\mathbf{e}_{k,in}$ and $\mathbf{e}_{k,out}$ joined at the new vertex on $\mathbf{e}_k$.

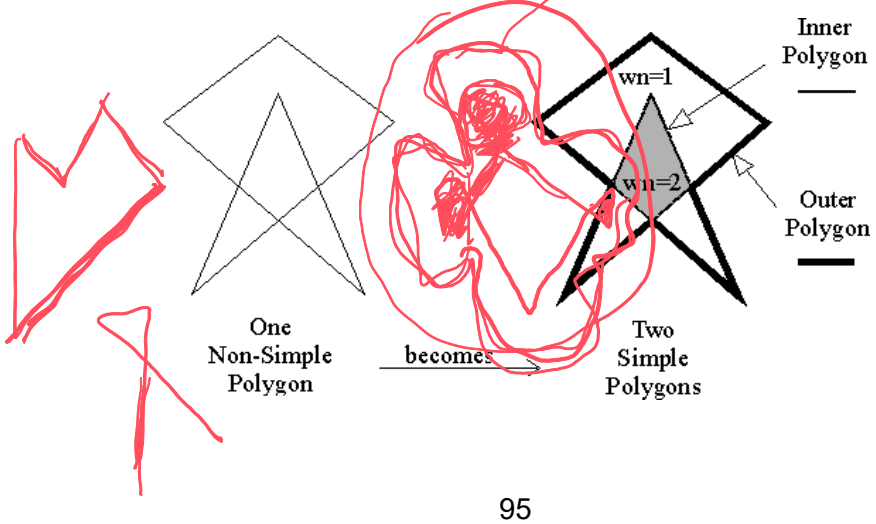
   Add these new edges to the sweepline list, and also record them as new polygon edges. Also, reassign any other intersections that $\mathbf{e}_i$ and $\mathbf{e}_j$ may have had (with other edges) to the new *in* & *out* edges. When implemented within the sweepline algorithm, this reassignment is made to the rightmost new edge, and the leftmost new edge gets deleted from the sweepline list.

3. Next, do surgery at these new vertices to remove the crossover. This is done by
   a) attaching $\mathbf{e}_{i,in}$ to $\mathbf{e}_{j,out}$ at V_{ij}, and
   b) attaching $\mathbf{e}_{j,in}$ to $\mathbf{e}_{i,out}$ at V_{ji},
   as shown in the following diagram.



4. After doing this at all intersections, then the remaining connected edge sets are the simple polygons decomposing the original non-simple one.

Note that the resulting simple polygons may not be disjoint since one could be contained inside another. In fact, the decomposition inclusion hierarchy is based on the inclusion winding number of each simple polygon in the original non-simple one. For example:



Polygon Set Operations

The Bentley-Ottmann algorithm can be used to speed up computing the intersection, union, or difference of two general non-convex simple polygons. Of course, before using any complicated algorithm to perform these operations, one should first test the bounding boxes or spheres of the polygons for overlap. If the bounding containers are disjoint, then so are the two polygons, and the set operations become trivial.

However, when two polygons overlap, the sweep line strategy of the Bentley-Ottmann algorithm can be adapted to perform a set operation on any two simple polygons. For further details see [O'Rourke, 1998, 266-269]. If the two polygons are known to be simple, then one just needs intersections for segments from different polygons, which is a red-blue intersection problem.

Planar Subdivisions

The Bentley-Ottmann algorithm can be used to efficiently compute the overlay of two planar subdivisions. For details, see [de Berg et al, 2000, 33-39]. A planar subdivision is a planar graph with straight line segments for edges, and it divides the plane into a finite number of regions. For example, boundary lines divide a country into states. When two such planar graphs are overlaid (or superimposed), then their combined graph defines a subdivision refinement of each one. To compute this refinement, one needs to calculate all intersections between the line segments in both graphs. For a segment in one graph, we only need the intersections with segments in the other graph, and so this is another red-blue intersection problem.

Implementations

Here are some sample "C++" implementations of these algorithms.

```
// Copyright 2001, 2012, 2021 Dan Sunday
// This code may be freely used and modified for any purpose
// providing that this copyright notice is included with it.
// There is no warranty for this code, and the author of it cannot
// be held liable for any real or imagined damage from its use.
// Users of this code must verify correctness for their application.


// Assume that classes are already given for the objects:
//    Point with 2D coordinates {float x, y;}
//    Polygon with n vertices {int n; Point *V;} with V[n]=V[0]
//    Tnode is a node element structure for a BBT
//    BBT is a class for a Balanced Binary Tree
//        such as an AVL, a 2-3, or a  red-black tree
//        with methods given by the  placeholder code:

typedef struct _BBTnode Tnode;
struct _BBTnode {
        void* val;
        // plus node mgmt info ...
};
```

```
class BBT {
        Tnode   *root;
public:
                 BBT() {root = (Tnode*)0;}   // constructor
                 ~BBT() {freetree();}        // destructor

        Tnode*  insert( void* ){};       // insert data into the tree
        Tnode*  find( void* ){};         // find data from the tree
        Tnode*  next( Tnode* ){};        // get next tree node
        Tnode*  prev( Tnode* ){};        // get previous tree node
        void    remove( Tnode*  ){};     // remove node from the tree
        void    freetree(){};            // free all tree data structs
};
// NOTE:
// Code for these methods must be provided for the algorithm to work.
// We have not provided it since binary tree algorithms are well-known
// and code is widely available such as [Brad Appleton, 1967]. Also,
// we want to reduce clutter in the essential sweep line algorithm.
//==================================================================


#define FALSE   0
#define TRUE    1
#define LEFT    0
#define RIGHT   1

extern void
qsort(void*, unsigned, unsigned, int(*)(const void*,const void*));

// xyorder(): determines the xy lexicographical order of two points
//      returns: (+1) if p1 > p2; (-1) if p1 < p2; and  0 if equal
int xyorder( Point* p1, Point* p2 )
{
    // test the x-coord first
    if (p1->x > p2->x) return 1;
    if (p1->x < p2->x) return (-1);
    // and test the y-coord second
    if (p1->y > p2->y) return 1;
    if (p1->y < p2->y) return (-1);
    // when you exclude all other possibilities, what remains  is...
    return 0;  // they are the same point
}

// isLeft(): tests if point P2 is Left|On|Right of the line P0 to P1.
//      returns: >0 for left, 0 for on, and <0 for  right of the line.
inline float
isLeft( Point P0, Point P1, Point P2 )
{
    return (P1.x - P0.x)*(P2.y - P0.y) - (P2.x - P0.x)*(P1.y -  P0.y);
}
//==================================================================



// EventQueue Class

// Event element data struct
typedef struct _event Event;
struct _event {
    int      edge;          // polygon edge i is V[i] to V[i+1]
    int      type;          // event type: LEFT or RIGHT vertex
```

```
    Point*   eV;            // event vertex
};

int E_compare( const void* v1, const void* v2 ) // qsort compare two events
{
    Event**    pe1 = (Event**)v1;
    Event**    pe2 = (Event**)v2;

    return xyorder( (*pe1)->eV, (*pe2)->eV );
}

// the EventQueue is a presorted array (no insertions needed)
class EventQueue {
    int      ne;               // total number of events in array
    int      ix;               // index of next event on queue
    Event*   Edata;            // array of all events
    Event**  Eq;               // sorted list of event pointers
public:
              EventQueue(Polygon P);     // constructor
             ~EventQueue(void)           // destructor
                  { delete Eq; delete Edata;}

    Event*   next();                    // next event on queue
};

// EventQueue Routines
EventQueue::EventQueue( Polygon P )
{
    ix = 0;
    ne = 2 * P.n;           // 2 vertex events for each edge
    Edata = (Event*)new Event[ne];
    Eq = (Event**)new (Event*)[ne];
    for (int i=0; i < ne; i++)          // init Eq array pointers
        Eq[i] = &Edata[i];

    // Initialize event queue with edge segment endpoints
    for (int i=0; i < P.n; i++) {       // init data for edge i
        Eq[2*i]->edge = i;
        Eq[2*i+1]->edge = i;
        Eq[2*i]->eV   = &(P.V[i]);
        Eq[2*i+1]->eV = &(P.V[i+1]);
        if (xyorder( &P.V[i], &P.V[i+1]) < 0)  { // determine type
            Eq[2*i]->type    = LEFT;
             Eq[2*i+1]->type = RIGHT;
        }
        else {
            Eq[2*i]->type    = RIGHT;
             Eq[2*i+1]->type = LEFT;
        }
    }
    // Sort Eq[] by increasing x and y
    qsort( Eq, ne, sizeof(Event*), E_compare );
}

Event* EventQueue::next()
{
    if (ix >= ne)
        return (Event*)0;
    else
        return Eq[ix++];
}
```

```
//===================================================================


// SweepLine Class

// SweepLine segment data struct
typedef struct _SL_segment SLseg;
struct _SL_segment {
    int      edge;          // polygon edge i is V[i] to V[i+1]
    Point    lP;            // leftmost vertex point
    Point    rP;            // rightmost vertex point
    SLseg*   above;         // segment above this one
    SLseg*   below;         // segment below this one
};

// the Sweep Line itself
class SweepLine {
    int      nv;            // number of vertices in polygon
    Polygon* Pn;            // initial Polygon
    BBT      Tree;          // balanced binary tree
public:
              SweepLine(Polygon P)            // constructor
                  { nv = P.n; Pn = &P; }
             ~SweepLine(void)                 // destructor
                  { Tree.freetree();}

    SLseg*   add( Event* );
    SLseg*   find( Event* );
    int      intersect( SLseg*, SLseg*  );
    void     remove( SLseg* );
};

SLseg* SweepLine::add( Event* E )
{
    // fill in SLseg element data
    SLseg* s = new SLseg;
    s->edge  = E->edge;

    // if it is being added, then it must be a LEFT edge event
    // but need to determine which endpoint is the left one
    Point* v1 = &(Pn->V[s->edge]);
    Point* v2 = &(Pn->V[s->edge+1]);
    if (xyorder( v1, v2) < 0) { // determine which is leftmost
        s->lP = *v1;
        s->rP = *v2;
    }
    else {
        s->rP = *v1;
        s->lP = *v2;
    }
    s->above = (SLseg*)0;
    s->below = (SLseg*)0;

    // add a node to the balanced binary tree
    Tnode* nd = Tree.insert(s);
    Tnode* nx = Tree.next(nd);
    Tnode* np = Tree.prev(nd);
    if (nx != (Tnode*)0) {
        s->above = (SLseg*)nx->val;
        s->above->below = s;
    }
```

```
    if (np != (Tnode*)0) {
        s->below = (SLseg*)np->val;
        s->below->above = s;
    }
    return s;
}

SLseg* SweepLine::find( Event* E )
{
    // need a segment to find it in the tree
    SLseg* s = new SLseg;
    s->edge  = E->edge;
    s->above = (SLseg*)0;
    s->below = (SLseg*)0;

    Tnode* nd = Tree.find(s);
    delete s;
    if (nd == (Tnode*)0)
        return (SLseg*)0;

    return (SLseg*)nd->val;
}

void SweepLine::remove( SLseg* s )
{
    // remove the node from the balanced binary tree
    Tnode* nd = Tree.find(s);
    if (nd == (Tnode*)0)
        return;       // not there

    // get the above and below segments pointing to each other
    Tnode* nx = Tree.next(nd);
    if (nx != (Tnode*)0) {
        SLseg* sx = (SLseg*)(nx->val);
        sx->below = s->below;
    }
    Tnode* np = Tree.prev(nd);
    if (np != (Tnode*)0) {
        SLseg* sp = (SLseg*)(np->val);
        sp->above = s->above;
    }
    Tree.remove(nd);       // now  can safely remove it
    delete s;
}

// test intersect of 2 segments and return: 0=none, 1=intersect
int SweepLine::intersect( SLseg* s1, SLseg* s2)
{
    if (s1 == (SLseg*)0 || s2 == (SLseg*)0)
        return FALSE;       // no intersect if either segment doesn't exist

    // check for consecutive edges in polygon
    int e1 = s1->edge;
    int e2 = s2->edge;
    if (((e1+1)%nv == e2) || (e1 == (e2+1)%nv))
        return FALSE;       // no non-simple intersect since consecutive

    // test for existence of an intersect point
    float lsign, rsign;
    lsign = isLeft(s1->lP, s1->rP, s2->lP);    //  s2 left point sign
    rsign = isLeft(s1->lP, s1->rP, s2->rP);    //  s2 right point sign
    if (lsign * rsign > 0) // s2 endpoints have same sign  relative to s1
```

```
        return FALSE;       // => on same side => no intersect is possible
    lsign = isLeft(s2->lP, s2->rP, s1->lP);    //  s1 left point sign
    rsign = isLeft(s2->lP, s2->rP, s1->rP);    //  s1 right point sign
    if (lsign * rsign > 0) // s1 endpoints have same sign  relative to s2
        return FALSE;       // => on same side => no intersect is possible
    // the segments s1 and s2 straddle each other
    return TRUE;            // => an intersect exists
}
//==================================================================



// simple_Polygon(): test if a Polygon is simple or not
//     Input:  Pn = a polygon with n vertices V[]
//     Return: FALSE(0) = is NOT simple
//             TRUE(1)  = IS simple
int
simple_Polygon( Polygon Pn )
{
    EventQueue  Eq(Pn);
    SweepLine   SL(Pn);
    Event*      e;                  // the current event
    SLseg*      s;                  // the current SL segment

    // This loop processes all events in the sorted queue
    // Events are only left or right vertices since
    // No new events will be added (an intersect => Done)
    while (e = Eq.next()) {         // while there are events
        if (e->type == LEFT) {      // process a left vertex
            s = SL.add(e);          // add it to the sweep line
            if (SL.intersect(  s, s->above))
                return FALSE;       // Pn is NOT simple
            if (SL.intersect(  s, s->below))
                return FALSE;       // Pn is NOT simple
        }
        else {                      // processs a right vertex
            s = SL.find(e);
            if (SL.intersect(  s->above, s->below))
                return FALSE;       // Pn is NOT simple
            SL.remove(s);           // remove it from the sweep line
        }
    }
    return TRUE;      // Pn IS simple
}
//==================================================================
```

References

Brad Appleton, C++ code for an AVL balanced Tree, see: www.bradapp.com, www.bradapp.com/ftp/src/libs/C++/AvlTrees.html (1997)

I.J. Balaban, "An Optimal Algorithm for Finding Segment Intersections", Proc. 11-th Ann. ACM Sympos. Comp. Geom., 211-219 (1995)

Ulrike Bartuschka, Kurt Mehlhorn & Stefan Naher, "A Robust and Efficient Implementation of a Sweep Line Algorithm for the Straight Line Segment Intersection Problem", Proc. Workshop on Algor. Engineering, Venice, Italy, 124-135 (1997)

Jon Bentley & Thomas Ottmann, "Algorithms for Reporting and Counting Geometric Intersections", IEEE Trans. Computers C-28, 643-647 (1979)

Mark de Berg et al, Computational Geometry : Algorithms and Applications (2nd Ed), Chapter 2 "Line Segment Intersection" (2010)

Tim Chan, "A Simple Trapezoid Sweep Algorithm for Reporting Red/Blue Segment Intersections", Proc. 6-th Can. Conf. Comp. Geom., Saskatoon, Saskatchewan, Canada, 263-268 (1994)

Bernard Chazelle & Herbert Edelsbrunner, "An Optimal Algorithm for Intersecting Line Segments in the Plane", Proc. 29-th Ann. IEEE Sympos. Found. Comp. Sci., 590-600 (1988)

Bernard Chazelle & Herbert Edelsbrunner, "An Optimal Algorithm for Intersecting Line Segments in the Plane", J. ACM 39, 1-54 (1992)

K.L. Clarkson & P.W. Shor, "Applications of Random Sampling in Computational Geometry, II", Discrete Comp. Geom. 4, 387-421 (1989)

John Hobby, "Practical Segment Intersection with Finite Precision Output", Comp. Geom. : Theory & Applics. 13(4), (1999) [Note: the original Bell Labs paper appeared in 1993]

H.G. Mairson & J. Stolfi, "Reporting and Counting Intersections between Two Sets of Line Segments", in: Theoretic Found. of Comp. Graphics and CAD, NATO ASI Series Vol. F40, 307-326 (1988)

E. Myers, "An O(E log E + I) Expected Time Algorithm for the Planar Segment Intersection Problem", SIAM J. Comput., 625-636 (1985)

Joseph O'Rourke, Computational Geometry in C (2nd Edition), Section 7.7 "Intersection of Segments" (1998)

Franco Preparata & Michael Shamos, Computational Geometry: An Introduction, Chapter 7 "Intersections" (1985)

Michael Shamos & Dan Hoey, "Geometric Intersection Problems", Proc. 17-th Ann. Conf. Found. Comp. Sci., 208-215 (1976)

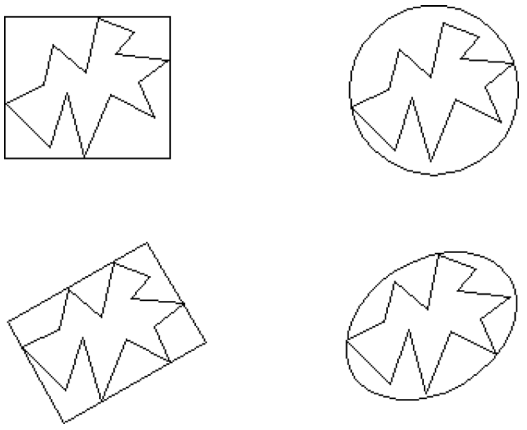
Bounding Containers

It is often useful to have a bounding container (BC), such as a bounding box or sphere, enclosing a finite geometric object. BCs can significantly speed up software for ray tracing, collision avoidance, hidden object detection, etc. Before invoking a computationally expensive intersection or containment algorithm for a complicated object, a simple test with an uncomplicated bounding container can often exclude the possibility of intersection or containment, and no further wasteful computation is needed. If a point is outside the bounding container, then it is also outside the object inside the container. If two bounding containers are disjoint, then so are the objects they contain, and thus they cannot intersect. The usefulness of such containers is sometimes only given passing mention as an obvious preprocessing test to make before attempting other algorithms. However, there are a number of different choices for bounding containers that a programmer should be familiar with.

We will restrict attention to finite linear objects, such as points, segments, triangles, polygons, and polyhedra. These objects, and collections of them, are specified by linear combinations of their vertices, and their complexity can be measured by the total number n of vertices they have. All containers we discuss can be computed directly from the set of vertices, and so we just have to consider algorithms for sets of points. Important characteristics of a good bounding container are:

Criteria	How-to-Achieve
If the BC contains the vertices of a linear object, then it must also contain the whole object.	The BC should be convex. That is, if two points are inside the BC, then the line segment joining them is also inside the BC.
It is easy to test that: 1) a point is outside the BC, 2) two BC's are disjoint, and 3) a line or ray intersects the BC.	Have a small number of easy-to-compute inequalities to test inclusion of a point in the BC.
The geometric object is closely approximated by the BC.	Minimize the area or volume of the BC.
It is efficient to compute and store the BC.	Aim for $O(n)$ time, or better, and a small constant space.
Bottom Line: Get a significant improvement in runtime speed.	Expect a large runtime speed-up in return for extra BC preprocessing time.

We consider two basic types of BC's: linear containers (such as rectangular boxes and convex polygons), and quadratic containers (such as spheres or ellipses). Example of different containers for the same object, a polygon, are shown in the diagram:



Linear Containers

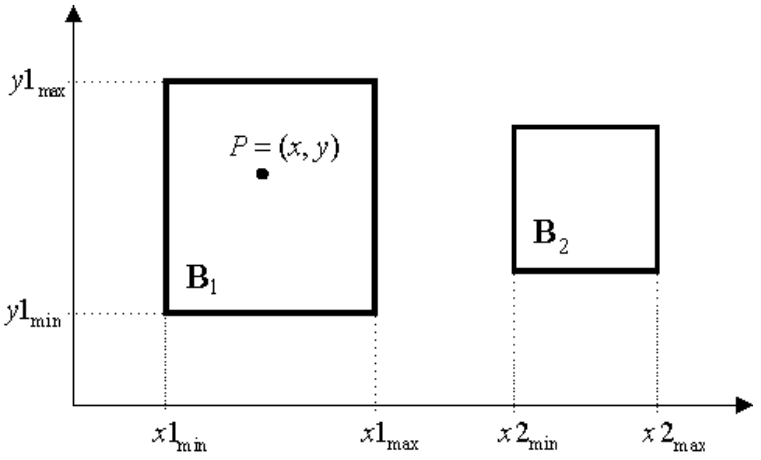
A linear container is one whose interior is specified by a finite number of linear inequalities. This implies that a bounded linear container is either a convex polygon (2D) or a convex polyhedron (3D). For example, in 2D, a container **C** could be specified by *k* inequalities: $\mathbf{f}_i(x,y) = a_i x + b_i y + c_i \le 0 \;\; \textbf{for}\, i = \mathbf{1},k$, all of which would have to be true for a point (*x,y*) to be in the region. If any one of the inequalities failed, then the test point would be outside **C**. So, when a point is outside **C**, this can be discovered on average by testing half of the inequalities. Also, each inequality defines a half-space $\mathbf{H}_i$ bounded by the line $\mathbf{L}_i$: $\mathbf{f}_i(x,y) = 0$. The region **C** is the intersection of all these half-spaces, and this implies that it is convex. When bounded, **C** is a *convex polygon* with segments of the lines $\mathbf{L}_i$ as edges.

Similarly in 3D, *k* linear inequalities: $\mathbf{f}_i(x,y) = a_i x + b_i y + c_i z + d_i \le 0 \;\; \textbf{for}\, i = \mathbf{1},k$, specify a linear container **D**. Each inequality defines a half-space $\mathbf{H}_i$ bounded by a plane $\mathscr{P}_i$. The region inside **D** is now a 3D *convex polyhedron* with convex polygons $\mathbf{C}_i$ (contained in the planes $\mathscr{P}_i$) as faces.

The Bounding Box

A "***box***" is a rectangular region whose edges are parallel to the coordinate axes, and is thus defined by its maximum and minimum extents for all axes. So, a 2D box is given by all (*x,y*) coordinates satisfying $x_{\min} \le x \le x_{\max}$ and $y_{\min} \le y \le y_{\max}$, and is specified by the extreme points $(x_{\min}, y_{\min})$ and $(x_{\max}, y_{\max})$, which are its bottom-left and top-right corners. Similarly, a 3D box is specified by it's extreme corners $(x_{\min}, y_{\min}, z_{\min})$ and $(x_{\max}, y_{\max}, z_{\max})$. Inclusion of a point $P = (x, y)$ in a box is tested by verifying that all inequalities are true; and if any one of them fails, then the point is outside the box. In 2D there are four (4) inequalities, and in 3D there are six (6). So, on the

average, an outside point will be rejected after 2 tests in 2D and 3 tests in 3D. Further, one can test whether two boxes $\mathbf{B}_1$ and $\mathbf{B}_2$ are disjoint by comparing their minimum and maximum extents. With respect to the *x*-axis, if either $x1_{max} < x2_{min}$ or $x2_{max} < x1_{min}$, then $\mathbf{B}_1$and $\mathbf{B}_2$ are disjoint. There are equivalent disjointness tests with respect to the *y*-axis and *z*-axis. If any one of these tests is true, then $\mathbf{B}_1$ and $\mathbf{B}_2$ are disjoint. If all of them are false, then the two boxes intersect, and more tests are needed to find out if objects inside the two boxes also intersect.



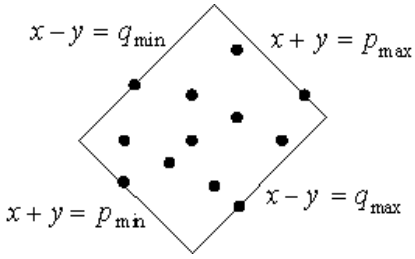
The "***bounding box***" of a finite geometric object is the box with minimal area (in 2D), or minimal volume (in 3D or higher dimensions), that contains a given geometric object. For any collection of linear objects (points, segments, polygons, and polyhedra), their bounding box is given by the minimum and maximum coordinate values for the point set **S** of all the object's *n* vertices. These values are easily computed in $O(n)$ time with a single scan of all the vertex points, sometimes while the object's vertices are being read or computed.

The bounding box is the computationally simplest of all linear bounding containers, and the one most frequently used in many applications. At runtime, the inequalities do not involve any arithmetic, and only compare raw coordinates with the precomputed *min* and *max* constants.

The Bounding Diamond

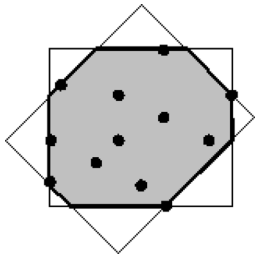
When one introduces some arithmetic, the simplest nontrivial expressions are those that just add and subtract raw coordinates. In 2D, we have the expressions $p = (x + y)$ and $q = (x - y)$, which correspond to lines with slopes of (–1) and 1. For a 2D set **S** of points, we can compute the minimum and maximum over **S** of each of these two expressions to get p_{min}, p_{max}, q_{min}, q_{max}. Then, the "***bounding diamond***" **D** for the

set **S** is the region given by coordinates (x, y) satisfying the inequalities: $p_{\min} \le (x+y) \le p_{\max}$ and $q_{\min} \le (x-y) \le q_{\max}$. Geometrically, it is a rectangle rotated by 45° and resembles a diamond.



The bounding diamond involves a small bit more computation than the bounding box, and a priori they are equivalent approximations of the sets they contain. However, after computing them both, one may be found to be better than the other. All the minimums and maximums involved (8 of them) can be computed in $O(n)$ time with a single scan of the set **S**. Then the areas of the bounding box **B** and the bounding diamond **D** can be compared, and the smaller container can be used if one wants. Since everything is rectangular, we have:

$$\text{Area}(\mathbf{B}) = (x_{\max} - x_{\min})(y_{\max} - y_{\min})$$
$$\text{Area}(\mathbf{D}) = (p_{\max} - p_{\min})(q_{\max} - q_{\min})$$



Further, one could use both containers, the box and the diamond, to get an even smaller combined "***bounding octagon"*** defined by all 8 inequalities together. In fact, this is probably the preferred usage. If one wants to test a point $P = (x, y)$ for inclusion in a polygon Ω, the only overhead is computing the expressions $(x+y)$ and $(x-y)$ just before testing the diamond inequalities after P was found to be inside the bounding box. To test disjointness of two polygons, no further expressions need to be evaluated since one only has to test the opposing parallel *max* and *min* extremes of the bounding octagon.

In 3D, one can also use the *max* and *min* of the four expressions ($x \pm y \pm z$) to get 8 inequalities that define a 3D ***"bounding octahedron"***. Specifically, the inequalities are:

$$(x+y+z)_{\min} \le x+y+z \le (x+y+z)_{\max}$$
$$(x+y-z)_{\min} \le x+y-z \le (x+y-z)_{\max}$$
$$(x-y+z)_{\min} \le x-y+z \le (x-y+z)_{\max}$$
$$(x-y-z)_{\min} \le x-y-z \le (x-y-z)_{\max}$$

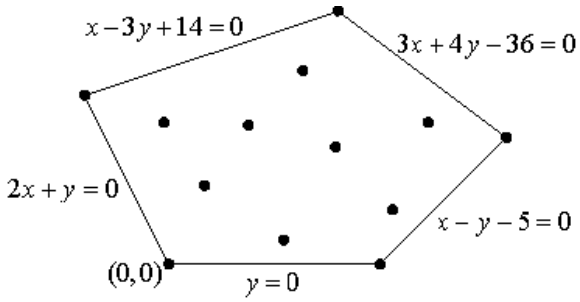
If the coordinates of a point $P=(x,y)$ fail to satisfy any of these, then P is outside the bounding octahedron. Also, two bounding octahedrons are disjoint if the *max* of an expression for one is less than the *min* of the same expression for the other. This gives 8 inequalities to test for disjointness.

One could further combine the 8 octahedron inequalities with the 6 inequalities for the 3D bounding box, and get a set of 14 inequalities which give a 3D "***bounding cuboctahedron"***. Whether it is efficient to test this many inequalities depends on the complexity of the geometric object inside the bounding container.

It should be noted though, that in *d*-dimensions with coordinates $(x_1, x_2, \ldots, x_d)$, the bounding box is given by 2*d* inequalities. But, the bounding octahedron is given by 2^d inequalities with 2^{d-1} defining expressions $(x_1 \pm x_2 \pm \ldots \pm x_d)$. Thus, the (cube)octahedron box method does not generalize efficiently to high dimensions.

The Convex Hull

The "***convex hull***" **H** of a point set **S** is the smallest convex region containing the points. Any other convex container for **S** must contain the convex hull **H** as a subset. This means that the convex hull is the bounding container that is the smallest and least area approximation for the object it contains. It is easy to show that **H**is a bounded polygon in 2D or a polyhedron in 3D with vertices from a finite set **S**. A good way to visualize the 2D convex hull is to imagine that the points of **S** are pegs stuck in a plane (or nails in a piece of wood), and that **H** is formed by an elastic band stretched around the outside of all the pegs. The band contracts to enclose the peg set as tightly as possible. This is the boundary of the convex hull. Each edge of the hull's boundary is a line segment that can be expressed as an implicit linear equation, and the half space containing the hull is given by an inequality: $ax+by+c \le 0$ in 2D or $ax+by+cz+d \le 0$ in 3D, where a,b,c,d are floating point numbers. The region inside the hull is defined by the collection of all these inequalities. For example:



The downside of using the convex hull as a container is that it may have a lot of edges (in 2D) or faces (in 3D), and to check for the inclusion of a point in the hull by testing

many independent inequalities takes a lot of computation. However, there are fast $O(\log n)$ methods to test for inclusion of a point in a 2D convex polygon. Further, it is not straightforward to test that two different noncongruent convex hulls are disjoint. This is because the hulls do not have consistent opposed parallel faces which can be tested for separation. So, although the convex hull gives a lower bound for how small a bounding container can be, it is not usually useful for non-intersection testing. Nevertheless, if one checks that all of the vertices for one convex hull are outside (not inside) another convex hull, then they are disjoint. Similarly, if all of the vertices for one convex hull are inside) another convex hull, then the first hull is contained in the second hull.

We will not pursue this topic further here, and will return to convex hulls some other time. We just note that there are many algorithms for computing the convex hull in 2D, 3D, and higher dimensions. The fastest algorithms run in $O(n \log n)$ time for the 2D or 3D hull of a set of *n* points. Many of these algorithms are described by [Preparata & Shamos, 1985, Chaps 3 & 4] and [O'Rourke, 1998, Chaps 3 & 4] both of whom have two whole chapters on computing convex hulls. O'Rourke also gives explicit C code (on his web site) for computing both the 2D and 3D convex hull for a set of points.

The Minimal Rectangle

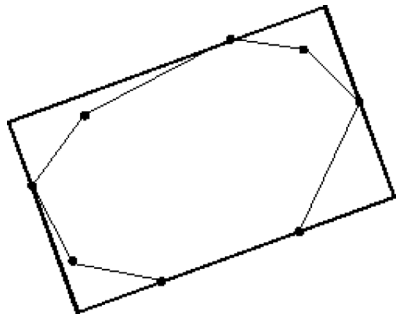
On the other hand, it is useful to find minimal area bounding containers which only involve the evaluation of a few expressions to test for inclusion. The ***minimal bounding rectangle*** is such a container. It is the minimum area (or volume in 3D) rectangle (or "***minimal bounding cuboid***" in 3D), with an arbitrary orientation, that contains the vertex set $\mathbf{S}$ of a geometric object. In 2D, it has two pairs of parallel edges, given by the *max* and *min* over $\mathbf{S}$ of two linear expressions $\mathbf{f}_1 = a_1 x + b_1 y$ and $\mathbf{f}_2 = a_2 x + b_2 y$. In 3D, there are three expressions $\mathbf{f}_i = a_i x + b_i y + c_i z$ with $i = \mathbf{1,3}$, defining 3 pairs of parallel planar faces. Without the restriction of rectangularity, one could define the ***minimal bounding parallelogram*** (or "parallelopied" in 3D) which could have an even smaller area than the minimal rectangular box. For such parallelotope regions, it is relatively easy to determine the inclusion of a point $P = (x, y)$. In 2D, the inequality tests are:

$$(a_1 x + b_1 y)_{\min} \le a_1 x + b_1 y \le (a_1 x + b_1 y)_{\max}$$
$$(a_2 x + b_2 y)_{\min} \le a_2 x + b_2 y \le (a_2 x + b_2 y)_{\max}$$

Again, if any one inequality fails, then the point is outside the rectangle or parallelogram. Although the more general parallelogram would be a useful container, most work in computational geometry has focused on algorithms for finding the minimal rectangle. This is partly because these algorithms are slightly more efficient.

In 2D, there is a well-known algorithm for finding the minimum rectangle using a technique referred to as the "rotating calipers" ([Toussaint, 1983], [Pirzadeh, 1999]).

This technique applies to a convex polygon, and can find the minimal rectangle in $O(n)$ time. For a nonconvex object, one must first compute its convex hull, which can be done in $O(n \log n)$ time. The caliper method depends on the observation that (in 2D) one side of the minimal rectangle must coincide with one edge of the convex polygon it must contain [Freeman & Shapira, 1975], as shown in the diagram:



Thus, one only has to consider the orientations given by edges of the convex polygon. Further, there is an efficient method to determine the minimal rectangle containing a specific edge as one moves successively around the edges of the polygon. This method can be imagined as having two pairs of parallel caliper lines pivoting about opposed vertices of the polygon until one caliper line meets an edge. Then, the area of the rectangle for that orientation is noted, and the caliper line starts pivoting about the next vertex of that edge. As a result, the algorithm keeps track of the other sides of the rectangle associated with each edge of the polygon. So, a single $O(n)$ scan of the polygon edges (actually, a 90° rotation is enough) suffices to find the minimal rectangle.

In 3D, it is considerably more complicated to find the minimum volume bounding cuboid of a convex polyhedron, because the faces of the minimal cuboid may not coincide with any face of the polyhedron. This is illustrated by the example of a regular tetrahedron, where only its edges coincide with the faces of the bounding cuboid.

The current fastest 3D bounding cuboid algorithm is by [O'Rourke, 1985], which runs in $O(n^3)$ time, has several special cases, and is more difficult to implement.
O'Rourke's algoruthm is based on his observation that "*A box of minimal volume circumscribing a convex polyhedron must have at least two adjacent faces that contain edges of the polyhedron*." No faster exact algorithm has been found.

Regardless, since $O(n^3)$ algorithms can be very slow in practice for large point sets, there is considerable interest in fast approximations for the minimum cuboid in 3D and higher dimensions. Several methods have been suggested, such as: (1) only consider cuboids that have a face coinciding with a polyhedron face (which is $O(n^2)$ as discussed in [O'Rourke, 1985], and can be 2 times too large), (2) principal component

analysis on the point set (which is fast, but often results in a poor solution), (3) brute-force methods that sample many possible orientations for a cuboid, (4) reduce a large point set to a smaller approximate set (e.g., by selecting the cells in a discrete grid that contain set points) [Barequet & Har-Peled, 1999], and others. All these methods have trade-offs between the solution accuracy versus the efficiency of execution.

A recent promising method [Chang, Gorissen & Melchior, 2011] involves optimization on the manifold $\mathrm{SO}(3,\mathrm{R})$ of 3D rotations. To find the minimum of a cuboid objective function, they use a hybrid of a simplex search and a genetic algorithm. Their Matlab® experimental results are very good, with accurate approximations (less than 1% error in over 98% of the cases), runtimes of less than 5 seconds for convex 3D polyhedra with up to 3000 vertices, and asymptotic speed observed to be $\mathrm{O}(n)$ for the cases they studied (with up to 10,000 vertices).

There are also algorithms for the approximate bounding cuboid in higher dimensions. Recent work can be found in Chapter 6 “Cuboids” from [Gartner, Wagner & Welzl, 2008], which is based on work by [Barequet & Har-Peled, 2001]. They show that for an *n* vertex convex polytope $\mathbf{\Omega}$ in dimension *d*, one can compute in $\mathrm{O}(d^2 n)$ time a cuboid $\mathbf{C}$ that contains $\mathbf{\Omega}$ and satisfies $\mathbf{vol(C)} \le 2^d d! \mathbf{vol(C(\Omega))}$, where $\mathbf{C(\Omega)}$ is the exact smallest bounding cuboid of $\mathbf{\Omega}$.

Quadratic Containers

There are two useful bounding containers whose boundary is given by a quadratic expression: the bounding ball (a disk or sphere), and the bounding ellipse or ellipsoid. These are somewhat harder to compute than linear containers, but they are more efficient to apply at runtime, especially in higher dimensions.

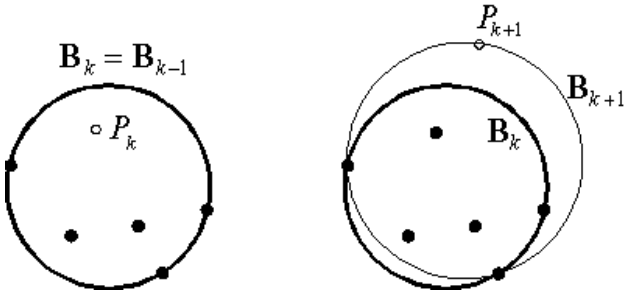
The Bounding Ball

The "***bounding ball***" (or "bounding disk" or "bounding sphere") of a geometric object is the smallest circle (in 2D space) or sphere (in 3D and *d*-D space) containing the object. It is also called the "***minimal spanning sphere***" of the object. In any dimension, the bounding ball of a linear geometric object (defined by the set $\mathbf{S}$of its vertices set) is unique, and is specified by a center point *C* and a radius *R*.

It is easy to test that a point is inside a bounding ball by checking that it is within distance *R* of the center C. Also, two balls, say $\mathbf{B}_1$ and $\mathbf{B}_2$, are disjoint if the distance between their centers is greater than the sum of their radii, that is: $\mathbf{d}(C_1, C_2) > R_1 + R_2$. These basic tests are very simple and efficient no matter what the dimension of the objects involved. This is not true of linear containers where the number of inequality

tests increases with the dimension of the space. On the other hand, it is more difficult to precompute the bounding ball of a point set since quadratic expressions are involved.

There are several algorithms for exactly computing the minimal ball for a set $\mathbf{S}$ of n points. Some authors have noted that the minimal ball can be derived directly from the "Furthest-point Voronoi Diagram" which can be computed in $O(n \log^3 n)$ time. This is a nice theoretical result, but is not easy to implement in practice. On the other hand, it has been shown ([Welzl, 1991], [de Berg et al, 2000]) that the bounding ball can be computed using a randomized linear programming algorithm that runs in $O(n)$ expected time, specifically in $O(d\,d!n)$ expected time for a (d–1)-D space bounding ball. The algorithm is incremental: it starts with a random permutation of the point set $\mathbf{S} = \{P_1, P_2, ..., P_n\}$ and an initial ball $\mathbf{B}_2$ containing the points P_1 and P_2. It then incrementally adds the other points, constructing increasingly larger balls to contain them as needed. So, $\mathbf{B}_k$ is the bounding ball for $\{P_1, P_2, ..., P_k\}$ and then we consider P_{k+1}. If it is in $\mathbf{B}_k$, then $\mathbf{B}_{k+1} = \mathbf{B}_k$. Otherwise, it is outside $\mathbf{B}_k$ and we must expand $\mathbf{B}_k$ to get a $\mathbf{B}_{k+1}$ that includes P_{k+1}. The trick is to make sure that the $\mathbf{B}_{k+1}$ we end up with is minimal. To do this, the incremental algorithm is applied recursively to the set $\{P_{k+1}, P_1, ..., P_k\}$ with the restriction that P_{k+1} must always be on the boundary of all balls constructed. In the process of doing this, there is yet another level of recursion where balls are constrained to have 2 specific points on their boundary. After that, newly constructed balls must go through 3 non-collinear points, but in 2D three points uniquely determine a ball, and this determines $\mathbf{B}_{k+1}$. So, for a 2D ball, the algorithm has 2 levels of recursion. In 3D, there are 3 levels to get 4 non-planar points that uniquely determine a 3D sphere.



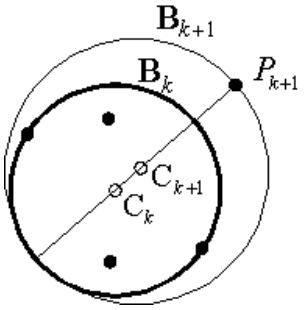
Despite sounding complicated, this algorithm is straightforward to implement (see [de Berg et al, 2000]). Further, clever heuristics can speed it up significantly in practice. An excellent fast implementation by [Bernd Gartner, 1999] is available on the web with downloadable source code. His routine works for 2D, 3D, and higher dimensions up to about 30-D with only about 300 lines of C++ code. It is numerically stable, and is very fast in low dimensions.

A Fast Approximate Bounding Ball

Another extremely fast $O(n)$ algorithm by [Ritter, 1990] computes a good approximation to the bounding ball. Although he presents a 3D algorithm, his method works efficiently in any *d*-dimensional space. His deterministic incremental algorithm avoids doing any recursion, and just scans the vertex list twice. For a point set **S**, it works as follows :

1. First, an initial good guess is made for a bounding ball **B**. This is done by finding two points of **S** that are far from each other, and using the line between them as an initial diameter. Then, the center of this diameter is the initial ball center, and half the length of the diameter is the initial ball radius. One can find points of **S** that are far from each other by selecting ones on opposite extremes of the bounding box for **S**.

2. Next, each point P of **S** is tested for inclusion in the current ball (by simply checking that its distance from the center is less than or equal to the radius). If the next point P_{k+1} is in the current $\mathbf{B}_k$, then $\mathbf{B}_{k+1} = \mathbf{B}_k$ and one just proceeds to the next point. But if P_{k+1} is outside $\mathbf{B}_k$, then $\mathbf{B}_k$ is expanded just enough to include both itself as well as the point P_{k+1}. This is done by drawing a line from P_{k+1} to the current center C_k of $\mathbf{B}_k$ and extending it further to intersect the far side of $\mathbf{B}_k$. This segment is then used as the new diameter for an expanded ball $\mathbf{B}_{k+1}$. As shown in the diagram, it clearly contains the prior ball $\mathbf{B}_k$ and (thus) all points of **S** already considered, and no additional recursion is needed.



Each of the two stages of this algorithm does $O(n)$ efficient computations. Ritter estimates that this approximation is within 5% of the actual minimal bounding ball. Below, we give a simplified 2D implementation fastBall().

The Bounding Ellipsoid

As you might expect, the "***bounding ellipsoid***" (or "***minimal spanning ellipsoid***") is the smallest volume ellipsoid (smallest area ellipse in 2D) that contains a vertex point set **S**. Like the minimal rectangle, it can have any orientation. Thus, it also is a very tight approximation for the object it contains, and is an excellent container. Also, it is not difficult to test inclusion of a point in an ellipse, especially in 2D where the sum of the distances of a point from the two focal points is a constant on the boundary of the ellipse.

For a set of non-collinear points (non-planar in 3D) the bounding ellipsoid exists and is unique. [Welzl, 1991] gives a fast randomized algorithm for computing the bounding ellipsoid in $O(d\,d!n)$ expected time for (d–1) dimensional space. A fast implementation has been developed by [Gartner & Schonherr, 1997]. The Welzl algorithm is basically the same as his randomized incremental bounding ball algorithm. Having constructed the bounding ellipse $\mathbf{E}_k$ for the first k points, one then adds the next point P_{k+1}. If P_{k+1} is inside $\mathbf{E}_k$, then $\mathbf{E}_{k+1} = \mathbf{E}_k$. Otherwise the algorithm must inductively construct $\mathbf{E}_{k+1}$. Again, the algorithm must recursively construct the minimum ellipse for $\{P_{k+1}, P_1, ..., P_k\}$ with the restriction that P_{k+1} is on the boundary of $\mathbf{E}_{k+1}$. The recursion stops when there are enough constraints to uniquely define an ellipse. Of course, this is somewhat more difficult to determine than for a spherical ball. See [Gartner & Schonherr, 1997] for details.

Nevertheless, especially in higher dimensions, the bounding ellipsoid is superior to the minimal cuboid in many ways. It is unique, whereas the minimal cuboid is not. There is a reasonable algorithm to implement it. And it is a good approximation of the object it contains, much better than the minimal cuboid. In fact, if $\mathbf{E}(\mathbf{S})$ is the bounding ellipsoid for a point set **S** with convex hull $\mathbf{C}(\mathbf{S})$ in dimension d, then:

$$(1/d)\mathbf{E}(\mathbf{S}) \subset \mathbf{C}(\mathbf{S}) \subset \mathbf{E}(\mathbf{S})$$

where scaling is with respect to the center of $\mathbf{E}(\mathbf{S})$.

Implementations

```
// Copyright 2001, 2012, 2021 Dan Sunday
// This code may be freely used and modified for any purpose
// providing that this copyright notice is included with it.
// There is no warranty for this code, and the author of it cannot
// be held liable for any real or imagined damage 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;}
//        operators for:
//            Point  = Point ± Vector
//            Vector = Point - Point
//            Vector = Vector ± Vector
//            Vector = Scalar * Vector    (scalar product)
//            Vector = Vector / Scalar    (scalar division)
//
//    Ball with a center and radius {Point center; float radius;}
//===================================================================

// dot product which allows  vector operations in arguments
#define dot(u,v)   ((u).x * (v).x + (u).y * (v).y)
#define norm2(v)   dot(v,v)        // norm2 = vector length squared
#define norm(v)    sqrt(norm2(v)) // norm = vector length
#define d(u,v)     norm(u-v)      // distance = norm of difference



// fastBall(): get a fast approximation for the 2D bounding ball
//              (algorithm given by [Jack Ritter, 1990])
//    Input:  an array P[] of n points (2D xy coords)
//    Output: a bounding ball = {Point center; float radius;}
void
fastBall( Point P[], int n, Ball* B)
{
    Point C;                            // Center of ball
    float rad, rad2;                    // radius and radius squared
    float xmin, xmax, ymin, ymax;       // bounding box extremes
    int   Pxmin, Pxmax, Pymin, Pymax;   // index of P[] at extremes

    // find a large diameter to start with
    // first get the bounding box and P[] extreme points for it
    xmin = xmax = P[0].x;
    ymin = ymax = P[0].y;
    Pxmin = Pxmax = Pymin = Pymax = 0;
    for (int i=1; i<n; i++) {
        if (P[i].x < xmin) {
            xmin = P[i].x;
            Pxmin = i;
        }
        else if (P[i].x > xmax) {
            xmax = P[i].x;
            Pxmax = i;
        }
        if (P[i].y < ymin) {
            ymin = P[i].y;
            Pymin = i;
        }
        else if (P[i].y > ymax) {
            ymax = P[i].y;
```

```
            Pymax = i;
        }
    }
    // select the largest extent as an initial ball diameter
    Vector dPx = P[Pxmax] - P[Pxmin]; // diff of Px max and min
    Vector dPy = P[Pymax] - P[Pymin]; // diff of Py max and min
    float dx2 = norm2(dPx); // Px diff squared
    float dy2 = norm2(dPy); // Py diff squared
    if (dx2 >= dy2) {       // x direction is largest extent
        C = P[Pxmin] + (dPx / 2.0);   // Center = midpoint
        rad2 = norm2(P[Pxmax] - C);   // radius squared
    }
    else {                  // y direction is largest extent
        C = P[Pymin] + (dPy / 2.0);   // Center = midpoint
        rad2 = norm2(P[Pymax] - C);   // radius squared
    }
    rad = sqrt(rad2);

    // now check that all points P[i] are in the ball
    // and if not, expand the ball just enough to include them
    Vector dP;
    float dist, dist2;
    for (int i=0; i<n; i++) {
        dP = P[i] - C;
        dist2 = norm2(dP);
        if (dist2 <= rad2)     // P[i] is inside the ball already
            continue;
        // P[i] not in ball, so expand ball  to include it
        dist = sqrt(dist2);
        rad = (rad + dist) / 2.0;       // enlarge radius enough
        rad2 = rad * rad;
        C = C + ((dist-rad)/dist) * dP; // shift Center toward P[i]
    }
    B->center = C;
    B->radius = rad;
    return;
}
```

References

Gill Barequet and Sariel Har-Peled, “Efficiently approximating the minimum-volume bounding box of a point set in three dimensions”, Journal of Algorithms 38 91-109 (2001)

Mark de Berg et al, Computational Geometry: Algorithms and Applications (3rd Edition), Section 4.7 "Smallest Enclosing Disks" (2010)

Chia-Tche Chang, Bastien Gorissen & Samuel Melchior, “Fast oriented bounding box optimization on the rotation group SO(3; R)”, ACM Transactions on Graphics, Vol. 30, No. 5, Article 122, Oct. 2011

H. Freeman & R. Shapira, “Determining the minimum-area encasing rectangle for an arbitrary closed curve”, Comm ACM 18(7) 409-413 (1975)

Bernd Gartner, "Smallest Enclosing Balls - Fast and Robust in C++" Web Site (1999)

Bernd Gartner & Sven Schonherr, "Smallest Enclosing Ellipses - Fast and Exact", Tech. Report B 97-03, Free Univ. Berlin, Germany (1997)

Bernd Gartner, Uli Wagner & Emo Welzl, Lecture Notes for Approximate Methods in Geometry (2008); esp Chapter 5 “Approximate Smallest Enclosing Balls” and Chapter 6 “Cuboids”.

Joseph O'Rourke, "Finding Minimal Enclosing Boxes", Int'l J. Comp. Info. Sci. 14 (1985), 183-199

Joseph O'Rourke, Computational Geometry in C (2nd Edition) (1998)

Hormoz Pirzadeh, Rotating Calipers Web Site (1999)

Franco Preparata & Michael Shamos, Computational Geometry: An Introduction (1985)

Jack Ritter, "An Efficient Bounding Sphere" in Graphics Gems (1990)

Godfried Toussaint, "Solving geometric problems with the rotating calipers" in Proc. IEEE MELECON'83 (1983)

Emo Welzl, "Smallest enclosing disks (balls and ellipsoids)" in New Results and New Trends in Computer Science, Lecture Notes in Computer Science, Vol. 555 (1991), 359-370

Convex Hull of a 2D Point Set

Computing a convex hull (or just "hull") is one of the first sophisticated geometry algorithms, and there are many variations of it. The most common form of this algorithm involves determining the smallest convex set (called the "convex hull") containing a discrete set of points. This algorithm also applies to a polygon, or just any set of line segments, whose hull is the same as the hull of its vertex point set. There are numerous applications for convex hulls: collision avoidance, hidden object determination, and shape analysis to name a few. The hull is a minimal linear bounding container, although it is not as easy to use as other containers.

The most popular hull algorithms are the "Graham scan" algorithm [Graham, 1972] and the "divide-and-conquer" algorithm [Preparata & Hong, 1977]. Implementations of both these algorithms are readily available (see [O'Rourke, 1998]). Both are $O(n \log n)$ time algorithms, but the Graham has a low runtime constant in 2D and runs very fast there. However, the Graham algorithm does not generalize to 3D and higher dimensions whereas the divide-and-conquer algorithm has a natural extension. We do not consider 3D algorithms here (see [O'Rourke, 1998] for more information).

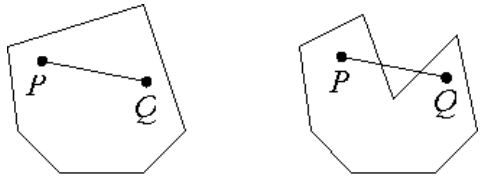
Here is a list of some well-known 2D hull algorithms. Let n = # points in the input set, and h = # vertices on the output hull. Note that $h \leq n$, so $nh \leq n^2$. The list is ordered by date of first publication.

Algorithm	**Speed**	**Discovered By**
Brute Force	$O(n^4)$	[Anon, the dark ages]
Gift Wrapping	$O(nh)$	[Chand & Kapur, 1970]
Graham Scan	$O(n \log n)$	[Graham, 1972]
Jarvis March	$O(nh)$	[Jarvis, 1973]
QuickHull	$O(nh)$	[Eddy, 1977], [Bykat, 1978]
Divide-and-Conquer	$O(n \log n)$	[Preparata & Hong, 1977]
Monotone Chain	$O(n \log n)$	[Andrew, 1979]
Incremental	$O(n \log n)$	[Kallay, 1984]
Marriage-before-Conquest	$O(n \log h)$	[Kirkpatrick & Seidel, 1986]

Convex Hulls

The convex hull of a geometric object (such as a point set or a polygon) is the smallest convex set containing that object. There are many equivalent definitions for a convex set **S**. The most basic of these is:

Def 1. A set **S** is ***convex*** if whenever two points P and Q are inside **S**, then the whole line segment PQ is also in **S**.



Convex NonConvex

But this definition does not readily lead to algorithms for constructing convex sets. More useful definitions are the following.

Def 2. A set **S** is ***convex*** if it is exactly equal to the intersection of all the half planes containing it.

It can be shown that these two definitions are equivalent. However, the second one gives us a better computational handle, especially when the set **S** is the intersection of a **finite** number of half planes. In this case, the boundary of **S** is polygon in 2D, and polyhedron in 3D, with which it can be identified.

Def 3. The ***convex hull*** of a finite point set **S** = $\{P\}$ is the smallest 2D convex polygon $\mathbf{\Omega}$ (or polyhedron in 3D) that contains **S**. That is, there is no other convex polygon (or polyhedron) $\mathbf{\Lambda}$ with $\mathbf{S} \subseteq \mathbf{\Lambda} \subset \mathbf{\Omega}$.

And this convex hull has the smallest area and the smallest perimeter of all convex polygons that contain **S**.

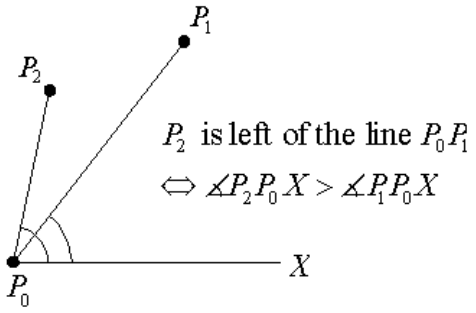
2D Hull Algorithms

For this algorithm we will cover two similar fast 2D hull algorithms: the Graham scan, and Andrew's Monotone Chain scan. They both use a similar idea, and are implemented as a stack. In practice, they are both very fast, but Andrew's algorithm will execute slightly faster since its sort comparisons and rejection tests are more efficient. An implementation of Andrew's algorithm is given below in our `chainHull_2D()` routine.

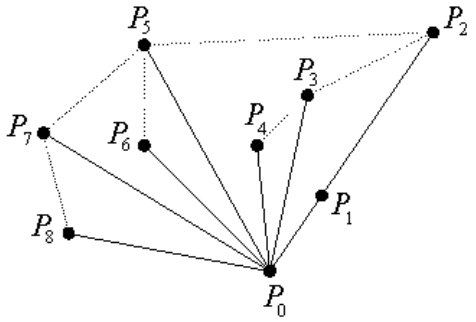
The "Graham Scan" Algorithm

The Graham scan algorithm [Graham, 1972] is often cited ([Preparata & Shamos, 1985], [O'Rourke, 1998]) as the first real "computational geometry" algorithm. As the size of the geometric problem (namely, n = the number of points in the set) increases, it achieves the optimal asymptotic efficiency of $O(n \log n)$ time. This algorithm and its implementation has been covered in great detail by [O'Rourke, 1998, Sect 3.5, 72-86] with downloadable C code available from his web site. We do not repeat that level of detail here, and only give a conceptual overview of the algorithm. The $O(n \log n)$ time for the Graham scan is spent doing an initial radial sort of the input set points. After that, the algorithm employs a stack-based method which runs in just $O(n)$ time. In fact, the method performs at most $2n$ simple stack push and pop operations. Thus, it executes very rapidly, bounded only by the speed of sorting.

Let **S** = {P} be a finite set of points. The algorithm starts by picking a point in **S** known to be a vertex of the convex hull. This can be done in $O(n)$ time by selecting the rightmost lowest point in the set; that is, a point with first a minimum (lowest) y coordinate, and second a maximum (rightmost) x coordinate. Call this base point P_0. Then, the algorithm sorts the other points P in **S** radially by the increasing counter-clockwise (ccw) angle the line segment P_0P makes with the x-axis. If there is a tie and two points have the same angle, discard the one that is closest to P_0. For efficiency, it is important to note that the sort comparison between two points P_1 and P_2 can be made without actually computing their angles. In fact, computing angles would use slow inaccurate trigonometry functions, and doing these computations would be a bad mistake. Instead, one just observes that P_2 would make a greater angle than P_1 if (and only if) P_2 lies on the left side of the directed line segment P_0P_1 as shown in the following diagram. This condition can be tested by a fast accurate computation that uses only 5 additions and 2 multiplications.

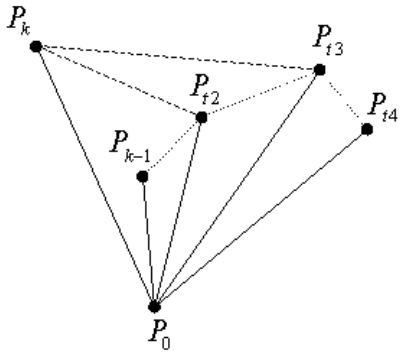


After sorting, let the ccw-radially-ordered point set be $\mathbf{S} = \{P_0, P_1, ..., P_{n-1}\}$. It looks like a fan with a pivot at the point P_0.



We next loop through the points of **S** one-by-one testing for convex hull vertices. The algorithm is an inductive incremental procedure using a stack of points. At each stage, we save (on the stack) the vertex points for the convex hull of all points already processed. This is the induction condition. We start with $P_0$and P_1 on the stack. Then at the k-th stage, we add the next point P_k, and compute how it alters the prior convex hull. Because of the way **S** was sorted, P_k is outside the hull of the prior points P_i with $i < k$, and it must be added as a new hull vertex on the stack. But it's addition may cause previous stack points to no longer be a

hull vertices. If this happens, the previous points must be popped off the stack and discarded. One tests for this by checking if the new point P_k is to the left or the right of the line joining the top two points of the stack. Again, we use the routine `isLeft()` to quickly make this test. If P_k is on the left of the top segment, then prior hull vertices remain intact, and P_k gets pushed onto the stack. But, if it is on the right side of the top segment, then the prior point at the stack top will get absorbed inside the new hull, and that prior point must be popped off the stack. This test against the line segment at the stack top continues until either P_k is left of that line or the stack is reduced to the single base point P_0. In either case, P_k gets pushed onto the stack, and the algorithm proceeds to the next point P_{k+1} in the set. The different possibilities involved are illustrated in the following diagram.



Old Stack $= S_{k-1} = \{P_0, \ldots, P_{t3}, P_{t2}, P_{k-1}\}$
P_k right of line $P_{t2}P_{k-1} \Rightarrow$ pop P_{k-1} off stack
P_k right of line $P_{t3}P_{t2} \Rightarrow$ pop P_{t2} off stack
P_k left of line $P_{t4}P_{t3} \Rightarrow$ push P_k onto stack
New Stack $= S_k = \{P_0, \ldots, P_{t3}, P_k\}$

It is easy to understand why this works by viewing it as an incremental algorithm. The old stack $\mathbf{S}_{k-1} = \{P_0, \ldots, P_{t2}, P_{k-1}\}$, with P_{k-1} at the top, is the convex hull of all points P_i with $i < k$. The next point P_k is outside this hull since it is left of the line P_0P_{k-1} which is an edge of the $\mathbf{S}_{k-1}$ hull. To incrementally extend $\mathbf{S}_{k-1}$ to include P_k, we need to find the two tangents from P_k to $\mathbf{S}_{k-1}$. One tangent is clearly the line P_kP_0. *The other is a line* P_kP_t such that P_k is left of the segment in $\mathbf{S}_{k-1}$ preceding P_t and is right of the segment following P_t (when it exists). This uniquely characterizes the second tangent since $\mathbf{S}_{k-1}$ is a convex polygon. The way to find P_t is simply to search from the top of the stack down until the point with the property is found. The points above P_t in $\mathbf{S}_{k-1}$ are easily seen to be contained inside the triangle $\Delta P_0P_tP_k$, and are thus no longer on the hull extended to include P_k. So, they can be discarded by popping them off the stack during the search for P_t. Then, the k-th convex hull is the new stack $\mathbf{S}_k = \{P_0, \ldots, P_t, P_k\}$.

At the end, when $k = n-1$, the points remaining on the stack are precisely the ordered vertices of the convex hull's polygon boundary. Note that for each point of $\mathbf{S}$ there is one push and at most one pop operation, giving at most $2n$ stack operations for the whole algorithm.

This procedure is summarized by the following pseudo-code.

Pseudo-Code: Graham Scan Algorithm

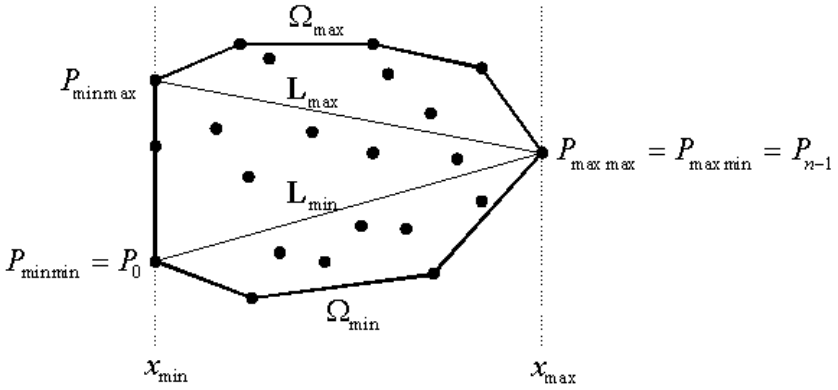
```
Input: a  set of points S = {P = (P.x,P.y)}

Select the rightmost lowest point P0 in S
Sort S radially (ccw) about P0 as a center {
    Use isLeft() comparisons
    For ties, discard the closer points
}
Let P[N] be the sorted array of points with P[0]=P0

Push P[0] and P[1] onto a stack Ω

while i < N
{
    Let PT1 = the top point on Ω
    If (PT1 == P[0]) {
        Push P[i] onto Ω
        i++     // increment i
    }
    Let PT2 = the second top point on Ω
    If (P[i] is strictly left of the line  PT2 to PT1) {
        Push P[i] onto Ω
        i++     // increment i
    }
    else
        Pop the top point PT1 off the stack
}

Output: Ω = the convex hull of S.
```

Andrew's Monotone Chain Algorithm

[Andrew, 1979] discovered an alternative to the Graham scan that uses a linear lexographic sort of the point set by the *x* and *y*-coordinates. This is an advantage if this ordering is already known for a set, which is sometimes the case. But even if sorting is required, this is a faster sort than the angular Graham-scan sort with its more complicated comparison function. The "Monotone Chain" algorithm computes the upper and lower hulls of a monotone chain of points, which is why we refer to it as the "Monotone Chain" algorithm. Like the Graham scan, it runs in $O(n \log n)$ time due to the sort time. After that, it only takes $O(n)$ time to compute the hull. This algorithm also uses a stack in a manner very similar to Graham's algorithm.

First the algorithm sorts the point set $\mathbf{S} = \{P_0, P_1, ..., P_{n-1}\}$ by increasing *x* and then *y* coordinate values. Let the minimum and maximum *x*-coordinates be x_{min} and x_{max}. Clearly, $P_0.x = x_{\min}$, but there may be other points with this minimum *x*-coordinate. Let $P_{\min\min}$ be a point in **S** with $P.x = x_{\min}$ first and then min *y* among all those points. Also, let $P_{\min\max}$ be the point with $P.x = x_{\min}$ first and then max *y* second. Note that $P_{\min\min} = P_{\min\max}$ when there is a unique *x*-minimum point. Similarly define $P_{\max\min}$ and $P_{\max\max}$ as the points with $P.x = x_{\max}$ first, and then *y* min or max second. Again note that $P_{\max\min} = P_{\max\max}$ when there is a

unique *x*-maximum point. Next, join the lower two points, P_{minmin} and P_{maxmin} to define a lower line $\mathbf{L}_{min}$. Also, join the upper two points, P_{minmax} and P_{maxmax} to define an upper line $\mathbf{L}_{max}$. These points and lines are shown in the following example diagram.



The algorithm now proceeds to construct a lower convex vertex chain $\mathbf{\Omega}_{min}$ below $\mathbf{L}_{min}$ and joining the two lower points P_{minmin} and P_{maxmin}; and also an upper convex vertex chain $\mathbf{\Omega}_{max}$ above $\mathbf{L}_{max}$ and joining the two upper points P_{maxmax} and P_{minmax}. Then the convex hull $\mathbf{\Omega}$ of **S** is constructed by joining $\mathbf{\Omega}_{min}$ and $\mathbf{\Omega}_{max}$ together.

The lower or upper convex chain is constructed using a stack algorithm almost identical to the one used for the Graham scan. For the lower chain, start with P_{minmin} on the stack. Then process the points of **S** in sequence. For $\mathbf{\Omega}_{min}$, only consider points strictly below the lower line $\mathbf{L}_{min}$. Suppose that at any stage, the points on the stack are the convex hull of points below $\mathbf{L}_{min}$ that have already been processed. Now consider the next point P_k that is below $\mathbf{L}_{min}$. If the stack contains only the one point P_{minmin} then put P_k onto the stack and proceed to the next stage. Otherwise, determine whether P_k is strictly left of the line between the top two points on the stack. If it is, put P_k onto the stack and proceed. If it is not, pop the top point off the stack, and test P_k against the stack again. Continue until P_k gets pushed onto the stack. After this stage, the stack again contains the vertices of the lower hull for the points already considered. The geometric rationale is exactly the same as for the Graham scan. After all points have been processed, push P_{maxmin} onto the stack to complete the lower convex chain.

The upper convex chain $\mathbf{\Omega}_{max}$ is constructed in an analogous manner. But, process **S** in decreasing order $\{P_{n-1}, P_{n-2}, ..., P_0\}$, starting at P_{maxmax}, and only considering points above $\mathbf{L}_{max}$. Once the two hull chains have been found, it is easy to join them together (but be careful to avoid duplicating the endpoints).

Pseudo-Code: Andrew's Monotone Chain Algorithm

```
    Input: a set S = {P = (P.x,P.y)} of N points

    Sort S by increasing x and then y-coordinate.
    Let P[] be the sorted array of N points.

    Get the points with 1st x min or max and 2nd y min or max
        minmin = index of P with min x first and min y second
        minmax = index of P with min x first and max y second
        maxmin = index of P with max x first and min y second
        maxmax = index of P with max x first and max y second

    Compute the lower hull stack as follows:
    (1) Let L_min be the lower line joining P[minmin] with  P[maxmin].
    (2) Push P[minmin] onto the stack.
    (3) for i = minmax+1 to maxmin-1 (the points between xmin and xmax)
        {
            if (P[i] is above or on L_min)
                 Ignore it and continue.
            while (there are at least 2 points on the stack)
            {
                 Let P_T1 = the top point on the stack.
                 Let P_T2 = the second point on the stack.
                 if (P[i] is strictly left of the line from P_T2 to P_T1)
                     break out of this while loop.
                 Pop the top point P_T1 off the stack.
            }
            Push P[i] onto the stack.
        }
    (4) Push P[maxmin] onto the stack.

    Similarly, compute the upper hull stack.

    Let Ω = the join of the lower and upper hulls.

    Output: Ω = the convex hull of S.
```

Implementation

Here is "C++" for the Monotone Chain Hull algorithm.

```
// Copyright 2001, 2012, 2021 Dan Sunday
// This code may be freely used and modified for any purpose
// providing that this copyright notice is included with it.
// There is no warranty for this code, and the author of it cannot
// be held liable for any real or imagined damage from its use.
// Users of this code must verify correctness for their application.


// Assume that a class is already given for the object:
//    Point with coordinates {float x, y;}
//===================================================================
```

```
// isLeft(): tests if a point is Left|On|Right of an infinite line.
//    Input:  three points P0, P1, and P2
//    Return: >0 for P2 left of the line through P0 and P1
//            =0 for P2 on the line
//            <0 for P2 right of the line
inline float
isLeft( Point P0, Point P1, Point P2 )
{
    return (P1.x - P0.x)*(P2.y - P0.y) - (P2.x - P0.x)*(P1.y - P0.y);
}
//===================================================================


// chainHull_2D(): Andrew's monotone chain 2D convex hull algorithm
//     Input:  P[] = an array of 2D points
//                   presorted by increasing x and y-coordinates
//             n =  the number of points in P[]
//     Output: H[] = an array of the convex hull vertices (max is n)
//     Return: the number of points in H[]
int
chainHull_2D( Point* P, int n, Point* H )
{
    // the output array H[] will be used as the stack
    int    bot=0, top=(-1);   // indices for bottom and top of the stack
    int    i;                 // array scan index

    // Get the indices of points with min x-coord and min|max y-coord
    int minmin = 0, minmax;
    float xmin = P[0].x;
    for (i=1; i<n; i++)
        if (P[i].x != xmin) break;
    minmax = i-1;
    if (minmax == n-1) {       // degenerate case: all x-coords == xmin
        H[++top] = P[minmin];
        if (P[minmax].y != P[minmin].y)  // a  nontrivial segment
            H[++top] =  P[minmax];
        H[++top] = P[minmin];            // add polygon endpoint
        return top+1;
    }

    // Get the indices of points with max x-coord and min|max y-coord
    int maxmin, maxmax = n-1;
    float xmax = P[n-1].x;
    for (i=n-2; i>=0; i--)
        if (P[i].x != xmax) break;
    maxmin = i+1;

    // Compute the lower hull on the stack H
    H[++top] = P[minmin];       // push  minmin point onto stack
    i = minmax;
    while (++i <= maxmin)
    {
        // the lower line joins P[minmin]  with P[maxmin]
        if (isLeft( P[minmin], P[maxmin], P[i])  >= 0 && i < maxmin)
            continue;           // ignore P[i] above or on the lower line

        while (top > 0)         // the stack has at least 2 points
```

```
        {
            // test if  P[i] is left of the line at the stack top
            if (isLeft(  H[top-1], H[top], P[i]) > 0)
                 break;         // P[i] is a new hull  vertex
            else
                 top--;         // pop top point off  stack
        }
        H[++top] = P[i];        // push P[i] onto stack
    }

    // Next, compute the upper hull on the stack H above the bottom hull
    if (maxmax != maxmin)       // if  distinct xmax points
         H[++top] = P[maxmax];  // push maxmax point onto stack
    bot = top;                  // the bottom point of the upper hull
    i = maxmin;
    while (--i >= minmax)
    {
        // the upper line joins P[maxmax]  with P[minmax]
        if (isLeft( P[maxmax], P[minmax], P[i])  >= 0 && i > minmax)
            continue;           // ignore P[i] below or on the upper line

        while (top > bot)     // at least 2 points on the upper stack
        {
            // test if  P[i] is left of the line at the stack top
            if (isLeft(  H[top-1], H[top], P[i]) > 0)
                 break;         // P[i] is a new hull  vertex
            else
                 top--;         // pop top point off  stack
        }
        H[++top] = P[i];        // push P[i] onto stack
    }
    if (minmax != minmin)
        H[++top] = P[minmin];  // push  joining endpoint onto stack

    return top+1;
}
```

References

S.G. Akl & Godfried Toussaint, "Efficient Convex Hull Algorithms for Pattern Recognition Applications", Proc. 4th Int'l Joint Conf. on Pattern Recognition, Kyoto, Japan, 483-487 (1978)

A.M. Andrew, "Another Efficient Algorithm for Convex Hulls in Two Dimensions", Info. Proc. Letters 9, 216-219 (1979)

A. Bykat, "Convex Hull of a Finite Set of Points in Two Dimensions", Info. Proc. Letters 7, 296-298 (1978)

W. Eddy, "A New Convex Hull Algorithm for Planar Sets", ACM Trans. Math. Software 3(4), 398-403 (1977)

Ronald Graham, "An Efficient Algorithm for Determining the Convex Hull of a Finite Point Set", Info. Proc. Letters 1, 132-133 (1972)

R.A. Jarvis, "On the Identification of the Convex Hull of of a Finite Set of Points in the Plane", Info. Proc. Letters 2, 18-21 (1973)

M. Kallay, "The Complexity of Incremental Convex Hull Algorithms in R^d", Info. Proc. Letters 19, 197 (1984)

D.G. Kirkpatrick & R. Seidel, "The Ultimate Planar Convex Hull Algorithm?", SIAM Jour. Comput. 15, 287-299 (1986)

Joseph O'Rourke, Computational Geometry in C (2nd Edition), Chap. 3 "Convex Hulls in 2D" (1998), with online code at www.science.smith.edu/~jorourke/books/compgeom.html

Franco Preparata & Michael Shamos, Computational Geometry: An Introduction, Chap. 3 "Convex Hulls: Basic Algorithms" (1985)

Franco Preparata & S.J. Hong, "Convex Hulls of Finite Sets of Points in Two and Three Dimensions", Comm. ACM 20, 87-93 (1977)

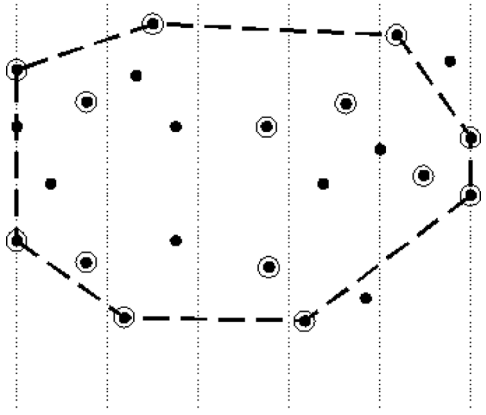
Convex Hull Approximation

Previously, we looked at some of the fastest $O(n \log n)$ algorithms for computing the convex hull of a planar point set. We now present an algorithm that gives a fast approximation for the 2D convex hull. The advantage of this algorithm is that it is much faster with just an $O(n)$ runtime. In many applications, an approximate hull suffices, and the gain in speed can be significant for very large point sets or polygons.

The BFP Approximate Hull Algorithm

Sometimes a good approximation for the convex hull is sufficient for an application, especially where the *n*data points have a sampling error. The main advantage of an approximation algorithm would be that it is significantly more efficient. In 2D, a very efficient approximate convex hull algorithm is the one of [Bentley-Faust-Preparata, 1982] (BFP) which runs in $O(n)$ time.

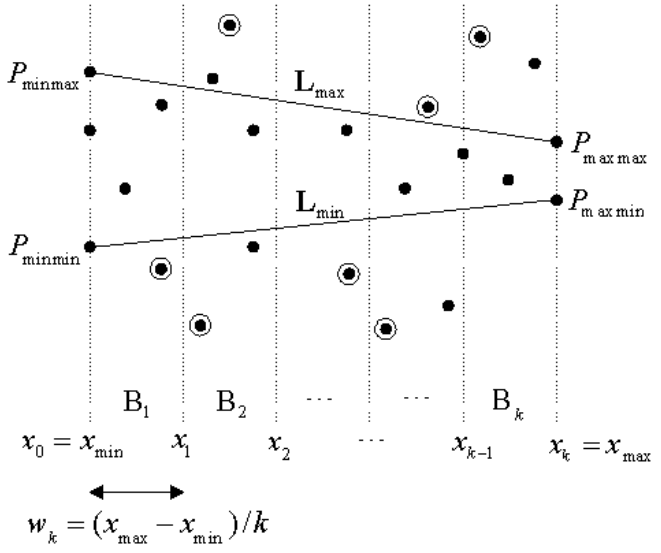
It is known that the speed of an algorithm for the convex hull of a 2D point set **S** is dominated by the need to initially sort the *n* points of the set, which takes $O(n \log n)$ time. After that, the best algorithms, such as the Graham Scan or the Monotone Chain, only require $O(n)$ time. So, to improve efficiency, one must replace the sort operation with something else. The Bentley-Faust-Preparata (BFP) algorithm does this by dividing the plane into narrow vertical strips, and (in a single linear pass) finding the maximum and minimum points of **S** in each strip. Then, they use the minimum points to get the lower hull chain, and the maximum points to get the upper hull chain. These chains are computed using a variation of the monotone chain algorithm [Andrew, 1979] that we previously presented. The "sorting" of the maximum and minimum point sets comes for free in the natural ordering of the vertical strip bins used by the algorithm. So, we replace sorting with a linear $O(n)$ procedure. But, one pays for this by possibly throwing away points which are non-extreme in their vertical strip bin, yet are still on the convex hull boundary. However, they can't be too far from the real convex hull, and thus we get a good approximation to the real convex hull. Further, the approximation gets better as we make the strips narrower. The extreme points in these vertical strips are circled in the following diagram that also shows the approximate hull as a dashed contour. In this example, several points are outside the approximate hull.



More accurately, the BFP algorithm finds the extrema of the vertical strip bins as follows:

1. Scan **S** once to find, in $O(n)$ time:
   A. the minimum and maximum x-coordinates, $x_{\min}$ and $x_{\max}$.
   B. $P_{\min\min}$ and $P_{\min\max}$ be the points with $P.x = x_{\min}$ first and then min y or max y respectively.
   C. $P_{\max\min}$ and $P_{\max\max}$ be the points with $P.x = x_{\max}$ first and then min y or max y respectively.
   D. $\mathbf{L}_{\min}$ is the lower line joining the two points $P_{\min\min}$ and $P_{\max\min}$.
   E. $\mathbf{L}_{\max}$ is the upper line joining the two points $P_{\min\max}$ and $P_{\max\max}$.
2. Divide the range $[x_{\min}, x_{\max}]$ into k equal subranges, $\mathbf{B}_1$ to $\mathbf{B}_k$, each of width $w_k = (x_{\max} - x_{\min})/k$. That is: $\mathbf{B}_i = [x_{i-1}, x_i]$ where $x_i = x_{\min} + iw_k$.
3. Define a (k+2)-element array of bins B[] to record the points B[].*min* and B[].*max* in each $\mathbf{B}_i$ with the minimum and maximum y values in that bin. Let:
   1. B[0] be the bin for $P.x = x_{\min}$. Thus, B[0].*min* = $P_{\min\min}$, and B[0].*max* = $P_{\min\max}$.
   2. B[i] (i=1,k) be the bin for the subrange B_i.
   3. B[k+1] be the bin for $P.x = x_{\max}$. Thus, B[k+1].*min* = $P_{\max\min}$, and B[k+1].*max* = $P_{\max\max}$.
4. Scan **S** again to find, in $O(n)$ time:
   A. B[i].*min* = the y-minimum point P below $\mathbf{L}_{\min}$ with $P.x$ in B_i.
   B. B[i].*max* = the y-maximum point P above $\mathbf{L}_{\max}$ with $P.x$ in B_i.

Note that for some ranges B_i these points, B[i].*min* and B[i].*max*, will not exist. But that is ok because all we are trying to do is select points that are valid candidates for the lower or upper convex hulls. That is why we only consider minimum points below $\mathbf{L}_{\min}$, since the lower hull must be below this line. Similarly, only maximum points above $\mathbf{L}_{\max}$ are considered since the upper hull is above this line. These selections are shown in the following diagram. Notice that there are fewer extreme points due to filtering with respect to $\mathbf{L}_{\min}$ and $\mathbf{L}_{\max}$. This will make the algorithm run slightly faster.



The algorithm now proceeds by computing the lower and upper hulls using a (simplified) variation of Andrew's monotone chain hull algorithm. Putting it all together in pseudo-code, we have:

Pseudo-Code: BFP Approximate Hull Algorithm

```
Input: a set S = {P = (P.x,P.y)} of n points
       k = the approximation accuracy (large k = more accurate)

Get the points with 1st x min or max and 2nd y min or max
    minmin = index of P with min x first and min y second
    minmax = index of P with min x first and max y second
    maxmin = index of P with max x first and min y second
    maxmax = index of P with max x first and max y second.
Let L_min be the lower line joining P[minmin] with P[maxmin].
Let L_max be the upper line joining P[minmax] with P[maxmax].

Let B[k+2] be an array of subrange bins as described above.
Compute the y-min and y-max in each subrange i=1,k to get:
    B[i].min = the P in the i-th subrange with min y below L_min
    B[i].max = the P in the i-th subrange with max y above L_max

Compute the lower hull stack as follows:
(1) Push P[minmin] onto the stack.
(2) for (each bin B[i], i = 1 to k)
    {
        if (B[i].min does not exist)
             continue;
        Let Pi = B[i].min.
        while (there are at least 2 points on the stack)
        {
             Let PT1 = the top point on the stack.
             Let PT2 = the second point on the stack.
             if (Pi is strictly left of the line from PT2 to PT1)
                 break out of this while loop.
             Pop the top point PT1 off the stack.
        }
        Push Pi onto the stack.
    }
(3) Push P[maxmin] onto the stack.

Similarly, compute the upper hull stack.

Let Ω = the join of the  lower and upper hulls.

Output: Ω = the convex hull of S.
```

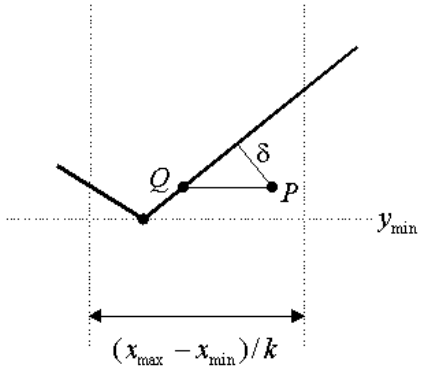
Our implementation `nearHull_2D()` of this algorithm in C++ is given below.

Error Analysis

As for how close the approximate hull is to the exact convex hull, it can be shown that:

> [Preparata & Shamos, 1985, p-156, **Theorem 4.6**]
> Any point P in **S** that is not inside the approximate convex hull is within distance $(x_{\max} - x_{\min})/k$ of it.

Thus, the approximation can be made arbitrarily close by picking a large enough *k*. This is easily illustrated by the following diagram.



which demonstrates that a point P below the lower hull and above the bin *y*-minimum, has an error: $\delta < \mathrm{d}(P,Q) < (x_{max} - x_{min})/k$.

Implementation

Here is a "C++" implementation of the approximate hull algorithm.

```
// Copyright 2001, 2012, 2021 Dan Sunday
// This code may be freely used and modified for any purpose
// providing that this copyright notice is included with it.
// There is no warranty for this code, and the author of it cannot
// be held liable for any real or imagined damage from its use.
// Users of this code must verify correctness for their application.

// Assume that a class is already given for the object:
//    Point with coordinates {float x, y;}
//===================================================================

// isLeft(): tests if a point is Left|On|Right of an infinite line.
//    Input:  three points P0, P1, and P2
//    Return: >0 for P2 left of the line through P0 and P1
//            =0 for P2 on the line
//            <0 for P2 right of the line
inline float
isLeft( Point P0, Point P1, Point P2 )
{
    return (P1.x - P0.x)*(P2.y - P0.y) - (P2.x - P0.x)*(P1.y - P0.y);
}
//===================================================================

#define NONE (-1)

typedef struct range_bin Bin;
struct range_bin {
    int    min;    // index of min point P[] in bin (>=0 or NONE)
    int    max;    // index of max point P[] in bin (>=0 or NONE)
};
```

```
// nearHull_2D(): the BFP fast approximate 2D convex hull algorithm
//     Input:  P[] = an (unsorted) array of 2D points
//               n = the number of points in P[]
//               k = the approximation accuracy (large k = more accurate)
//     Output: H[] = an array of the convex hull vertices (max is n)
//     Return: the number of points in H[]
int
nearHull_2D( Point* P, int n, int k, Point* H )
{
    int    minmin=0, minmax=0;
    int    maxmin=0, maxmax=0;
    float  xmin = P[0].x, xmax = P[0].x;
    Point* cP;                  // the current point being considered
    int    bot=0, top=(-1);  // indices for bottom and top of the stack

    // Get the points with (1) min-max x-coord, and (2) min-max y-coord
    for (int i=1; i<n; i++) {
        cP = &P[i];
        if (cP->x <= xmin) {
            if (cP->x <  xmin) {          // new xmin
                xmin = cP->x;
                minmin = minmax = i;
            }
            else {                        // another xmin
                if (cP->y < P[minmin].y)
                    minmin = i;
                else if (cP->y > P[minmax].y)
                    minmax = i;
            }
        }
        if (cP->x >= xmax) {
            if (cP->x > xmax) {           // new xmax
                xmax = cP->x;
                maxmin = maxmax = i;
            }
            else {                        // another xmax
                if (cP->y < P[maxmin].y)
                    maxmin = i;
                else if (cP->y > P[maxmax].y)
                    maxmax = i;
            }
        }
    }
    if (xmin == xmax) {      //  degenerate case: all x-coords == xmin
        H[++top] = P[minmin];             // a point, or
        if (minmax != minmin)             // a nontrivial segment
            H[++top] =  P[minmax];
        return top+1;                     // one or two points
    }

    // Next, get the max and min points in the k range bins
    Bin*   B = new Bin[k+2];   // first allocate the bins
    B[0].min = minmin;          B[0].max = minmax;        // set bin 0
    B[k+1].min = maxmin;        B[k+1].max = maxmax;      // set bin k+1
    for (int b=1; b<=k; b++) { // initially nothing is in the other bins
        B[b].min = B[b].max = NONE;
    }
    for (int b, i=0; i<n; i++) {
        cP = &P[i];
```

```
    if (cP->x == xmin || cP->x == xmax)  // already have bins 0 and k+1
        continue;
    // check if a lower or upper point
    if (isLeft( P[minmin], P[maxmin], *cP) < 0) {  // below lower line
        b = (int)( k  * (cP->x - xmin) / (xmax - xmin) ) + 1;  // bin #
        if (B[b].min  == NONE)       // no min point in this range
             B[b].min = i;           //  first min
        else if (cP->y < P[B[b].min]->y)
             B[b].min = i;           // new  min
        continue;
    }
    if (isLeft( P[minmax], P[maxmax], *cP) > 0) {  // above upper line
        b = (int)( k  * (cP->x - xmin) / (xmax - xmin) ) + 1;  // bin #
        if (B[b].max == NONE)        // no max point in this range
             B[b].max = i;           //  first max
        else if (cP->y > P[B[b].max]->y)
             B[b].max = i;           // new  max
        continue;
    }
}

// Now, use the chain algorithm to get the lower and upper  hulls
// the output array H[] will be used as the stack
// First, compute the lower hull on the stack H
for (int i=0; i <= k+1; ++i)
{
    if (B[i].min == NONE)   // no min point in this range
        continue;
    cP = &P[ B[i].min ];    // select the current min point

    while (top > 0)         // there are at least 2 points on the stack
    {
        // test if current point is left of the line at the stack top
        if (isLeft(  H[top-1], H[top], *cP) > 0)
             break;         // cP is a new hull vertex
        else
             top--;         // pop top point off stack
    }
    H[++top] = *cP;         // push current point onto stack
}

// Next, compute the upper hull on the stack H above the bottom hull
if (maxmax != maxmin)       // if  distinct xmax points
     H[++top] = P[maxmax];  // push maxmax point onto stack
bot = top;                  // the bottom point of the upper hull stack
for (int i=k; i >= 0; --i)
{
    if (B[i].max == NONE)   // no max  point in this range
        continue;
    cP = &P[ B[i].max ];    //  select the current max point

    while (top > bot)       // at least 2 points on the upper stack
    {
        // test if  current point is left of the line at the stack top
        if (isLeft( H[top-1], H[top], *cP) > 0)
             break;         // current point is a new hull vertex
        else
             top--;         // pop top point off stack
    }
    H[++top] = *cP;         // push current point onto stack
```

```
    }
    if (minmax != minmin)
        H[++top] = P[minmin];   // push joining endpoint onto stack

    delete B;                   // free bins before returning
    return top+1;               // # of points on the stack
}
```

References

A.M. Andrew, "Another Efficient Algorithm for Convex Hulls in Two Dimensions", Info. Proc. Letters 9, 216-219 (1979)

Jon Bentley, G.M. Faust & Franco Preparata, "Approximation Algorithms for Convex Hulls", Comm. ACM 25, 64-68 (1982)

Franco Preparata & Michael Shamos, Computational Geometry: An Introduction, Section 4.1.2 "Approximation Algorithms for Convex Hull" (1985)

Convex Hull of a Simple Polyline

Previously, we showed how to compute the convex hull of any 2D point set or polygon with no restrictions. The polygon could have been simple or not, connected or not. It could even have been just a random set of segments or points. The algorithms given, the "Graham Scan" and the "Andrew Chain", computed the hull in $O(n \log n)$ time. Now, we present a different algorithm that improves this efficiency to $O(n)$ linear time for a ***connected simple polyline*** with no abnormal self-intersections. It is a very elegant and fast algorithm. It can be applied to the boundary of any connected 2D region, such as a geographic area (country, town, lake, etc).

How is this possible? Well, the general 2D hull algorithms first sort the vertex point set in $O(n \log n)$ time, and then use a stack to compute the hull in $O(n)$ time. All the nonlinear overhead is in the initial sorting. However, this month's algorithm uses the given sequential ordering of a simple polygon's edges along with a similar algorithm using a "deque" (a double-ended queue). It turns out that the property of simplicity is enough to guarantee that the vertices on the deque are the convex hull. In fact, the algorithm by [Melkman, 1987] that we present just assumes that the vertices form a ***simple polyline*** which is more general than earlier $O(n)$ algorithms for simple polygons.

Before proceeding, we note that some polygon constructions can be performed more efficiently using the convex hull of the polygon. For example, there are efficient algorithms for tangents to and between convex polygons. And, the solution for a polygon's convex hull is also the solution for the polygon itself. Thus, having a fast $O(n)$ simple polygon hull algorithm also speeds up computing tangents for arbitrary simple polygons.

Background

The evolution of this algorithm has an interesting history that is given in detail at the outstanding web site of [Aloupis, 2000]. Aloupis describes both correct and incorrect algorithms and ideas discovered along the way, culminating in the [Melkman, 1987] algorithm, as well as attempts to make further improvements.

Initially, the $O(n)$ improvement for simple polygons was proposed, and an algorithm implementation was given, by [Sklansky, 1972]. Unfortunately, six years later, [Bykat, 1978] showed that the algorithm was flawed. Nevertheless, the Sklansky conjecture was still valid, and his central idea was used by others to develop a correct algorithm. The first valid algorithm was by [McCallum and Davis, 1979], but they had a complicated two stack method that was difficult to implement. However, this was later simplified to a single stack method by [Graham & Yao, 1983] and [Lee, 1983] (see also [Preparata & Shamos, 1985]). At this point, correct algorithms worked for simple polygons, and did $O(n)$ preprocessing to initialize a stack with a known extreme point on the hull.

Then, [Melkman, 1987] made a significant breakthrough by describing an algorithm that:

1. Works for a simple polyline (it does not have to be closed).

2. Does not do any preprocessing, and only processes the vertices sequentially once. Thus, it can be an online algorithm that reads a single input stream.

3. Uses a double-ended queue (a deque) to store the incremental hull for the processed vertices.

4. Greatly simplifies the logic of the algorithm.

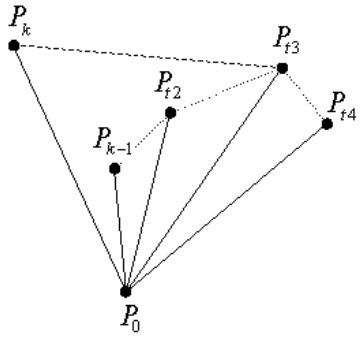
Simple Polyline Hull Algorithms

The Basic Incremental Strategy

Why should there be a faster $O(n)$ convex hull algorithms for simple polylines and polygons? To understand this, recall that most convex hull algorithms for point sets take $O(n \log n)$ time because they initially sort the n points . After that, they generally only require $O(n)$ time. So, one needs to ask why sorting is needed; that is, what does it accomplish? Consider how other algorithms proceed after sorting is done.

Most 2D convex hull algorithms use a basic incremental strategy. At the k-th stage, they have constructed the hull $\mathbf{H}_{k-1}$ of the first k points $\Omega_{k-1} = \{P_0, P_1, \ldots, P_{k-1}\}$, incrementally add the next point P_k, and then compute the next hull $\mathbf{H}_k$. How does presorting facilitate this process? The answer is that at each stage, one knows that the next point P_k is exterior to the previous hull $\mathbf{H}_{k-1}$, and thus one does not have to test for this. Otherwise, one would have to test P_k against all k edges of $\mathbf{H}_{k-1}$, resulting in $O(n^2)$ such tests totaled over all stages of the algorithm. Instead, one immediately knows that that P_k is outside $\mathbf{H}_{k-1}$, and can proceed to construct $\mathbf{H}_k$by extending $\mathbf{H}_{k-1}$. Thus, if presorting in $O(n \log n)$ time has been done, no such tests need to be made, and this results in a faster algorithm.

Additionally, most fast convex hull algorithms construct $\mathbf{H}_k$ from P_k and $\mathbf{H}_{k-1}$ in a similar manner. Namely, they find the tangents from P_k to $\mathbf{H}_{k-1}$ and use these as new edges for $\mathbf{H}_k$. These tangents could be computed from scratch at each stage in $O(\log k)$ time, but many algorithms maintain a stack that simplifies the process. Because of the presorting, the points on the stack are also sorted, and the ones closest to P_k are at the stack top. So, one just needs to push P_k onto the stack after removing (pop off) any points at the stack top that get absorbed into the interior of the new hull $\mathbf{H}_k$. This can be done by testing P_k against the line joining the top two points of the stack, and making sure that a left turn is being made for a counterclockwise (ccw) hull. The whole process is $O(n)$ linear since each point is pushed onto the stack once and popped off at most once. For example, the Graham Scan algorithm radial sort computes this tangent on the stack as illustrated:

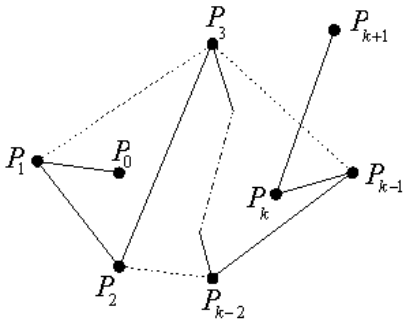


Old Stack = $\mathbf{H}_{k-1} = \{P_0, \ldots, P_{t3}, P_{t2}, P_{k-1}\}$
P_k right of line $P_{t2}P_{k-1}$ $\Rightarrow$ pop P_{k-1} off stack
P_k right of line $P_{t3}P_{t2}$ $\Rightarrow$ pop P_{t2} off stack
P_k left of line $P_{t4}P_{t3}$ $\Rightarrow$ push P_k onto stack
New Stack = $\mathbf{H}_k = \{P_0, \ldots, P_{t3}, P_k\}$

The major advantage of presorting is to avoid expensive tests for whether each new incremental point P_k is included in the previous hull $\mathbf{H}_{k-1}$.

However, if one knows that the initial ordered point set is a simple polyline, then inclusion can be tested by comparing P_k with only two edges of $\mathbf{H}_{k-1}$. This results in $O(n)$ inclusion edge tests in total over the life of the algorithm. Additionally, one can maintain $\mathbf{H}_{k-1}$ as a stack that allows tangents from an exterior P_k to be computed by popping elements from the stack. But, most importantly, the presorting is skipped, and the whole algorithm takes only $O(n)$ time.

But, how does one simplify the inclusion test? Suppose at the *k*-th stage that the simple polyline $\Omega_k = \{P_0, P_1, ..., P_k\}$ has its last vertex point P_k inside the hull $\mathbf{H}_{k-1}$. The vertices of $\mathbf{H}_{k-1}$ are a subset of the previous polyline vertices (but not necessarily in the same order). The geometry is illustrated in the diagram:

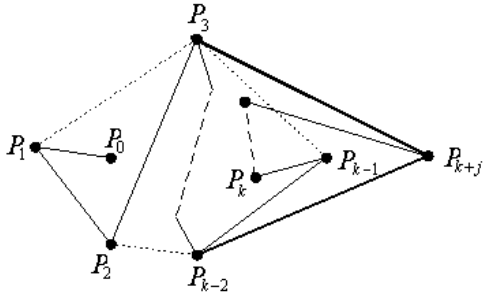


$\mathbf{H}_{k-1} = \{P_{k-1}, P_3, P_1, P_2, P_{k-2}, P_{k-1}\}$

$P_k P_{k+1}$ cannot cross $P_3 \ldots P_{k-2} P_{k-1}$

$P_k P_{k+1}$ must cross $P_3 P_{k-1}$ to leave $\mathbf{H}_{k-1}$

The important observation to make is that the end of the polyline is now trapped inside a pocket formed by Ω_k, since it cannot cross over prior edges. Successive polyline points P_{k+j} ($j > 0$) are also trapped in this pocket until they escape to the exterior of $\mathbf{H}_{k-1}$. However, a new vertex P_{k+j} can only escape across an edge of $\mathbf{H}_{k-1}$ that contains P_{k-1} since all other edges are blocked by Ω_k. Thus, each new P_{k+j} can be tested for inclusion with respect to at most two edges of $\mathbf{H}_{k-1}$ until P_{k+j} is found to be outside one of those edges. After that, tangents have to be found from this external vertex to the prior hull $\mathbf{H}_{k-1}$ to form the extended hull $\mathbf{H}_{k+j}$.



$\mathbf{H}_{k-1} = \{P_{k-1}, P_3, P_1, P_2, P_{k-2}, P_{k-1}\}$

P_{k+j} is the next vertex exterior to $\mathbf{H}_{k-1}$

$P_{k+j} P_3$ and $P_{k+j} P_{k-2}$ are tangent to $\mathbf{H}_{k-1}$

$\mathbf{H}_{k+j} = \{P_{k+j}, P_3, P_1, P_2, P_{k-2}, P_{k+j}\}$

All simple polygon or polyline convex hull algorithms implement this strategy in one form or another.

The Melkman Algorithm

[Melkman, 1987] devised an ingenious method for organizing and implementing the operations to compute the hull of a simple polyline that we will now describe. [**Note**: We have modified

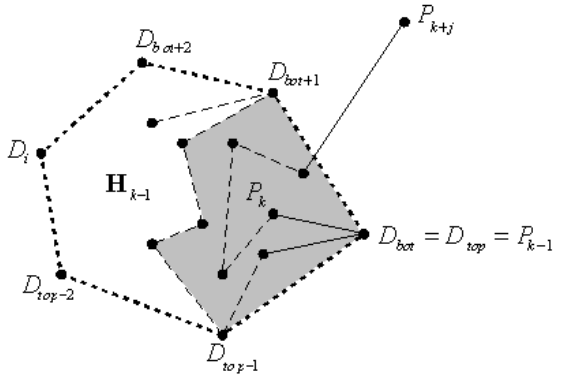
Melkman's algorithm to produce a convex hull with a ccw orientation, whereas the published algorithm gives a clockwise hull. We have also changed his notation to match ours].

The strategy of the Melkman algorithm is straightforward. It sequentially processes each of the polyline vertices in order. Let the input polyline be given by the ordered vertex set: $\mathbf{\Omega} = \{P_0, P_1, ..., P_n\}$. At each stage, the algorithm determines and stores (on a double-ended queue) those vertices that form the ordered hull for all polyline vertices considered so far. Then, the next vertex P_k is considered. It satisfies one of two conditions: either (1) it is inside the currently constructed hull, and can be ignored; or (2) it is outside the current hull, and becomes a new hull vertex extending the old hull. However, in case (2), vertices that are on the list for the old hull, may become interior to the new hull, and need to be discarded before adding the new vertex to the new list.

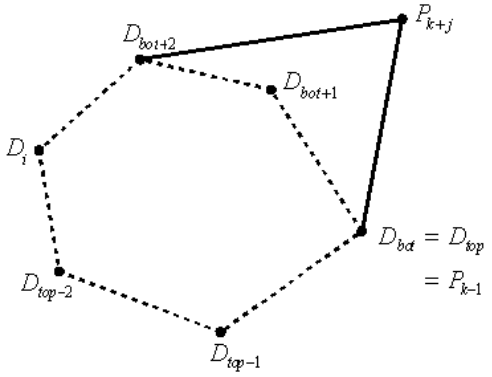
The double-ended queue (called a "deque") has both a top and a bottom. At both ends of the deque, elements can be either added or removed. At the top, we say an element is pushed or popped; while at the bottom, we say an element is inserted or deleted. The deque is given by an ordered list $\mathbf{\Phi} = \{D_{bot}, ..., D_{top}\}$ where *bot* is the index at the bottom, and *top* is for the top of $\mathbf{\Phi}$. The elements D_i are vertices that form a polyline. When $D_{top} = D_{bot}$, then $\mathbf{\Phi}$ forms a polygon. In the Melkman hull algorithm, after processing vertex P_k, the deque $\mathbf{\Phi}_k$ satisfies:

1. The polygon $\mathbf{\Phi}_k$ is the ccw convex hull $\mathbf{H}_k$ of the vertices $\mathbf{\Omega}_k = \{P_0, P_1, ..., P_k\}$ already processed.
2. $D_{top} = D_{bot}$ is the most recent vertex processed that was added to $\mathbf{\Phi}_k$.

If P_k is inside $\mathbf{H}_{k-1}$, then $\mathbf{\Phi}_k = \mathbf{\Phi}_{k-1}$, and there is no associated processing. In this case, because the polyline is $\mathbf{\Omega}_k$ simple, P_k is inside the subregion of $\mathbf{H}_{k-1}$ bounded by the the vertices $D_{bot}, D_{bot+1}, ..., D_{top-1}, D_{top}$, where "..." is the subpolyline of $\mathbf{\Omega}_k$ that joins D_{bot+1} and D_{top-1}. And, P_{k+j} can only escape from this subregion by crossing over one of the edge segments $D_{bot}D_{bot+1}$ at the bottom or $D_{top-1}D_{top}$ at the top of $\mathbf{\Phi}_k$. One can easily test for this by determining whether P_{k+j} is left of these edge segments. When it is left of both segments, this implies that P_{k+j} is still inside $\mathbf{H}_{k-1}$; but as soon as P_{k+j} becomes right of either segment, then the vertex is exterior to $\mathbf{H}_{k-1}$. Thus, at every incremental stage, inclusion of P_k in $\mathbf{H}_{k-1}$ can be determined with only two `isLeft()` tests.



When P_k is exterior to $\mathbf{H}_{k-1}$, we must then change $\mathbf{\Phi}_{k-1}$ to produce a new deque $\mathbf{\Phi}_k$ that satisfies the above two conditions. Clearly, by adding P_k to both the bottom and top of the deque, condition (2) is satisfied and P_k will be inside the new polygon defined by $\mathbf{\Phi}_k$. However, other points already in $\mathbf{\Phi}_{k-1}$ may get absorbed into the new hull $\mathbf{H}_k$, and they need to be removed before P_k is added to the deque ends. As previously noted, vertices are removed from the two ends of $\mathbf{\Phi}_{k-1}$ until the lines from P_k to the remaining deque endpoints form tangents to $\mathbf{H}_{k-1}$. Again, this is easily determined by testing whether P_k is left of the edge segments at the bottom and top of the deque. If it is not to the left of a segment, then the current vertex at the end of the deque would be inside $\mathbf{H}_k$, and we must remove that vertex from the deque. This continues until P_k is left of both the bottom and top edge segments of the deque. After that, we push P_k onto both ends, producing the required new deque $\mathbf{\Phi}_k$.



$\Phi_{k-1} = \{D_{bot}, \ldots, D_{top}\}$
P_{k+j} is right of $D_{bot}D_{bot+1} \Rightarrow$ delete D_{bot}
P_{k+j} is right of $D_{bot+1}D_{bot+2} \Rightarrow$ delete D_{bot+1}
P_{k+j} is left of $D_{bot+2}D_{bot+3} \Rightarrow$ insert P_{k+j}
P_{k+j} is left of $D_{top-1}D_{top} \Rightarrow$ push P_{k+j}
$\Phi_k = \{P_{k+j}, D_{bot+2}, \ldots, D_{top}, P_{k+j}\}$

The speed of this algorithm is easily analyzed. Each vertex of Ω can get put on the deque at most twice (once at each end). Thereafter, elements on the deque can be removed at most once. Each of these events, to potentially add or remove a vertex to/from the deque, is associated with exactly one (constant) `isLeft()` test. Thus, the worst case behavior of the Melkman algorithm is bounded by $3n$ tests and queue operations. The best case behavior would have $2n$ tests and only 4 queue operations (when the initial triangle $\Delta P_0P_1P_2$ is the final hull). Thus, the Melkman algorithm is a very efficient and elegant $O(n)$ time and space algorithm.

Pseudo-Code: Melkman Algorithm

```
Input: a simple polyline Ω with n vertices P[i]

Put first 3 vertices onto deque D[] so that:
a) 3rd vertex P[2] is at bottom and top of D[]
b) on D[] they form a ccw triangle

While there are more polyline vertices of Ω to process
Get the next vertex P[i]
{
    Note that:
    a) D[] is now the convex hull of already processed vertices
    b) D[bot] = D[top] = the last vertex added to D[]

    // Test if P[i] is inside D (as a polygon)
    If P[i] is left of both D[bot]D[bot+1] and D[top-1]D[top]
        Skip P[i] and Continue with the next vertex
    // Otherwise P[i] extends the hull and must be added to D[]
```

```
        // Get the tangent to the bottom
        While P[i] is right of D[bot]D[bot+1]
            Delete D[bot] from the bottom of D[]
        Insert P[i] at the bottom of D[]

        // Get the tangent to the top
        While P[i] is right of D[top-1]D[top]
            Pop D[top] from the top of D[]
        Push P[i] onto the top of D[]
    }
```

Output: D[] = the ccw convex hull of Ω

Implementation

Here is a "C++" implementation of this algorithm.

```
// Copyright 2001, 2012, 2021 Dan Sunday
// This code may be freely used and modified for any purpose
// providing that this copyright notice is included with it.
// There is no warranty for this code, and the author of it cannot
// be held liable for any real or imagined damage from its use.
// Users of this code must verify correctness for their application.


// Assume that a class is already given for the object:
//    Point with coordinates {float x, y;}
//===================================================================


// isLeft(): test if a point is Left|On|Right of an infinite line.
//    Input:  three points P0, P1, and P2
//    Return: >0 for P2 left of the line through P0 and P1
//            =0 for P2 on the line
//            <0 for P2 right of the line
inline float
isLeft( Point P0, Point P1, Point P2 )
{
    return (P1.x - P0.x)*(P2.y - P0.y) - (P2.x - P0.x)*(P1.y - P0.y);
}



// simpleHull_2D(): Melkman's 2D simple polyline O(n) convex hull algorithm
//    Input:  P[] = array of 2D vertex points for a simple polyline
//            n   = the number of points in V[]
//    Output: H[] = output convex hull array of vertices (max is n)
//    Return: h   = the number of points in H[]
int
simpleHull_2D( Point* P, int n, Point* H )
{
    // initialize a deque D[] from bottom to top so that the
    // 1st three vertices of P[] are a ccw triangle
    Point* D = new Point[2*n+1];
    int bot = n-2, top = bot+3;    // initial bottom and top deque indices
    D[bot] = D[top] = P[2];        // 3rd vertex is at both bot and top
    if (isLeft(P[0], P[1], P[2]) > 0) {
        D[bot+1] = P[0];
        D[bot+2] = P[1];           // ccw vertices are: 2,0,1,2
```

```
    }
    else {
        D[bot+1] = P[1];
        D[bot+2] = P[0];           // ccw vertices are: 2,1,0,2
    }

    // compute the hull on the deque D[]
    for (int i=3; i < n; i++) {   // process the rest of vertices
        // test if next vertex is inside the deque hull
        if ((isLeft(D[bot], D[bot+1], P[i]) > 0) &&
            (isLeft(D[top-1], D[top], P[i]) > 0) )
                continue;         // skip an interior vertex

        // incrementally add an exterior vertex to the deque hull
        // get the rightmost tangent at the deque bot
        while (isLeft(D[bot], D[bot+1], P[i]) <= 0)
            ++bot;                // remove bot of deque
        D[--bot] = P[i];          // insert P[i] at bot of deque

        // get the leftmost tangent at the deque top
        while (isLeft(D[top-1], D[top], P[i]) <= 0)
            --top;                // pop top of deque
        D[++top] = P[i];          // push P[i] onto top of deque
    }

    // transcribe deque D[] to the output hull array H[]
    int h;        // hull vertex counter
    for (h=0; h <= (top-bot); h++)
        H[h] = D[bot + h];

    delete D;
    return h-1;
}
```

References

Greg Aloupis, A History of Linear-time Convex Hull Algorithms for Simple Polygons, McGill Univ website (2000) at cgm.cs.mcgill.ca/~athens/cs601/

A. Bykat, "Convex hull of a finite set of points in two dimensions", Info. Proc. Letters 7, 296-298 (1978)

Ronald Graham & F. Yao, "Finding the convex hull of a simple polygon", J. Algorithms 4(4), 324-33 (1983)

D.T. Lee, "On finding the convex hull of a simple polygon", Int'l J. Comp. & Info. Sci. 12(2), 87-98 (1983)

D. McCallum and D. Davis, "A linear algorithm for finding the convex hull of a simple polygon", Info. Proc. Letters 9, 201-206 (1979)

A. Melkman, "On-line construction of the convex hull of a simple polygon", Info. Proc. Letters 25, 11-12 (1987)

Franco Preparata & Michael Shamos, Computational Geometry: An Introduction, Section 4.1.4 "Convex hull of a simple polygon" (1985)

Sklansky J., "Measuring Concavity on a Rectangular Mosaic", IEEE Transactions on Computing 21, p-1355 (1972)

Line and Convex Polytope Intersection

One of the most common computer graphics computations is the clipping of a line segment with a convex polygonal object. There is an efficient algorithm for determining this that works in all dimensions, and we will describe its implementation for 2D and 3D convex polygons and polyhedrons. It is also an efficient method for determining if a line segment or ray do not intersect the convex object, in applications that need to determine visibility or collision avoidance/detection.

For this, it is very useful to have convex bounding containers for more complicated objects, as previously discussed. The convex bounding container will have a smaller number of facets (2D edges or 3D faces) than a complicated object, which may have hundreds or thousands of them. The axis-oriented box (AOB) container has only $2n$ facets in n dimensional space. Also, the convex hull is the smallest convex container that can closely approximate an object. Efficient algorithms for constructing 2D convex hulls were previously discussed. For efficency in applications, it can be useful to have a hierarchy of containers, including bounding balls, boxes, and the hull.

Several efficient algorithms have been developed for computing the intersection (or not) of an oriented line segment with with a convex polytope. One that is frequently used in implementations, referred to as "parametric line-clipping" [Foley et al, 1996], was originally developed by [Cyrus & Beck, 1978] and later improved by [Liang & Barsky, 1984]. Their algorithm is usually described as a method for clipping a segment with a 2D rectangular window. However, it easily generalizes to 2D convex polygons and 3D convex polyhedrons, and is a very efficient algorithm in both cases. The root of this efficiency is because one only has to determine the intersections with the hyperplanes (ie: 2D lines or 3D planes) for the facets of the convex container. That is, one does not have to determine if the intersection is interior to the object's facets. This greatly simplifies and speeds up the algorithm.

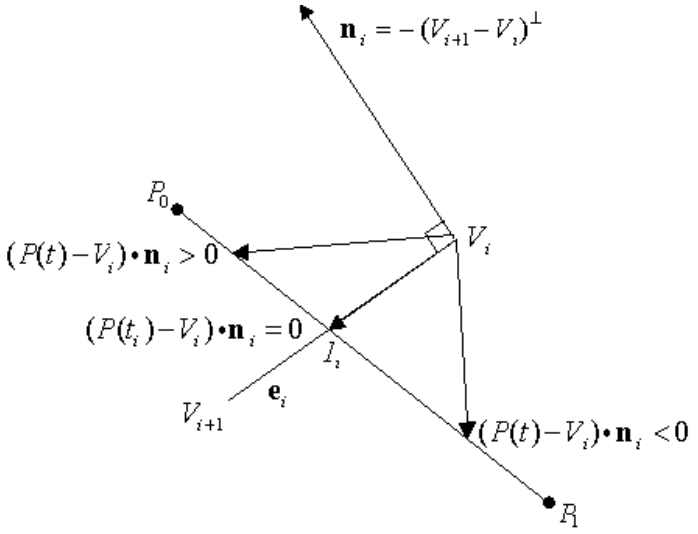
Throughout, let **S** be a line segment between endpoints P_0 and P_1. The extended line through P_0 and P_1 is given by the parametric equation: $P(t) = P_0 + t(P_1 - P_0) = P_0 + t\,\mathbf{v}$ with $\mathbf{v} = P_1 - P_0$ the line direction vector. Then, the segment **S** contains those points $P(t)$ with $0 \le t \le 1$. This representation is valid for all dimensions.

Intersect a Segment and a Convex Polygon (2D)

Let a convex polygon Ω be given by n vertices $V_0, V_1, \ldots, V_{n-1}$ going counterclockwise (ccw) around the polygon, and let $V_n = V_0$. Also let $\mathbf{e}_i$ be the i-th edge (line segment) V_iV_{i+1} for $i = 0, n-1$; and $\mathbf{ev}_i = (e_{i1}, e_{i2}) = V_{i+1} - V_i$ be the edge vector. Then,

an *outward-pointing* normal vector for $\mathbf{e}_i$ is given by: $\mathbf{n}_i = -\mathbf{ev}_i^{\perp} = (e_{i2}, -e_{i1})$, where "$\perp$" is the 2D perp-operator.

We first compute the intersection of the (extended) line $P(t)$ with the extended line for a single edge $\mathbf{e}_i$. Consider the following diagram:



As indicated, intersection occurs when $(P(t) - V_i) \bullet \mathbf{n}_i = 0$, since any vector parallel to the edge $\mathbf{e}_i$ is perpendicular to the edge normal vector. Substituting for $P(t)$ and solving for t, we get:

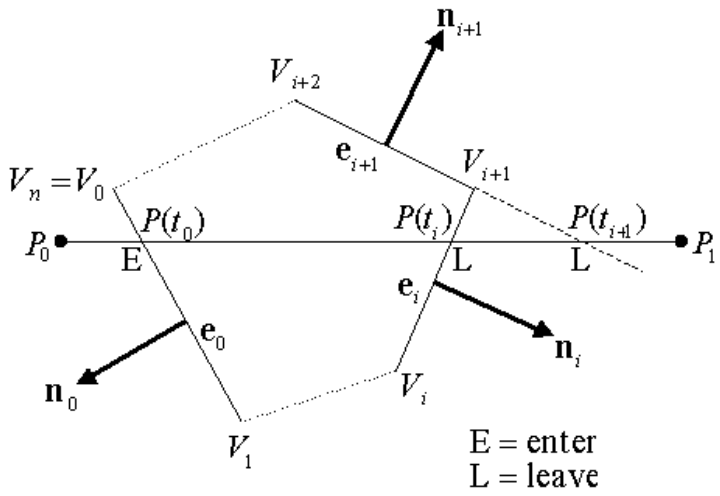
$$t_i = \frac{(V_i - P_0) \bullet \mathbf{n}_i}{(P_1 - P_0) \bullet \mathbf{n}_i}$$

at the intersection point $I_i = P(t_i)$. The denominator here is nonzero unless the segment and the edge are parallel, and we treat that as a special case. In particular, when **S** is *outside* the edge $\mathbf{e}_i$, then the segment cannot intersect the convex polygon at all, and we are done. This condition can be checked by testing if the vector from V_i to P_0 points to the outside of the edge. This is true when $(P_0 - V_i) \bullet \mathbf{n}_i > 0$, in which case processing can stop since the segment is completely outside the polygon. Otherwise, the segment **S** is *inside* the parallel edge, so we can simply ignore this edge, and continue processing the other edges.

But when they are not parallel and a unique I_i exists, then for $\mathbf{e}_i$, classify the associated t_i as ***entering*** or ***leaving*** by the criteria:

$$(P_1 - P_0) \bullet \mathbf{n}_i < 0 \Rightarrow P_0P_1 \text{ crosses } \mathbf{e}_i \text{ from outside to inside} \Rightarrow t_i \text{ is } \textit{entering}$$
$$(P_1 - P_0) \bullet \mathbf{n}_i > 0 \Rightarrow P_0P_1 \text{ crosses } \mathbf{e}_i \text{ from inside to outside} \Rightarrow t_i \text{ is } \textit{leaving}$$

as illustrated in the following diagram.



Observe that the line $P(t)$ must have crossed from the outside to the inside of all the edges where it is entering before it can be inside the whole polygon. This happens when it reaches the maximum value of all the entering t_i values. Conversely, once $P(t)$ has crossed from the inside to the outside across any edge where it is leaving, then it can never re-enter the polygon [Note: this is only true for convex polygons!]. So, once t gets larger than the minimum of all the leaving t_i values, then it is outside the polygon for good. Put:

$$t_E = \max(0, \{t_i \text{ that are } \textit{entering}\})$$
$$t_L = \min(1, \{t_i \text{ that are } \textit{leaving}\})$$

Then, the subsegment of the line inside the polygon will start at $P(t_E)$ and end at $P(t_L)$ for increasing t. But this will be non-empty only when $0 \le t_E \le t_L \le 1$. If $t_L < t_E$, the segment essentially leaves the polygon before it fully enters it, and is never inside all faces at the same time.

Pseudo-Code: Intersect Segment with a 2D Convex Polygon

The following pseudo-code efficiently encapsulates the logic of this algorithm.

```
Input: a 2D segment S from point P0 to point P1
       a 2D convex polygon Ω with n vertices V0,...,Vn-1,Vn=V0

if (P0 == P1) then S is a single point, so {
    test for point inclusion of P0 in Ω; and
    return the test result (TRUE or FALSE);
}
```

```
Initialize:
    t_E = 0 for the maximum entering segment parameter;
    t_L = 1 for the minimum leaving segment parameter;
   dS = P_1 - P_0 is the segment direction vector;

for each (edge e_i = V_iV_i+1 of Ω; i=0,n-1)
{
    Let n_i = an outward normal of the edge e_i;
    N = - dot product of (P_0-V_i) and n_i;
    D = dot product of dS and n_i;
    if (D == 0) then S is parallel to the edge e_i {
        if (N < 0) then P_0 is outside the edge e_i
             return FALSE since S cannot intersect Ω;
        else S cannot enter or leave Ω across edge e_i {
             ignore edge e_i and
             continue to process the next edge;
        }
    }

    Put t = N / D;
    if (D < 0) then segment S is entering Ω across edge e_i {
        New t_E = max of current t_E and this t
        if (t_E > t_L) then segment S enters Ω after leaving
             return FALSE since S cannot intersect Ω
    }
    else (D > 0) then segment S is leaving Ω across edge e_i {
        New t_L = min of current t_L and this t
        if (t_L < t_E) then segment S leaves Ω before entering
             return FALSE since S cannot intersect Ω
    }
}

Output: [Note: to get here, one must have t_E <= t_L]
there is a valid intersection of S with Ω
    from the entering point: P(t_E) = P_0 + t_E * dS
    to the leaving point:    P(t_L) = P_0 + t_L * dS
return TRUE
```

A C++ implementation intersect2D_SegPoly() is given below.

Intersect a Segment and a Convex Polyhedron (3D)

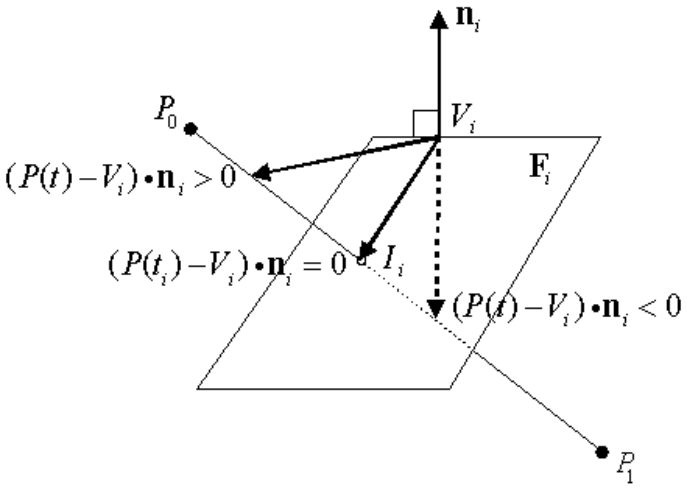
The 2D computations and algorithm extend to 3D with very few changes. The main difference concerns specifying the polyhedron data structure. We let a polyhedron Ω be given by a list of n (planar) polygon faces $\mathbf{F}_i$ for $i = 0, n-1$. We do not assume any relationship between succesive faces $\mathbf{F}_i$ and $\mathbf{F}_{i+1}$, such as having a common edge or

vertex (like adjacent edges of a polygon). It is just an unstructured list of all the distinct faces. However, to make the algorithm work, we need some extra information about each face, namely:

1. Each face $\mathbf{F}_i$ lies completely in a plane.
2. V_i = a point in the plane of face $\mathbf{F}_i$ (e.g.: a vertex of $\mathbf{F}_i$).
3. $\mathbf{n}_i$ = an *outward-pointing* normal vector to $\mathbf{F}_i$; that is, a normal vector pointing to the side of $\mathbf{F}_i$ that is exterior to the closed polyhedron Ω.

This information can be precomputed from any decent data structure for a polyhedron.

Again, the 3D line segment $\mathbf{S} = P_0P_1$ is given by a parametric equation $P(t)$. For the intersection of the extended line segment with the plane of a specific face $\mathbf{F}_i$, consider the following diagram.



This is exactly the same condition as in the 2D case; namely, the intersection point occurs when $(P(t)-V_i)\bullet\mathbf{n}_i = 0$ as shown. Again, solving this gives:

$$t_i = \frac{(V_i - P_0)\bullet\mathbf{n}_i}{(P_1 - P_0)\bullet\mathbf{n}_i}$$

When the denominator is zero, then the segment **S** is parallel to the face $\mathbf{F}_i$ which is treated exactly the same as before. Nevertheless, when t_i exists, there is a unique intersection point $I_i = P(t_i)$.

Further, since $\mathbf{n}_i$ is an *outward-pointing* normal vector to the face plane, we have **exactly the same criteria**for classifying each t_i as ***entering*** or ***leaving***; namely:

$$(P_1 - P_0)\cdot\mathbf{n}_i < 0 \Rightarrow t_i \text{ is } entering$$
$$(P_1 - P_0)\cdot\mathbf{n}_i > 0 \Rightarrow t_i \text{ is } leaving$$

And again we compute:

$$t_E = \max(0, \{t_i \text{ that are } entering\})$$
$$t_L = \min(1, \{t_i \text{ that are } leaving\})$$

Finally, the part of the segment inside the convex polyhedron is the subsegment from $P(t_E)$ to $P(t_L)$ when $0 \le t_E \le t_L \le 1$. Again, if $t_L < t_E$, then the segment is completely outside the polyhedron.

Pseudo-Code: Intersect Segment with a 3D Polyhedron

The 3D pseudo-code is almost the same as for 2D polygons, but with the obvious modifications.

```
Input: a 3D segment S from point P0 to point P1
     a 3D convex polyhedron Ω with n faces F0,...,Fn-1 and
          Vi = a vertex for each face Fi
          ni = an outward normal vector for each face Fi

if (P0 == P1) then S is a single point, so {
    test for point inclusion of P0 in Ω; and
    return the test result (TRUE or FALSE);
}

Initialize:
    tE = 0 for the maximum entering segment parameter;
    tL = 1 for the minimum leaving segment parameter;
    dS = P1 - P0 is the segment direction vector;

for (each face Fi of Ω; i=0,n-1)
{
    N = - dot product of (P0 - Vi) and ni;
    D = dot product of dS and ni;
    if (D == 0) then S is parallel to the face Fi {
        if (N < 0) then P0 is outside the face Fi
             return FALSE since S cannot intersect Ω;
        else S cannot enter or leave Ω across face Fi {
             ignore face Fi and
             continue to process the next face;
        }
    }

    Put t = N / D;
    if (D < 0) then segment S is entering Ω across face Fi {
        New tE = max of current tE and this t
```

```
            if (tE > tL) then segment S enters Ω after leaving
                 return FALSE since S cannot intersect Ω
        }
        else (D > 0) then segment S is leaving Ω across face Fi {
            New tL = min of current tL and this t
            if (tL < tE) then segment S leaves Ω before entering
                 return FALSE since S cannot intersect Ω
        }
    }

    Output: [Note: to get here, one must have tE <= tL]
    there is a valid intersection of S with Ω
        from the entering point: P(tE) = P0 + tE * dS
        to the leaving point:    P(tL) = P0 + tL * dS
    return TRUE
```

Beyond 3D, this algorithm can be easily generalized to n-dimensional space for the intersection of a parametric line with the interior of a convex polytope. The details are basically the same.

Implementation

Here is a sample "C++" implementation of this algorithm.

```
// Copyright 2001, 2012, 2021 Dan Sunday
// This code may be freely used and modified for any purpose
// providing that this copyright notice is included with it.
// There is no warranty for this code, and the author of it cannot
// be held liable for any real or imagined damage 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;}
//        operators for:
//            == to test equality
//            != to test inequality
//            Point  = Point ± Vector
//            Vector = Point - Point
//            Vector = Vector ± Vector
//            Vector = Scalar * Vector    (scalar product)
//    Segment with defining endpoints {Point P0, P1;}
//===================================================================
```

```
#define TRUE       1
#define FALSE      0
#define SMALL_NUM  0.00000001   // to avoid division overflow

#define dot(u,v)   ((u).x * (v).x + (u).y * (v).y) // dot product
#define perp(u,v)  ((u).x * (v).y - (u).y * (v).x) // perp product

// intersect2D_SegPoly(): intersect a 2D segment and convex polygon
//    Input:  S = 2D segment to intersect with the polygon
//            n = number of 2D points in the polygon
//            V[] = array of n+1 vertex points with V[n] = V[0]
//      Note: The polygon MUST be convex and
//            have vertices oriented counterclockwise (ccw).
//            This code does not check for these conditions.
//    Output: *IS = the intersection segment (when it exists)
//    Return: FALSE = no intersection
//            TRUE  = a valid intersection segment exists
int
intersect2D_SegPoly( Segment S, Point* V, int n, Segment* IS)
{
    if (S.P0 == S.P1) {   // the segment S is a single point
        // test for inclusion of S.P0 in the polygon
        int ptst = wn_PnPoly( S.P0, V, n );  // winding number test
        if (ptst != 0) {  // S.P0 is not outside the polygon
            *IS = S;      // the point S.P0 is the intersection
        }
        return ptst;
    }

    float  tE = 0;           // the max entering segment parameter
    float  tL = 1;           // the min leaving segment parameter
    float  t, N, D;          // intersect parameter t = N / D
    Vector dS = S.P1- S.P0;  // the  segment direction vector
    Vector e;                // edge vector

    for (int i=0; i < n; i++)   // process polygon edge V[i]V[i+1]
    {
        e = V[i+1] - V[i];
        N = perp(e, S.P0-V[i]);      // = -dot(ne, S.P0 - V[i])
        D = -perp(e, dS);            // = dot(ne, dS)
        if (fabs(D) < SMALL_NUM) {  // S is ~parallel to this edge
            if (N < 0)               // P0 is outside this edge, so
                 return FALSE;       // S is outside the polygon
            else                     // S cannot cross this edge, so
                 continue;           // ignore this edge
        }

        t = N / D;
        if (D < 0) {                 // segment S is entering
            if (t > tE) {            // new max tE
                 tE = t;
                 if (tE > tL)        // S enters after leaving
                     return FALSE;
```

```
            }
        }
        else {                          // segment S is leaving
            if (t < tL) {               // new min tL
                 tL = t;
                 if (tL < tE)           // S leaves before entering
                     return FALSE;
            }
        }
    }

    // tE <= tL implies there is a valid intersection subsegment
    IS->P0 = S.P0 + tE * dS;    // P(tE) = where S enters polygon
    IS->P1 = S.P0 + tL * dS;    // P(tL) = where S leaves polygon
    return TRUE;
}
```

References

M. Cyrus & J. Beck, "Generalized Two- and Three-Dimensional Clipping", Computers and Graphics 3(1), 23-28 (1978)

James Foley, Andries van Dam, Steven Feiner & John Hughes, "A Parametric Line-Clipping Algorithm" in Computer Graphics (3rd Edition) (2013)

Y-D. Liang & B. Barsky, "A New Concept and Method for Line Clipping", ACM TOG 3(1), 1-22 (1984)

Extreme Points of Convex Polygons

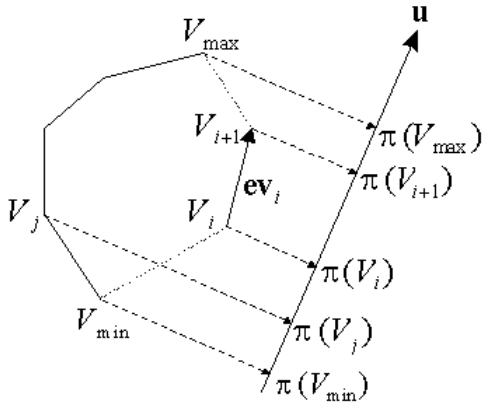
It is often useful to find an extreme point of a 2D polygon. For example, the vertices with maximal or minimal x-coordinates or y-coordinates define a polygon's bounding box. More generally, one might want to find an extreme point in an arbitrary direction. For any set of *n* points, such as an *n*-vertex polygon, this is easily done in $O(n)$ time by testing each point against the previously found extreme. But, for the special case of a convex polygon, an adaptation of binary searching can find the extreme point in only $O(\log n)$ time [O'Rourke, 1998]. We describe this algorithm in detail for an arbitrary query direction vector **u**. Also, we show how this algorithm can be applied to compute the distance from a convex polygon to a line.

Note that the convex hull of a point set or polygon is precisely the collection of extreme points in all possible directions. Thus, if several extreme point queries are expected for an arbitrary polygon, it may make sense to first compute its convex hull, and then do queries on this hull in $O(\log h)$ time, where $h \leq n$ is the number of hull vertices.

Polygon Extreme Point Algorithms

Let a 2D polygon Ω be given by *n* vertices $V_0, V_1, ..., V_{n-1}, V_n = V_0$ going counterclockwise (ccw) around the polygon. Also let $\mathbf{e}_i$ be the *i*-th edge segment from vertex V_i to V_{i+1} for $i = 0, n-1$; and $\mathbf{ev}_i = V_{i+1} - V_i$ be the edge vector.

Next, let a direction be given by a vector **u**. We want to find the extreme, maximum and minimum, vertices of Ω in the direction **u**. That is, if the vertices V_i are orthogonally projected onto a line **L** in the direction **u**, then the extreme vertices are the ones whose projections are the extreme points on **L**, as shown in the following diagram, where $\pi: \mathbb{R}^2 \rightarrow \mathbf{L}$ is the projection function.



We will say that V_i is ***above*** V_j ***relative to*** **u**, if its orthogonal projection onto **L** is further in the **u**-positive direction. That is, the vector from $\pi(V_j)$ to $\pi(V_i)$ is in the same direction as **u**. This is equivalent to the vector $(V_i - V_j)$ making an acute angle with **u**, and this corresponds to the condition that: $\mathbf{u} \bullet (V_i - V_j) > 0$. So, this test can determine if an edge $\mathbf{e}_i$ of Ω is increasing or decreasing relative to **u**; namely, it is increasing if its edge vector $\mathbf{ev}_i$ satisfies: $\mathbf{u} \bullet \mathbf{ev}_i > 0$, and decreasing when: $\mathbf{u} \bullet \mathbf{ev}_i < 0$.

Brute-Force Search

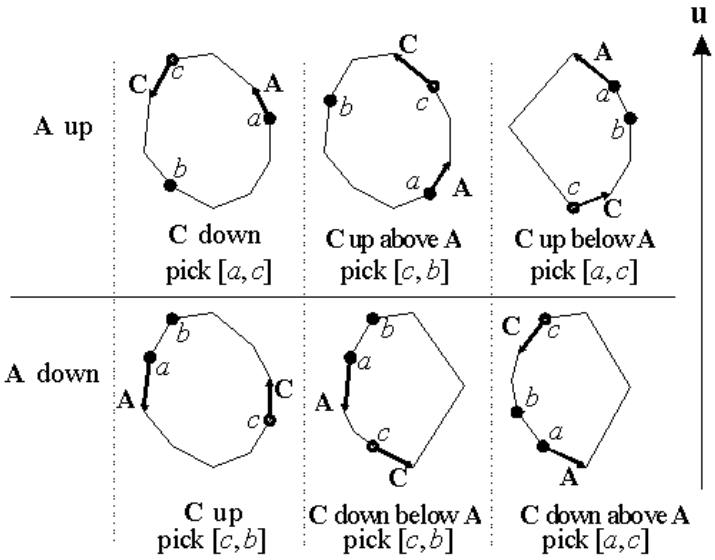
The straightforward brute-force way to find an extreme point is to test all points incrementally, and remember the current extremes, max and min, for the points tested so far. Each new point considered only has to be compared to the current extreme ones. This works for any set of *n* points, and is clearly an $O(n)$ algorithm. The algorithm for finding the extreme maximum and minimum relative to **u** is simple, as shown in the following pseudo-code.

```
Input: W = {V0,V1,...,Vn-1}  is a set of n points
       u = a direction vector

Put: max = min = 0, for V0
for each point Vi in {V1,...,Vn-1}  (i=1,n-1)
{
    if (u · (Vi - Vmax) > 0) {  // Vi is above the prior max
        max = i;                 // new max index = i
        continue;                // a new max can't be a new min
    }
    if (u · (Vi - Vmin) < 0)    // Vi is below the prior min
        min = i;                 // new min index = i
}
return indexes of the max and min vertices
```

For an arbitrary set of points without structure, this is the best one can do. For a set of *n* points, this algorithm will perform at most 2 (n−1) dot product comparisons to compute the extreme points.

Binary Search

However, if the point set is the ordered set of vertices for a convex polygon, then we can find the maximum (or similarly, the minimum) relative to **u** more quickly, in $O(\log n)$ time, with a binary search. But the algorithm is no longer simple. Say we know that the maximum vertex lies on the part of the polygon starting at vertex V_a and ending at vertex V_b. This "chain" is given by the index range $[a,b] = \{a, a+1, ..., a+k = b \pmod n\}$, where $k > 0$ is the first k for which the final equality holds. We next pick a vertex midway between V_a and V_b, call it V_c. To do the binary search, we need a simple test that will restrict our search to one of the two subchains [*a*,*c*] or [*c*,*b*]. Because the polygon is convex, we can easily determine which one to pick by comparing the edge vectors $\mathbf{A} = \mathbf{ev}_a$ at V_a and $\mathbf{C} = \mathbf{ev}_c$ at V_c. Six cases can occur, three with **A** up and three with **A** down, as shown in the following diagram.



For each case, one picks the segment, either [*a*,*c*] or [*c*,*b*], that would contain the maximum. These are the only cases that can occur because the polygon Ω is monotonic in the direction **u**. That is, it consists of exactly two monotone polyline segments, one increasing and one decreasing relative to the direction **u**, that join the max and min vertices. Since a convex polygon is monotonic in all directions, this is true no matter which direction **u** is selected. The converse is also true; namely, a polygon that is monotonic in all directions must be convex.

We can now construct the whole algorithm by starting with the chain $[a=0, b=n]$, which is the whole polygon. As long as $b > a+1$, we will always have that:
$0 \le a < c = \mathbf{int}((a+b)/2) < b \le n$, and do not have to do any additional (mod *n*) computations. Then, at each step, after testing V_c and determining which case we have, we either increase *a* to *c* or decrease *b* to *c*, and get a new range [*a*,*b*] that is half as large as the former one. Of course, we do have to check whether we have found a local maximum by testing each new subdivision vertex V_c. This is done by checking if V_c is greater than or equal to it's two neighboring vertices relative to **u**. This test works as long as the two neighbors do not both have equal heights with V_c. However, if that were true, then V_c would not be an essential ***proper vertex*** of Ω; instead, it would be on the interior of the segment from V_{c-1} to V_{c+1}, and so could be removed without changing Ω's boundary. A convex polygon with only proper vertices is called a ***proper convex*** polygon. For any proper convex polygon, a local maximum must also be a global maximum, and we are done when we find one.

Pseudo-Code: Binary Search

Putting this together, the complete algorithm to find a **u**-maximum vertex is given by the pseudo-code:

```
Input: Ω = {V0,V1,...,Vn-1,Vn=V0} is a 2D proper convex polygon
       u = a 2D direction vector
```

```
if V0 is a local maximum, then return 0;
Put a=0; b=n;
Put A = the edge vector at V0;
forever {
    Put c = the midpoint of [a,b] = int((a+b)/2);
    if Vc is a local maximum, then it is the global maximum
        return c;

    // no max yet, so continue with the binary search
    // pick one of the two subchains [a,c] or [c,b]
    Put C = the edge vector at Vc;
    if (A points up) {
        if (C points down) {
            b = c;                      select [a,c]
        }
        else C points up {
            if Va is above Vc {
                 b = c;                  select [a,c]
            }
            else Va is below Vc {
                 a = c;                  select [c,b]
                 A = C;
            }
        }
    }
    else A points down {
        if (C points up) {
            a = c;                      select [c,b]
            A = C;
        }
        else C points down {
            if Va is below Vc {
                 b = c;                  select [a,c]
            }
            else Va is above Vc {
                 a = c;                  select [c,b]
                 A = C;
            }
        }
    }
}
```

A C++ implementation is given below in the routine polyMax_2D(). As implemented, we use one dot product comparison to test if a vector is up or down, and two dot product tests for a local maximum. For a polygon with n vertices, the algorithm uses 2.5 such tests on average for each loop iteration, and thus has $2.5\log(n)$ dot product comparisons in total. For small n, this is less efficient than the brute force algorithm when $(n-1) < 3\log(n)$. The break even point is at $n = 9$, where both methods use 8 comparisons (although the binary search may terminate faster when a maximum is found early). Since many frequently occurring convex polygons have fewer than 10 vertices, one should use the simpler brute force method for small polygons.

Distance of a Polygon to a Line

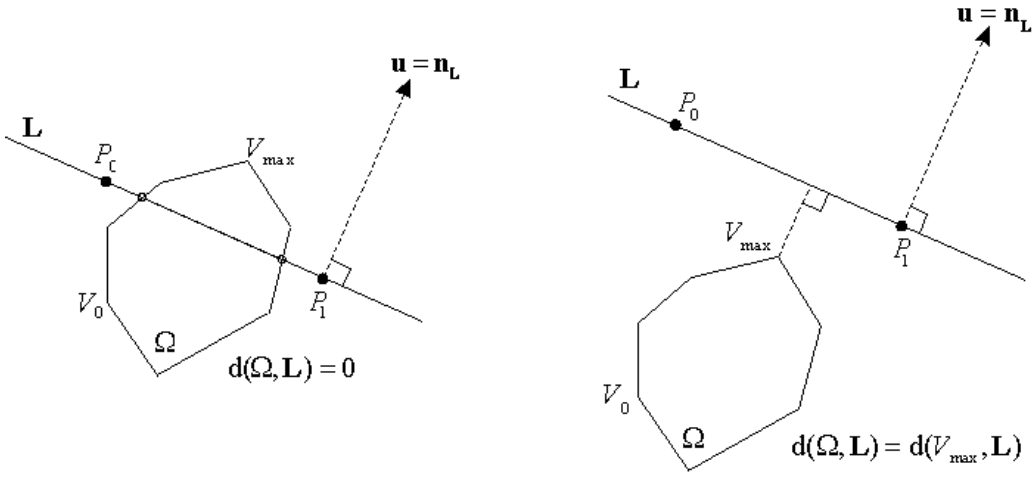
We can use the above algorithm to quickly find the minimum distance from a proper convex polygon $\Omega = \{V_i\}$ to a line **L** through two points P_0 and P_1. Denote this distance by $\mathbf{d(\Omega,L)}$. If the polygon is not convex, we should compute and use its convex hull if we expect to compute the distance from it to many different lines at different orientations.

The idea here is simple. First, pick **u** to be perpendicular normal vector to the line **L** so that at least one vertex of Ω is on the side of **L** not pointed to by **u**. More precisely, a point P is on the side of **L** pointed to by **u**, called the **u**-positive side of **L**, if $\mathbf{u} \bullet (P - P_0) > 0$. Otherwise it is on the **u**-backside of **L**. To start, we get a normal vector **n** for **L** by using the perp-operator (see Vector Products Perp Operator) to select $\mathbf{n} = (P_1 - P_0)^{\perp}$. If $\mathbf{n} \bullet (V_0 - P_0) \leq 0$, V_0 is on the **n**-backside of **L**, and we put **u** = **n**. If not, we simply reverse the normal vector and put **u** = –**n**. In either case, **u** is normal to **L,** and V_0 is on the **u**-backside.

Next, we find the maximum vertex $V_{\max}$ of Ω relative to this normal vector **u**. There are two cases:

1. $V_{\max}$ is on the **u**-positive side of L, and thus any polyline chain from V_0 to $V_{\max}$ must cross L. This means that $\mathbf{d(\Omega,L)} = 0$, since they have a common intersection point.
2. $V_{\max}$ is on the **u**-backside of Ω, and thus all vertices are on the **u**-backside. Then, $V_{\max}$ is the closest vertex to **L**, and $\mathbf{d(\Omega,L)} = \mathbf{d}(V_{\max},\mathbf{L})$, the distance from $V_{\max}$ to the line.

These cases are shown in the following diagram.



C++ code for this algorithm is given below in the routine `dist2D_Poly_to_Line()`.

Implementations

Here are some sample "C++" implementations of these algorithms.

```
// Copyright 2001, 2012, 2021 Dan Sunday
// This code may be freely used and modified for any purpose
// providing that this copyright notice is included with it.
// There is no warranty for this code, and the author of it cannot
// be held liable for any real or imagined damage 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 (2D) with:
//        coordinates {float x, y;}
//        operators for:
//            == to test equality
//            != to test inequality
//            =  for assignment
//            -Vector for unary minus
//            Point  = Point ± Vector
//            Vector = Point - Point
//            Vector = Vector ± Vector
//    Line with defining points {Point P0, P1;}
//===================================================================


// 2D dot product which allows vector operations in arguments
#define dot(u,v)   ((u).x * (v).x + (u).y * (v).y)

// tests for vector orientation relative to a direction vector u
#define up(u,v)         (dot(u,v) > 0)
#define down(u,v)       (dot(u,v) < 0)
#define dr(u,Vi,Vj)     (dot(u, (Vi)-(Vj))) // direction sign
#define above(u,Vi,Vj)  (dr(u,Vi,Vj) > 0)   // Vi is above Vj
#define below(u,Vi,Vj)  (dr(u,Vi,Vj) < 0)   // Vi is below Vj


// polyMax_2D(): find polygon max vertex in specified direction
//    Input:  U   = a 2D direction vector
//            V[] = array vertices of a proper convex polygon
//            n   = number of polygon vertices, with V[n]=V[0]
//    Return: index (>=0) of the maximum vertex, or
//            (-1) = an error [Note: should be impossible, but...]
int
polyMax_2D( Vector U, Point* V, int n )
{
    if (n < 10) {           // use brute force for small polygons
        int max = 0;
        for (int i=1; i<n; i++) // for each point in {V1,...,Vn-1}
            if (above(U, V[i], V[max]))  // V[i] is above V[max]
                max = i;                 // new max index = i
        return max;
```

```
}

// use binary search for large polygons
int     a, b, c;            // indices for edge chain endpoints
Vector  A, C;               // edge vectors at V[a] and V[c]
int     upA, upC;           // test for "up" direction of A and C

a=0; b=n;                   // start chain = [0,n] with V[n]=V[0]
A = V[1] - V[0];
upA = up(U,A);
// test if V[0] is a local maximum
if (!upA && !above(U, V[n-1], V[0]))    //  V[0] is the maximum
    return 0;

for(;;) {
    c = (a + b) / 2;        // midpoint of [a,b], and 0<c<n
    C = V[c+1] - V[c];
    upC = up(U,C);
    if (!upC && !above(U, V[c-1], V[c])) // V[c] is a local max
        return c;                        // thus it is the max

    // no max yet, so continue with the  binary search
    // pick one of the two subchains [a,c]  or [c,b]
    if (upA) {                      // A points up
        if (!upC) {                     // C points down
            b = c;                      // select [a,c]
        }
        else {                          // C points up
            if (above(U, V[a], V[c])) {     // V[a] above V[c]
                b = c;                      // select [a,c]
            }
            else {                          // V[a] below V[c]
                a = c;                      // select [c,b]
                A = C;
                upA = upC;
            }
        }
    }
    else {                          // A points down
        if (upC) {                      // C points up
            a = c;                      // select [c,b]
            A = C;
            upA = upC;
        }
        else {                          // C points down
            if (below(U, V[a], V[c])) {     // V[a] below V[c]
                b = c;                      // select [a,c]
            }
            else {                          // V[a] above V[c]
                a = c;                      // select [c,b]
                A = C;
                upA = upC;
            }
        }
    }
    // have a new (reduced) chain [a,b]
```

```
    }
}
//===================================================================


// dist2D_Poly_to_Line(): find the distance from a polygon to a line
//    Input:  V[] = array vertices of a proper convex polygon
//            n    = the number of polygon vertices, with V[n] = V[0]
//            L    = a Line (defined by 2 points P0 and P1)
//    Return: minimum distance from V[] to L
float
dist2D_Poly_to_Line( Point* V, int n, Line L )
{
    Vector    U, N;
    int       max;

    // get a leftward normal N to L
    N.x = -(L.P1.y - L.P0.y);
    N.y =  (L.P1.x - L.P0.x);
    // get a normal U to L with V[0] on U-backside
    if (dot(N, V[0] - L.P0) <= 0)
        U = N;
    else U = -N;

    max = polyMax_2D( U, V, n );          // max vertex in U direction

    if (dot(U, V[max] - L.P0) > 0)        // V[max] on U-positive side
        return 0;                         // polygon and line intersect
    else
        return dist_Point_to_Line( V[max], L);  // min dist to line L
}
//===================================================================
```

References

Joseph O'Rourke, Computational Geometry in C (2nd Edition), Sect 7.9 "Extreme Point of Convex Polygon" (1998)

Polygon Tangents

Constructing tangents is a fundamental geometric operation. The most efficient algorithms for constructing tangents assume they are to convex objects. However, the tangents for a nonconvex object are the same as the tangents to its convex hull, which can generally be efficiently computed. This month we consider tangents to convex polygons. We discuss two cases:

1. Tangents from a point to a convex polygon.
2. Tangents between two disjoint convex polygons.

Finding polygon tangents can be viewed as finding extreme vertices of the polygon. We previously showed how to find the maximum and minimum points of a convex polygon in a given direction. For tangents, we want to find the maximum and minimum in a pencil of lines. By applying similar binary search methods, we can construct fast, $O(\log n)$ time, algorithms to find tangents.

Tangents: Point to Polygon

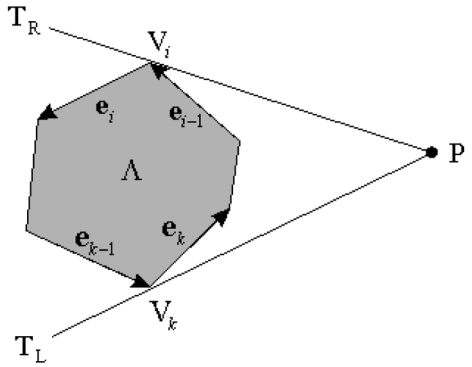
> A straight line is said to touch a circle which, meeting the circle and being produced, does not cut the circle. [Book III, Definition 2]
>
> From a given point to draw a straight line touching a given circle. [Book III, Proposition 17]
> [Euclid, 300 BC]

A 2D line **L** is tangent to a polygon Ω when **L** touches Ω it without crossing its boundary. If **L** is an infinite line, then it must be external to Ω, and intersect the polygon in either an isolated point (a vertex of Ω) or a line segment (an edge of Ω). In both cases, the tangent goes through a vertex of the polygon, and Ω must lie completely on one side of **L**. Further, there are exactly two tangents from an exterior point to a nondegenerate polygon.

Let the 2D polygon Ω be given by n vertices $V_0, V_1, \ldots, V_{n-1}, V_n = V_0$ going counterclockwise (ccw) around the polygon. Also let $\mathbf{e}_i$ be the i-th edge segment from vertex V_i to V_{i+1} for i=0,n–1; and $\mathbf{ev}_i = V_{i+1} - V_i$ be the edge vector. We say that point P is inside $\mathbf{e}_i$ if P is left of the line through $\mathbf{e}_i$, otherwise P is outside $\mathbf{e}_i$.

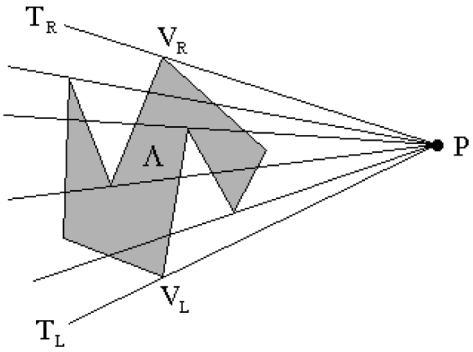
Next, let P be an arbitrary point external to Ω. Then, for a vertex V_i of Ω, there is an easy test to determine whether the line from P to V_i is a tangent. Consider the two edges, $\mathbf{e}_{i-1}$ and $\mathbf{e}_i$, meeting at V_i. If P is inside both of these edges, or outside both

edges, then those two edges are on different sides of PV_i, and thus PV_icannot be a tangent. On the other hand, if P is inside one edge and outside the other, then PV_i is locally tangent to Ω at V_i. The two possible cases are shown in the following diagram. For vertex V_i, the point P is outside (to the right of) edge $\mathbf{e}_{i-1}$ and inside (to the left of) edge $\mathbf{e}_i$. Thus, V_i gives a rightmost tangent $\mathbf{T}_R$from P to Ω. Similarly, for V_k, P is inside edge $\mathbf{e}_{k-1}$ and outside edge $\mathbf{e}_k$; and thus V_k is a leftmost tangent $\mathbf{T}_L$ from P to Ω.



For a convex polygon, as shown, there is exactly one instance of each case, corresponding with the two unique tangents. For a nonconvex polygon, one could first compute the convex hull, and then find the hull's tangents which will also be tangent to the polygon. An advantage of doing this is that the hull generally has significantly fewer vertices than the original polygon. So, if many tangents will be computed to the same polygon, using its convex hull makes sense. In many applications, the hull can computed quickly since the convex hull of a simple polygon can be computed in $O(n)$ time using Melkman's fast algorithm.

Alternatively, using a brute-force algorithm, one could find all local tangents to the original polygon and then use the extreme rightmost and leftmost tangents, $\mathbf{T}_R$ and $\mathbf{T}_L$ at extreme right and left vertices V_R and $V_L$respectively. This algorithm also takes $O(n)$ time.



Brute-Force Algorithm

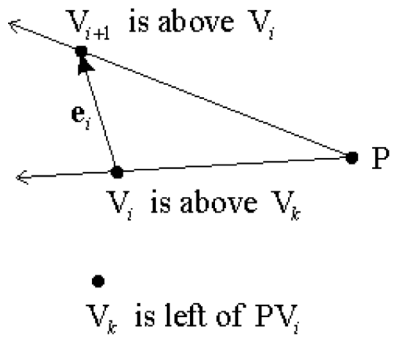
The brute-force algorithm for finding the tangents to an arbitrary polygon just tests the edge conditions at every vertex of Ω, and thus discovers the two vertices, the rightmost V_R and the leftmost V_L, giving the tangents from P to Ω. This is an $O(n)$ algorithm that is easy to implement, is usually adequate when n is relatively small, and works for nonconvex polygons. Here is pseudo-code that works for any polygon:

```
Input: Ω = {V0,V1,...,Vn-1,Vn=V0}  is any 2D polygon
       P = a 2D point

VR = VL = V0;
for each vertex Vi (i=1,n-1)
{
    eprev = (P is left of Vi-1Vi);     // left of edge ei-1
    enext = (P is left of ViVi+1);     // left of edge ei
    if ((NOT eprev) and enext) {
        if (Vi is not below VR)      // handles nonconvex case
            VR = Vi;
    }
    else if (eprev and (NOT enext)) {
        if (Vi is not above VL)      // handles nonconvex case
            VL = Vi;
    }
}

return tangent points VR and VL
```

An efficient implementation of this generally useful algorithm is given below in tangent_PointPoly(). Note that if the polygon Ω is known to be convex, then the nonconvex tests indicated can be omitted. However, for convex polygons it is usually better to use a binary search algorithm.

Binary Search Algorithm

For a convex polygon, a faster algorithm to find its tangents uses a binary search on the vertices, the same as the algorithm to find the extreme points. This algorithm simply defines an ordering on the polygon's vertices so that the two tangent vertices correspond to the maximum and minimum for the ordering. The natural order is based on lines going through the point P. We say that a vertex V_i is ***above*** V_k relative to P, if V_k is left of the line from P to V_i, as illustrated in the diagram:



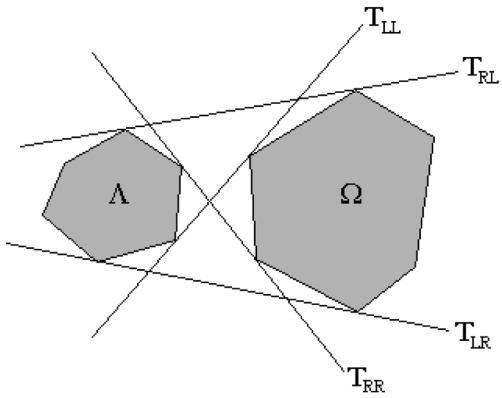
Additionally, an edge $\mathbf{e}_i$ of Ω is ***increasing*** relative to P if V_{i+1} is above V_i for this ordering. This is equivalent to the condition that P is right of, and thus is outside, the edge $\mathbf{e}_i$. Conversely, $\mathbf{e}_i$ is ***decreasing*** relative to P if V_{i+1} is below V_i, which is equivalent to P being inside the edge $\mathbf{e}_i$. With these definitions, Algorithm 14 can be directly transcribed to produce a binary search for tangents in $O(\log n)$ time.

Pseudo-code for an algorithm that selects the maximum tangent using this new ordering is exactly the same as for the extreme point binary search pseudo-code. The only difference in the implementation is using a different test for the new order relation. With this algorithm, instead of finding the two tangents simultaneously, the rightmost and leftmost tangents are found separately by running the algorithm twice. Despite this overhead, the convex binary search is more efficient even for very small polygons. Timing tests with our implementation show the break even point to be n=4, a quadrilateral. So, it is generally best to use the binary search algorithm. Our implementation is given below in `tangent_PointPolyC()`.

Tangents: Polygon to Polygon

Next, we will find common tangents between two distinct polygons: Λ with m vertices = $\{V_i\}$ (i=0, m) , and Ω with n vertices = $\{W_k\}$ (k=0, n). Finding these tangents is similar to finding the point-to-polygon tangents, although the algorithm is somewhat more complicated because:

1. One must scan the vertices of two distinct polygons in some synchronous manner.
2. For two disjoint convex polygons, there are 4 distinct tangents, as illustrated.



There are fast polygon-to-polygon tangency algorithms when both polygons are convex. For nonconvex polygons one could use a slow brute-force algorithm; or instead, replace the polygons with their convex hulls. If the polygons are simple, the latter alternative is faster since the two convex hulls can be computed in $O(m)$ and $O(n)$ time using Melkman's algorithm, and then the tangents between them can be found in $O(m+n)$ or even $O(\log(m+n))$ time.

Brute-Force Algorithm

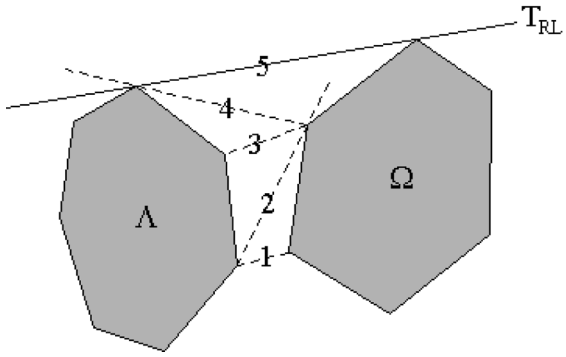
The simple brute-force tangency algorithm tests every line that joins a vertex of Λ to a vertex of Ω for tangency with both polygons. Since there are *mn* vertex pairs, there are *mn* such lines to test, and this gives an $O(mn)$ quadratic time algorithm. Although this is slow, it does work for any two polygons whether or not they are convex. However, it is faster and easier to just compute and use the convex hulls of the two polygons.

If even one of the polygons, say Ω, is convex, then this algorithm can be easily improved to $O(m \log n)$ time. For each of the *m* vertices of Λ, the two tangents from it to Ω are found using an $O(\log n)$ point-to-convex_polygon algorithm. For each *m*, there are only two lines to test for tangency to Λ. The result is an $O(m \log n)$ algorithm that might be useful for some applications.

Linear Search Algorithm

But, if both polygons are convex, then there is a straightforward linear $O(m+n)$ tangency algorithm. The idea of this algorithm is to alternate searching for tangent endpoints between the two polygons, Λ and Ω, until both ends of the line segment simultaneous satisfy the tangency condition. This algorithm was originally described by [Preparata & Hong, 1977] as part of the convex hull divide-and-conquer algorithm, and is presented by [Boissonnat & Yvinec, 1998] and [O'Rourke, 1998].

The algorithm starts by finding vertices on each polygon that can clearly see the other polygon; that is, the tangents from each point to the other polygon are exterior to both polygons. Then, the endpoints of the line segment joining the vertices are sequentially changed. First one endpoint is sequentially advanced on its polygon until the line segment becomes tangent to that polygon. Then, the endpoint on the other polygon is advanced until the line is tangent to the second polygon. Next, one goes back to the first polygon, and continues with this procedure until the line joining the vertices becomes tangent to both polygons at the same time. The direction in which the vertices are changed determines which of the four tangents will be found. The direction can either be clockwise or counterclockwise on each of the two polygons, and this gives four cases that are associated with the four tangents. For example, in the following diagram, starting with line #1, the algorithm advances clockwise on Ω and counterclockwise on Λ, and . This results in finding line #5 as the tangent T_{RL} which is rightmost on polygon Λ and leftmost on Ω. After initialization, the vertices chosen are always advancing in the same direction, either increasing or decreasing. Thus, there is no backtracking, and there are at most (m+n) vertex pairs and lines to test for tangency, so this is an $O(m+n)$ linear algorithm.



One complication with this algorithm is the need to select two initial vertices, one each from Λ and Ω, that can clearly see the other polygon. This can be done, in $O(\log(mn))$ time, by selecting vertices that are closest to a separating line between Λ and Ω. However, another alternative is to use the binary search point-to-convex polygon algorithm. First, select any point from Λ, say V_0, and find its upper (or lower) tangent to Ω, say at vertex W_{k0}. Then, from this vertex, find the upper (or lower) tangent to Λ, say at V_{i0}. These two vertices V_{i0} and W_{k0} are very good initial points for the linear search algorithm. Note that, like [Kirkpatrick & Snoeyink, 1995], this approach does not require finding a separating line between Λ and Ω.

An implementation of this algorithm is given below as `RLtangent_PolyPolyC()` which finds the Right-Left tangent from Λ to Ω. By interchanging the polygons when calling this function, it also finds the Left-Right tangent. A slightly modified routine will also find the Right-Right and Left-Left tangents.

Binary Search Algorithms

There are even faster sublinear time algorithms that do synchronized binary searches on the two polygons. A nested binary search on the two polygons results in a $O(\log(m)\log(n))$ algorithm. Further, both [Overmars & van Leeuwen, 1981] and [Kirkpatrick & Snoeyink, 1995] have described $O(\log(m+n))$ algorithms that find the outer tangents between two convex polygons. The [Kirkpatrick & Snoeyink, 1995] (KS) algorithm is particularly interesting, using a "tentative prune-and-search" technique of theirs. Their paper, which is downloadable from Snoeyink's web site, both describes and analyzes this algorithm in detail, as well as giving a C code implementation in an appendix.

Implementation

Here are some sample "C++" implementations of these algorithms.

```
// Copyright 2001, 2012, 2021 Dan Sunday
// This code may be freely used and modified for any purpose
// providing that this copyright notice is included with it.
// There is no warranty for this code, and the author of it cannot
// be held liable for any real or imagined damage from its use.
// Users of this code must verify correctness for their application.


// Assume that classes are already given for the objects:
//    Point with coordinates {float x, y;}
//===================================================================

// isLeft(): test if a point is Left|On|Right of an infinite line.
//    Input:  three points P0, P1, and P2
//    Return: >0 for P2 left of the line through P0 and P1
//            =0 for P2 on the line
//            <0 for P2 right of the line
inline float
isLeft( Point P0, Point P1, Point P2 )
{
    return (P1.x - P0.x)*(P2.y - P0.y)
         - (P2.x - P0.x)*(P1.y - P0.y);
}

// tests for polygon vertex ordering relative to a fixed point P
#define above(P,Vi,Vj)  (isLeft(P,Vi,Vj) > 0)   // Vi is above Vj
#define below(P,Vi,Vj)  (isLeft(P,Vi,Vj) < 0)   // Vi is below Vj
//===================================================================

// tangent_PointPoly(): find any polygon's exterior tangents
//    Input:  P = a 2D point (exterior to the polygon)
```

```
//            n = number of polygon vertices
//            V = array of vertices for polygon with V[n]=V[0]
//    Output: *rtan = index of rightmost tangent point V[*rtan]
//            *ltan = index of leftmost tangent point V[*ltan]
void
tangent_PointPoly( Point P, int n, Point* V, int* rtan, int* ltan )
{
    float  eprev, enext;    // V[i] previous and next edge turns

    *rtan = *ltan = 0;      // initially assume V[0] = both tangents
    eprev = isLeft(V[0], V[1], P);
    for (int i=1; i<n; i++) {
        enext = isLeft(V[i], V[i+1], P);
        if ((eprev <= 0) && (enext > 0)) {
            if (!below(P, V[i], V[*rtan]))
                *rtan = i;
        }
        else if ((eprev > 0) && (enext <= 0)) {
            if (!above(P, V[i], V[*ltan]))
                *ltan = i;
        }
        eprev = enext;
    }
    return;
}
//===================================================================

// tangent_PointPolyC(): binary search for convex polygon tangents
//    Input:  P = a 2D point (exterior to the polygon)
//            n = number of polygon vertices
//            V = array of vertices for a polygon with V[n] = V[0]
//    Output: *rtan = index of rightmost tangent point V[*rtan]
//            *ltan = index of leftmost tangent point V[*ltan]
void
tangent_PointPolyC( Point P, int n, Point* V, int* rtan, int* ltan )
{
    *rtan = Rtangent_PointPolyC(P, n, V);
    *ltan = Ltangent_PointPolyC(P, n, V);
}

// Rtangent_PointPolyC(): binary search for right tangent to polygon
//    Input:  P = a 2D point (exterior to the polygon)
//            n = number of polygon vertices
//            V = array of vertices for polygon with V[n] = V[0]
//    Return: index "i" of rightmost tangent point V[i]
int
Rtangent_PointPolyC( Point P, int n, Point* V )
{
    // use binary search for large convex polygons
    int     a, b, c;        // indices for edge chain endpoints
    int     upA, dnC;       // test up direction of edges a and c

    // rightmost tangent = maximum for the isLeft() ordering
```

```
    // test if V[0] is a local maximum
    if (below(P, V[1], V[0]) && !above(P, V[n-1], V[0]))
        return 0;           // V[0] is the maximum tangent point

    for (a=0, b=n;;) {      // start chain = [0,n] with V[n]=V[0]
        c = (a + b) / 2;    // midpoint of [a,b], and 0<c<n
        dnC = below(P, V[c+1], V[c]);
        if (dnC && !above(P, V[c-1], V[c]))
            return c;       // V[c] is the maximum tangent point

        // no max yet, so continue with the binary search
        // pick one of the two subchains [a,c] or [c,b]
        upA = above(P, V[a+1], V[a]);
        if (upA) {                          // edge a points up
            if (dnC)                            // edge c points down
                b = c;                              // select [a,c]
            else {                              // edge c points up
                if (above(P, V[a], V[c]))       // V[a] above V[c]
                    b = c;                          // select [a,c]
                else                            // V[a] below V[c]
                    a = c;                          // select [c,b]
            }
        }
        else {                              // edge a points down
            if (!dnC)                           // edge c points up
                a = c;                              // select [c,b]
            else {                              // edge c points down
                if (below(P, V[a], V[c]))       // V[a] below V[c]
                    b = c;                          // select [a,c]
                else                            // V[a] above V[c]
                    a = c;                          // select [c,b]
            }
        }
    }
}


// Ltangent_PointPolyC(): binary search for left tangent to polygon
//    Input:  P = a 2D point (exterior to the polygon)
//            n = number of polygon vertices
//            V = array of vertices for polygon with V[n]=V[0]
//    Return: index "i" of leftmost tangent point V[i]
int
Ltangent_PointPolyC( Point P, int n, Point* V )
{
    // use binary search for large convex polygons
    int     a, b, c;        // indices for edge chain endpoints
    int     dnA, dnC;       // test down direction of edges a and c

    // leftmost tangent = minimum for the isLeft() ordering
    // test if V[0] is a local minimum
    if (above(P, V[n-1], V[0]) && !below(P, V[1], V[0]))
        return 0;           // V[0] is the minimum tangent point
```

```
    for (a=0, b=n;;) {      // start chain = [0,n] with V[n] = V[0]
        c = (a + b) / 2;    // midpoint of [a,b], and 0<c<n
        dnC = below(P, V[c+1], V[c]);
        if (above(P, V[c-1], V[c]) && !dnC)
            return c;       // V[c] is the minimum tangent point

        // no min yet, so continue with the binary search
        // pick one of the two subchains [a,c] or [c,b]
        dnA = below(P, V[a+1], V[a]);
        if (dnA) {                          // edge a points down
            if (!dnC)                           // edge c points up
                 b = c;                             // select [a,c]
            else {                              // edge c points down
                 if (below(P, V[a], V[c]))      // V[a] below V[c]
                     b = c;                         // select [a,c]
                 else                           // V[a] above V[c]
                     a = c;                         // select [c,b]
            }
        }
        else {                              // edge a points up
            if (dnC)                            // edge c points down
                 a = c;                             // select [c,b]
            else {                              // edge c points up
                 if (above(P, V[a], V[c]))      // V[a] above V[c]
                     b = c;                         // select [a,c]
                 else                           // V[a] below V[c]
                     a = c;                         // select [c,b]
            }
        }
    }
}
//===================================================================

// RLtangent_PolyPolyC(): get the RL tangent for two convex polygons
//    Input:  m = number of vertices in polygon 1
//            V = array of vertices for polygon 1 with V[m]=V[0]
//            n = number of vertices in polygon 2
//            W = array of vertices for polygon 2 with W[n]=W[0]
//    Output: *t1 = index of tangent point V[t1] for polygon 1
//            *t2 = index of tangent point W[t2] for polygon 2
void
RLtangent_PolyPolyC( int m, Point* V, int n, Point* W, int* t1, int*
t2)
{
    int ix1, ix2;      // search indices for polygons 1 and 2

    // first get the initial vertex on each polygon
    ix1 = Rtangent_PointPolyC(W[0], m, V);   // Rtangent W[0] to V
    ix2 = Ltangent_PointPolyC(V[ix1], n, W); // Ltangent V[ix1] to W

    // ping-pong linear search until it stabilizes
    int done = FALSE;            // flag when done
```

```
    while (done == FALSE) {
        done = TRUE;              // assume done until...
        while (isLeft(W[ix2], V[ix1], V[ix1+1]) <= 0){
            ++ix1;                // get Rtangent from W[ix2] to V
        }
        while (isLeft(V[ix1], W[ix2], W[ix2-1]) >= 0){
            --ix2;                // get Ltangent from V[ix1] to W
            done = FALSE;         // not done if had to adjust this
        }
    }
    *t1 = ix1;
    *t2 = ix2;
    return;
}
//=====================================================================
```

References

Jean-Daniel Boissonnat & Mariette Yvinec, Algorithmic Geometry, Chap 9 "Convex Hulls" (1998)

Euclid, The Elements (300 BC)

Thomas Heath, The Thirteen Books of Euclid's Elements, Vol 2 (Books III-IX) (1956)

David Kirkpatrick & Jack Snoeyink, "Computing Common Tangents without a Separating Line", Workshop on Algorithms and Data Structures, 183-193 (1995) [downloadable with C code from http://www.cs.ubc.ca/spider/snoeyink/papers/nosep.ps.gz]

Joseph O'Rourke, Computational Geometry in C (2nd Edition), Sects 3.7-3.8, 88-95 (1998)

Mark Overmars & J. van Leeuwen, "Maintenance of configurations in the plane", J. Comput. Sys. Sci. 23, 166-204 (1981)

Franco Preparata & S.J. Hong, "Convex Hulls of Finite Sets of Points in Two and Three Dimensions", Comm ACM 20, 87-93 (1977)

Polyline Decimation

Often a polyline has too much resolution for an application, such as visual displays of geographic map boundaries or detailed animated figures in games or movies. That is, the points on the polylines representing the object boundaries are too close together for the resolution of the application. For example, in a computer display, successive vertexes of the polyline may be displayed at the same screen pixel so that successive edge segments start, stay at, and end at the same displayed point. The whole polyline may even have all its vertexes mapped to the same pixel, so that it appears simply as a single point in the display!

For the sake of efficiency, one doesn't want to draw all these degenerate lines, since drawing a single point would be enough. To achieve this, one wants to reduce the vertexes and edges of the polyline to essential ones that suffice for the resolution of one's application. There are several algorithms for doing this. In effect, these algorithms approximate a high resolution polyline with a low resolution reduced polyline having fewer vertices. This also speeds up subsequent algorithms, such as area fill or intersection, that may applied to the reduced polyline or polygon.

Overview

We will consider several different algorithms for reducing the points in a polyline to produce a “decimated” polyline that approximates the original within a specified tolerance. Most of these algorithms work in any dimension since they only depend on computing the distance between points and lines. Thus, they can be applied to arbitrary 2D or 3D curves that have been approximated by a polyline, for example, by sampling a parametric curve at regular small intervals.

The first decimation algorithm, vertex reduction (VR), is a fast $O(n)$ algorithm. It is the fastest and least complicated algorithm, but gives the coarsest result. However, it can be used as a preprocessing stage before applying other algorithms. This results in an faster combined algorithm since vertex reduction can significantly decrease the number of vertexes that remain for input to other decimation algorithms.

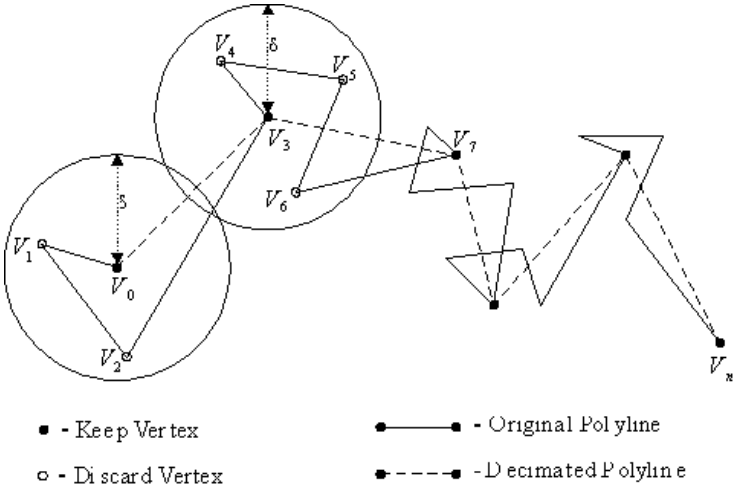
The second algorithm is the classical Douglas-Peucker (DP) approximation algorithm that is used extensively for both computer graphics and geographic information systems. There are two variants of this algorithm, the original $O(n^2)$ method [Douglas & Peucker, 1973] and a more recent $O(n \log n)$ one [Hershberger & Snoeyink, 1992]. Unfortunately, as is often the case, the faster algorithm is more complicated to implement. Additionally, it is not as general, and only works for simple 2D planar polylines, but not in higher dimensions.

Finally, we combine these algorithms, VR followed by DP, in our C++ code implementation of `poly_decimate()` as a fast practical high-quality polyline approximation algorithm.

Vertex Cluster Reduction

In vertex cluster reduction, successive vertexes that are clustered too closely are reduced to a single vertex. For example, if a polyline is being drawn in a computer display, successive polygon vertexes may be drawn at the same pixel if they are closer than some fixed application tolerance. For example, in a large range geographic map display, two successive vertexes of a boundary polyline may be separated by many miles, and still be drawn at the same pixel pointg on the display screen; and the edge segments joining them are also all being drawn from this pixel to itself. One would like to discard these redundant vertexes so that successive displayed vertexes are separated several pixels, and edge segments are not just drawn as points.

Vertex cluster reduction is a brute-force algorithm for polyline simplification. For this algorithm, a polyline vertex is discarded when its distance from a prior initial vertex is less than some minimum tolerance $\delta > 0$. Specifically, after fixing an initial vertex V_0, successive vertexes V_i are tested and rejected if they are less than δ away from V_0. But, when a vertex is found that is further away than δ, then it is accepted as part of the new simplified polyline, and it also becomes the new initial vertex for further simplification of the polyline. Thus, the resulting edge segments between accepted vertexes are larger than the δ tolerance. This procedure is easily visualized as follows:



Pseudo-Code: Vertex Reduction

Here is the straightforward pseudo-code for this algorithm:

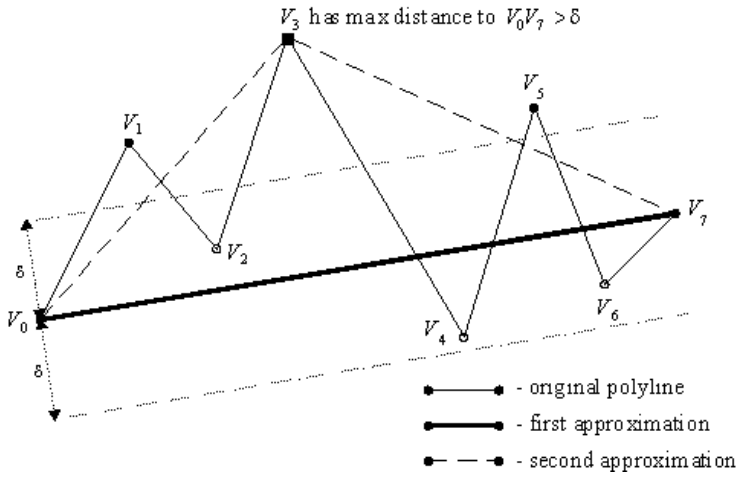
```
Input: tol = the approximation tolerance
        Λ = {V0,V1,...,Vn-1} is a n-vertex polyline

Set start = 0;
Set k = 0;
Set W0 = V0;
for each vertex Vi (i=1,n-1)
{
    if Vi is within tol from Vstart
    then ignore it, and continue with the next vertex

    else Vi is further than tol away from Vstart {
        add it as a new vertex of the reduced polyline
        Increment k++;
        Set Wk = Vi;
        Set start = i; as the new initial vertex
    }
}

Output: Ω = {W0,W1,...,Wk-1} = the k-vertex reduced polyline
```

This is a fast $O(n)$ algorithm. It should be implemented comparing squares of distances with the squared tolerance to avoid expensive square root calculations. This algorithm is the initial preprocessing stage in our implementation for poly_decimate() given below.

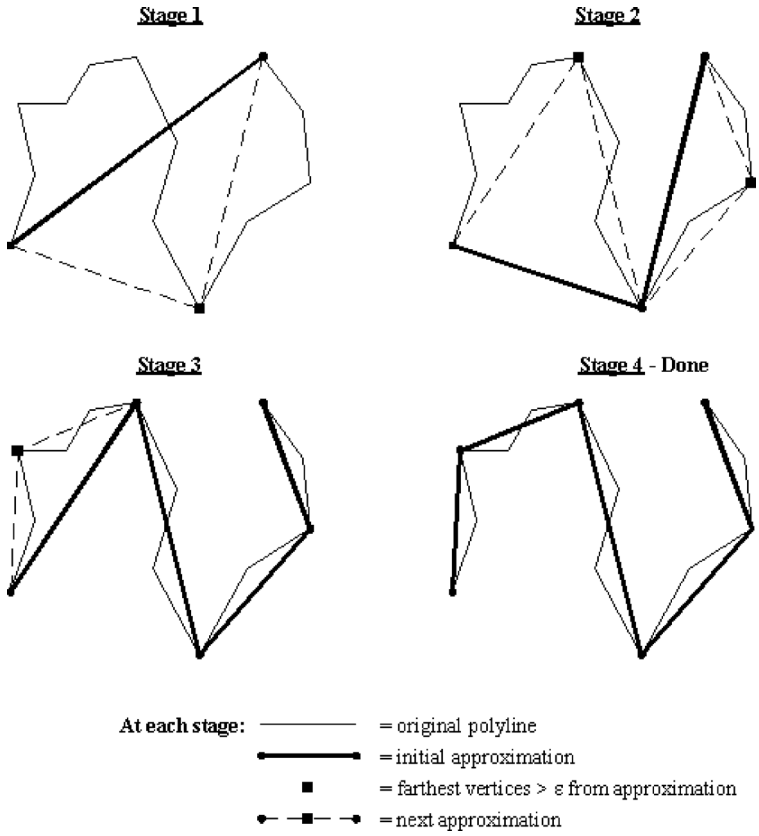
Douglas-Peucker Algorithm

Whereas vertex reduction uses closeness of vertexes as a rejection criterion, the Douglas-Peucker (DP) algorithm uses the closeness of a vertex to an edge segment. This algorithm works from the top down by starting with a crude initial guess at a simplified polyline, namely the single edge joining the first and last vertexes of the polyline. Then the remaining vertexes are tested for closeness to that edge. If there are vertexes further than a specified tolerance, $\delta > 0$, away from the edge, then the vertex furthest from it is added the simplification. This creates a new approximation for the simplified polyline. Using recursion, this process continues for each edge of th current approximation until all vertexes of the original polyline are within tolerance of the simplification.

More specifically, in the DP algorithm, the two extreme endpoints of a polyline are connected with a straight line as the initial rough approximation of the polyline. Then, how well it approximates the whole polyline is determined by computing the distances from all intermediate polyline vertexes to that (finite) line segment. If all these distances are less than the specified tolerance δ, then the approximation is good, the endpoints are retained, and the other vertexes are eliminated. However, if any of these distances exceeds the δ tolerance, then the approximation is not good enough. In this case, we choose the point that is furthest away as a new vertex subdividing the original polyline into two (shorter) polylines, as illustrated in the following diagram.



This procedure is repeated recursively on these two shorter polylines. If at any time, all of the intermediate distances are less than the δ threshold, then all the intermediate points are eliminated. The routine continues until all possible points have been eliminated. Successive stages of this process are shown in the following example.



Pseudo-Code: Douglas-Peucker

Here is pseudo-code for this algorithm:

```
Input:  tol = the approximation tolerance
          L = {V0,V1,...,Vn-1}  is any n vertex polyline

// Mark vertexes that will be in the reduced polyline
Initially Mark V0 and Vn

// Recursively decimate by selecting vertex furthest away
decimate(tol, L,  0, n);

// Copy Marked vertexes to the reduced polyline
for (i=m=0; i<=n; i++) {
    if (Vi is marked) { add it to the output polyline W
        Wm = Vi;
        m++;
    }
}

Output: W = {W0,W1,...,Wm-1} = the reduced m-vertex polyline

//------------------------------------------------------------
// This is the recursive decimation routine
decimate(tol, L,  j, k)
{
    if (k £ j+1) there is nothing to decimate
        return immediately
    otherwise
    test distance of intermediate vertexes from segment Vj to Vk

    Put Sjk  = the segment from Vj to  Vk
    Put maxd = 0 is the distance of farthest vertex from Sjk
    Put maxi = j is the index of the vertex farthest from Sjk

    for each intermediate vertex Vi (i=j+1,k-1)
    {
        Put dv = distance from Vi to segment Sjk
        if (dv <= maxd) then Vi is  not farther away, so
            continue to  the next vertex
        otherwise Vi is a new max  vertex
        Put maxd = d and maxi = i to remember the farthest vertex
    }
    if (maxd > tol) a vertex is further than tol from Sjk
    {
        // split the polyline at the farthest  vertex
        Mark Vmaxi as part of the reduced polyline
        and recursively decimate the two  subpolylines
        decimate(tol, L, j, maxi);
        decimate(tol, L, maxi, k);
```

```
    }
}
```

This algorithm is $O(mn)$ worst case time, and $O(n \log m)$ expected time, where *m* is the size of the reduced polyline. Note that this is an output dependent algorithm, and will be very fast when *m* is small. This algorithm is implemented as the second stage of our `poly_decimate()` routine given below. The first stage of our implementation does vertex reduction on the polyline before invoking the DP algorithm. This results in a fast high-quality polyline approximation and simplification algorithm.

Convex Hull Speed-Up

[Hershberger & Snoeyink, 1992] describe an interesting improvement for speeding up the Douglas-Peucker algorithm, making it a worst case $O(n \log n)$ time algorithm. They do this by speeding up the selection of the farthest intermediate vertex from the segment $S_{ij} = V_i V_j$. This is achieved by noting that the farthest vertex must be on the convex hull of the polyline chain from V_i to V_j. Then, they compute this hull in $O(n)$ time using Melkman's algorithm for the hull of a simple polyline, and find the vertex farthest from S_{ij} using an $O(\log n)$ binary search on the hull vertexes. They next show how to reuse the hull information when a polyline chain is split at the farthest vertex. The details for putting the whole algorithm together are a bit delicate, but in the final analysis it is shown to have worst case $O(n \log n)$ time.

There are a few caveats about this algorithm. First, it depends on using the Melkman $O(n)$ time hull algorithm, and thus the polyline must be simple, which is often true in many applications. Second, since this algorithm depends on a 2D convex hull algorithm, it only applies to planar polylines, whereas the VR and DP algorithms can be used for polylines in any dimension. Next, this algorithm is more complicated than the original DP algorithm, and is more difficult to code and debug. Fortunately, the [Hershberger & Snoeyink, 1992] paper has an Appendix with a complete "C" code implementation. Finally, although they improve on the worst case behavior, the original DP algorithm is usually $O(n \log m)$ and can do better in the best cases where *m* is small. So, altogether its a stalemate over which algorithm is best, and using the easier to code, more general DP algorithm is usually preferred.

Implementation

Here is a sample "C++" implementation of this algorithm.

```
// Copyright 2001, 2012, 2021 Dan Sunday
// This code may be freely used and modified for any purpose
// providing that this copyright notice is included with it.
// There is no warranty for this code, and the author of it cannot
// be held liable for any real or imagined damage 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;}     // as many as are needed
//        operators for:
//            == to test equality
//            != to test inequality
//            (Vector)0 = (0,0,0)          (null vector)
//            Point  = Point ± Vector
//            Vector = Point - Point
//            Vector = Vector ± Vector
//            Vector = Scalar * Vector     (scalar product)
//            Vector = Vector * Vector     (cross product)
//    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 norm2(v)   dot(v,v)          // norm2 = squared vector length
#define norm(v)    sqrt(norm2(v))    // norm = length of vector
#define d2(u,v)    norm2((u)-(v))    // distance squared
#define d(u,v)     norm((u)-(v))     // distance = norm of difference


// poly_decimate(): - remove vertices to get a reduced polygon
//    Input:  tol = approximation tolerance
//            V[] = polyline array of vertex points
//            n   = the number of points in V[]
//    Output: sV[]= reduced polyline vertexes (max is n)
//    Return: m   = the number of points in sV[]
int
poly_decimate( float tol, Point* V, int n, Point* sV )
{
    int    i, k, m, pv;             // misc counters
    float  tol2 = tol * tol;        // tolerance squared
    Point* vt = new Point[n];       // vertex buffer
    int*   mk = new int[n] = {0};   // marker  buffer

    // STAGE 1.  Vertex Reduction within tolerance
    vt[0] = V[0];           // start at the beginning
    for (i=k=1, pv=0; i<n; i++) {
        if (d2(V[i], V[pv]) < tol2)
            continue;
        vt[k++] = V[i];
        pv = i;
    }
    if (pv < n-1)
        vt[k++] = V[n-1];   // finish at the end

    // STAGE 2.  Douglas-Peucker polyline reduction
    mk[0] = mk[k-1] = 1;    //  mark the first and last vertexes
    poly_decimateDP( tol, vt, 0, k-1, mk  );

    // copy marked vertices to the reduced polyline
```

```
    for (i=m=0; i<k; i++) {
        if (mk[i])
            sV[m++] =  vt[i];
    }
    delete vt;
    delete mk;
    return m;          //  m vertices in reduced polyline
}


// poly_decimateDP():
//  This is the Douglas-Peucker recursive reduction routine
//  It marks vertexes that are part of the reduced polyline
//  for approximating the polyline subchain v[j] to v[k].
//    Input:  tol  = approximation tolerance
//            v[]  = polyline array of vertex points
//            j,k  = indices for the subchain v[j] to v[k]
//    Output: mk[] = array of markers matching vertex array v[]
void
poly_decimateDP( float tol, Point* v, int j, int k, int* mk )
{
    if (k <= j+1) // there is nothing to decimate
        return;

    // check max distance of segment S from polyline v[j] to v[k]
    int     maxi = j;           // index of vertex farthest from S
    float   maxd2 = 0;          // farthest vertex distance squared
    float   tol2 = tol * tol;   // tolerance squared
    Segment S = {v[j], v[k]};   // segment from v[j] to v[k]
    Vector  u = S.P1 - S.P0;    // segment direction vector
    double  cu = dot(u,u);      // segment length squared

    // test each vertex v[i] for max distance from S
    // compute the distance of each point to the segment
    // Note: this works in any dimension (2D, 3D, ...)
    Vector  w;
    Point   Pb;             // base of perpendicular from v[i] to S
    double  b, cw, dv2;     // dv2 = distance v[i] to S squared

    for (int i=j+1; i<k; i++)
    {
        // compute distance squared
        w = v[i] - S.P0;
        cw = dot(w,u);
        if ( cw <= 0 )
            dv2 =d2(v[i], S.P0);
        else if ( cu <= cw )
            dv2 =d2(v[i], S.P1);
        else {
            b = cw / cu;
            Pb = S.P0 + b  * u;
            dv2 =d2(v[i], Pb);
        }
        // test with max distance squared
```

```
        if (dv2 <= maxd2)
            continue;
        // v[i] is a new max vertex
        maxi = i;
        maxd2 = dv2;
    }
    if (maxd2 > tol2)          // error is worse than the tolerance
    {
        // split the polyline at the farthest  vertex from S
        mk[maxi] = 1;       // mark v[maxi] for the reduced polyline
        // recursively decimate the two subpolylines at v[maxi]
        poly_decimateDP(  tol, v, j, maxi, mk );  // v[j] to v[maxi]
        poly_decimateDP(  tol, v, maxi, k, mk );  // v[maxi] to v[k]
    }
    // else the approximation is OK, so ignore intermediate vertexes
    return;
}
//===================================================================
```

References

David Douglas & Thomas Peucker, "Algorithms for the reduction of the number of points required to represent a digitized line or its caricature", The Canadian Cartographer 10(2), 112-122 (1973)

John Hershberger & Jack Snoeyink, "Speeding Up the Douglas-Peucker Line-Simplification Algorithm", Proc 5th Symp on Data Handling, 134-143 (1992). UBC Tech Report available online from CiteSeer.

C++ CODE INDEX

This is an index for all the code in this book. Also, one can download a zip file which includes all the code in this book. The download also includes prototype Point and Vector class structures that are not given in this book since these classes are often application specific and provided by standard libraries. Nevertheless, some readers may find this code useful, so they are included in the download. To get the download, go to GeometryAlgorithms.com/code.html.

Primitive Functions

Application Functions

Function	Description	Page
EventQueue	A Class to manage an ordered event queue	97
SweepLine	A Class to manage a sweep line which uses a balanced binary tree (BBT not provided)	99
simple_Polygon()	Test if a polygon is simple	101
fastBall()	Get a bounding ball approximation for a set of points	114
chainHull_2D()	Andrew's monotone chain convex hull	124
nearHull_2D()	The BFP approximate convex hull	131
simpleHull_2D()	Melkman's simple polyline convex hull	140
intersect2D_SegPoly()	Intersect segment and convex polygon	150
polyMax_2D()	Get the max vertex a polygon in a direction	158
dist2D_Poly_to_Line()	Get distance from a polygon to a line	160
tangent_PointPoly()	Get tangents from a point to a polygon	167
tangent_PointPolyC()	Get tangents to a convex polygon	168
RLtangent_PolyPolyC()	Get the RL tangent between convex polygons	170
poly_decimate()	Remove polygon vertices to get a smaller approximate polygon	179

This book presents practical geometry algorithms with computationally fast C++ code implementations. It covers algorithms for fundamental geometric objects, such as points, lines, rays, segments, triangles, polygons, and planes. These determine basic 2D and 3D properties, such as area, distance, inclusion, and intersections. There are also algorithms compute bounding containers for these objects, including a fast bounding ball, various convex hull algorithms, as well as polygon extreme points and tangents. And there is a fast algorithm for polyline simplification using decimation that works in any dimension.

These algorithms have been used in practice for several decades, and are robust, easy to understand, code, and maintain. And they execute very rapidly in practice, not just in theory. For example, the winding number point in polygon inclusion test, first developed by the author in 2000, is the fastest inclusion algorithm known, and works correctly even for nonsimple polygons. There is also a fast implementation of the Melkman algorithm for the convex hull of a simple polyline. And much more. If your programming involves geometry, this will be an invaluable reference.

Dr. Daniel Sunday is a retired mathematician who previously worked in the fields of Algebraic Topology, Differential Geometry, Computational Geometry, and Computer Graphics. He worked as a mathematician at Universidad de Los Andes, UC Berkeley, and Johns Hopkins University Applied Physics Lab. At JHU, he taught courses in Computer Programming, Software Development, UNIX O/S, Computer Graphics and Computational Geometry.

ISBN 9798749449730