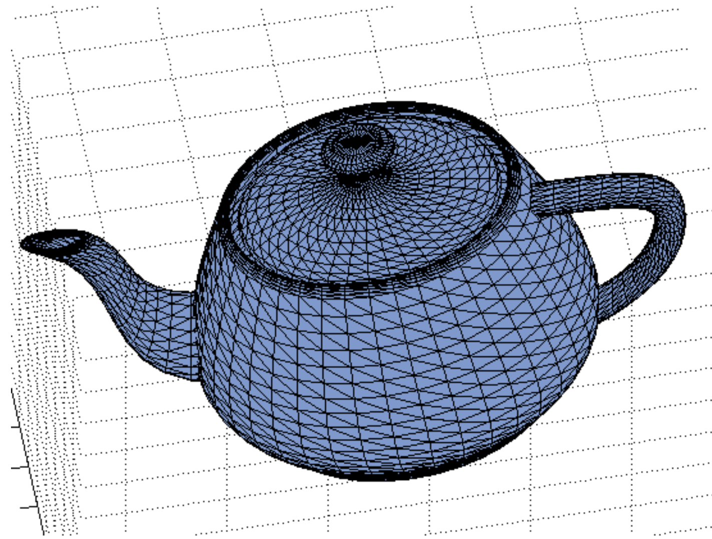
# Computational Geometry: Points and Lines
# Mathematical Programming with Python

https://people.sc.fsu.edu/~jburkardt/classes/math1800_2023/geometry1/geometry1.pdf

---



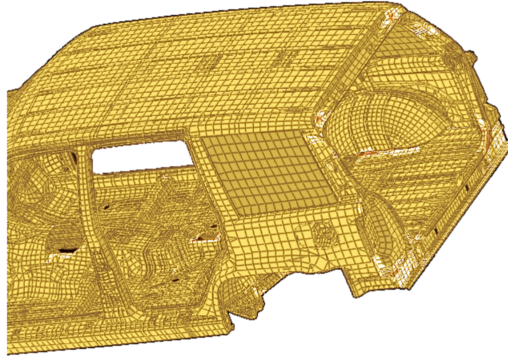
*The teapot is classic test in computational geometry.*

"Computational Geometry"

- *Geometry is the study of position, shape, distance and angles;*
- *Often a picture makes a geometric problem easier;*
- *A computer algorithm can't see anything;*
- *We have to express our problems and algorithms computationally;*
- *We need to define "near to", "inside of", "perpendicular to", "central to", "on the border of";*
- *Computational geometry is vital for computer graphics.*
- *Computer games rely on rapid computation of changing geometry.*
- *Computation of heat transfer, fluid flow, and industrial processes rely on geometric models.*
- *Medical scanners reconstruct a body from image slices, and then reconstruct individual organs.*

## 1 Computational Geometry

Computational geometry moves geometry from the study of simple polygons to the analysis and modeling of very complicated real-world shapes. We are still using fairly simple ideas about the positions and shapes of points and lines and surfaces and solids; but the computer allows us to deal with hundreds and thousands of such items. Our eye can no longer see the solution to such problems; we need to think hard about how to describe a particular object, what we want to know about it, and how to go about answering our questions.



*Geometric models are used to design and test new models of cars.*

For computational work, we need to decide how to represent points, lines, and other objects, how to determine relationship between them, and how to construct complicated structures out of these basic objects. In this discussion, we will concentrate on two dimensional space. The extension to 3D objects is not difficult to imagine.

We will consider some of the following computational geometry tasks:

- measurement of length, area, volume, distance, direction, orientation;
- discretization of curves and surfaces;
- indexing or parameterizing the parts of an object;
- finding the nearest object;
- intersection of two objects;
- containment of one object in another;
- parallel and orthogonal components;
- decomposition of an object into basic objects;
- transformations: shift, rotate, reflect, shear an object;
- sampling a random object in a set.
- integrating some function, such as mass, over the range of an object;
- plan a path for an object around a series of obstacles;

# 2 A point

The "atoms" of geometry are points; every geometric object can be thought of as a collection of points that satisfy some property. Depending on what we are studying, our geometry can be 1D, 2D, 3D or a higher, abstract dimension. Unless we are studying a special surface like the sphere, we will usually think of a point in terms of its Cartesian coordinates. Thus, in 2D, a point named $p0$ would have coordinates $(x, y)$. In Python, a single point would be a `numpy` array:

```
p0 = np.array ( [ x, y ] )
```

There's not much more we can say about a single point. But if we have a second point $p1$, we can certainly measure the distance between them. In 2D, this would be

$$r = ||p1 - p0|| = \sqrt{(p1.x - p0.x)^2 + (p1.y - p0.y)^2}$$

or, in Python

```
r = np.linalg.norm ( p1 - p0 )
```

For example,

```
p0 = np.array ( [ 0, 4 ] )
p1 = np.array ( [ 3, 1 ] )
r = np.linalg.norm ( p1 - p0 )
```

we should compute a distance of approximately $r = 4.24$.

The actual coordinates of a point depend on the origin and orientation of our coordinate system. However the distance $r$ between two points, and the distance vector $p1 - p0$ are independent of the position and orientation of our coordinate system. In physics, this is an important fact to notice. For our computational work, we won't worry so much about this distinction.

# 3 A line

We can see a line in our minds, or sketch one on paper, but now we need to explain this to a computer. One way to see what is going on is to recall the statement that *two points determine a line*. So given points $p0$ and $p1$, there is some way to determine whether another point $q$ is on the line they determine.

We would think that $q$ is on the line if you can start at $p0$, and go...in the direction defined by $p1$. A direction is another word for a *vector*, and a vector can be computed as the difference between two points. So presumably, our direction $v$ is something like

$$v = p1 - p0$$

A point $q$ is on the line if it can be written as the sum of $p0$ plus a certain amount of this displacement vector $v$

$$q = p0 + s * (p1 - p0)$$

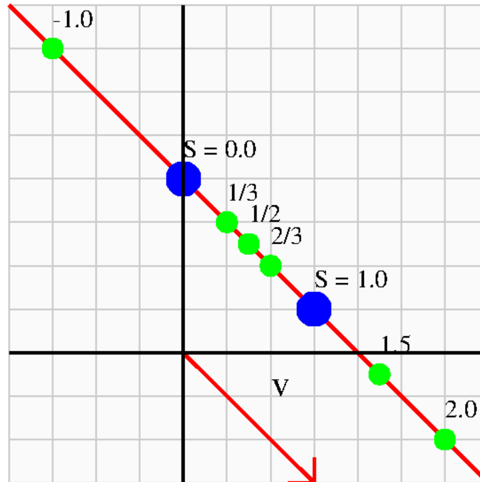
We can rewrite this in the *convex combination* form:

$$q(s) = (1 - s) * p0 + s * p1$$

Now note how the value $s$ describes the position of the point $q(s)$:

| | |
|---|---|
| $s < 0$ | $q(s)$ is on the $p0$ side of the line |
| $s = 0$ | $q(s) = p0$ |
| $0 < s < 1$ | $q(s)$ is between $p0$ and $p1$ |
| $s = 1$ | $q(s) = p1$ |
| $s > 1$ | $q(s)$ is on the $p1$ side of the line |

For example, choose $p0$ = (0,4), and $p1$ = (3,1). Then $v = p1 - p0$ = (3,-3).




```
 s     p0    + s   * p1-p0  =    q(s)
---   ----  ----    ------    ---------
-1   (0,4) -  1   * (3,-3) = (-3,   7  )
 0   (0,4) +  0   * (3,-3) = ( 0,   4  )
1/3  (0,4) + 1/3 * (3,-3) = ( 1,   3  )
1/2  (0,4) + 1/2 * (3,-3) = ( 1.5, 2.5)
2/3  (0,4) + 2/3 * (3,-3) = ( 2,   2  )
 1   (0,4) +  1   * (3,-3) = ( 3,   1  )
 2   (0,4) +  2   * (3,-3) = ( 6,  -2  )
```

and we could create a corresponding helper function:

```
def point_from_parameter ( p0, p1, s ):
  q = ( 1.0 - s ) * p0 + s * p1
  return q
```

# 4 Representing a line

There are several equivalent ways of representing a line. It is useful to know the details, and to understand how to switch from one to another.

- *geometric*:: points on line through $p_0$ and $p_1$;
- *parametric*:: $p(s) = (1 - s) * p_0 + s * p_1$;
- *vector*:: $p(s) = p_0 + s * (p_1 - p_0)$;
- *algebraic*: $y = \text{slope} * x + \text{intercept}$;
- *linear algebra*: $a_0 * x + a_1 * y = b$;

Given a geometric description of a line, we can derive the algebraic formula as follows. Write $p_0 = (x_0, y_0)$ and $p_1 = (x_1, y_1)$

$$\begin{aligned}
\text{slope} &= \frac{y_1 - y_0}{x_1 - x_0} \\
y_0 &= \text{slope} * x_0 + \text{intercept} \\
\text{intercept} &= y0 - \text{slope} * x_0 \\
y &= \text{slope} * x + \text{intercept}
\end{aligned}$$

and if we have the algebraic form, we can easily rearrange it into one version of the linear algebraic form:

$$\text{slope} * x - y = -\text{intercept}$$

## 5 Given a point $q$ on the line, what is its $s$ coordinate?

Having this formula for a line is a huge advantage:

- the formula can be regarded as the definition of the line;
- we can compute points on a line;
- we can solve for $s$ given a desired $x$ or $y$ location;
- we can guess how to define lines in 1D, 3D, or any dimension;
- the value $s$ is like a coordinate;
- in any dimension, we only need 1 number to locate points.

We commonly refer to $s$ as a parameter, a sort of index that allows us to keep track of all the points on the line in an orderly way.

Suppose we start with the $(x, y)$ coordinates of a point $q$ that we know is on the line. Can we determine $s$? We could reason that $s$ is the signed relative distance from $p0$ to $q$, so check which of the following is true:

$$q = p0 + ||q - p0|| * v/||v||$$
$$q = p0 - ||q - p0|| * v/||v||$$

But a better way uses the dot product of two vectors:

$$u \cdot v = \sum u_i \, v_i = ||u|| \, ||v|| \cos(\theta)$$

where $\theta$ is the angle between the two vectors. We can use the dot product to compare the line definition vector $v$ to the difference vector $q - p0$:

$$(q - p0) \cdot v = \text{dot} = ||q - p0|| \, ||v|| \cos(\theta)$$
$$s = \pm \frac{||q - p0||}{||v||} = \frac{\text{dot}}{||v||^2}$$

For example, suppose $p0 = (0, 4)$ and $p1 = (3, 1)$ so $v = (3, -3)$, and the point $q = (-2, 6)$. Then the following program will compute $s = -0.666...$ and can reconstruct the point $q$:

```
def parameter_from_point ( p0, p1, q ):
  import numpy as np
  vnorm = np.linalg.norm ( p1 - p0 )
  s = np.dot ( q - p0, p1 - p0 ) / vnorm**2
  return s
```

Notice that this formula doesn't seem to depend on working in 2D. If we have a line in 3D, defined by points $p0$ and $p1$, and we want to get the $s$ coordinate of a point $q$ that we know is on the line, we do exactly the same sequence of steps, in this case getting a value of $s = 0.333...$:

```
  p0 = np.array ( [ 1, 5, 3 ] )
  p1 = np.array ( [ 4, 2, 9 ] )
  q = np.array ( [ 2, 4, 5 ] )
```

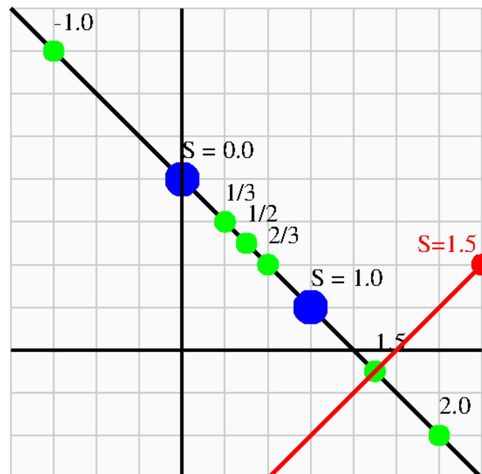
# 6 Points not on the line?

Consider the point $q = (7, 2)$, which is not on our line. Nonetheless, we can compute its $s$ value and see what we get:

```
  q = np.array ( [ 7, 2 ] )
  s = parameter_from_point ( p0, p1, q )
  q2 = point_from_parameter ( p0, p1, s )
```

The value of $s$ comes out to 1.5, and the recomputed $q2 = (4.5, -0.5)$. The value of $q2$ must be a point on the line. We can see that it is actually the point on the line that is closest to $q$; the line from $q$ to $q2$ is perpendicular to the line.

```
def point_nearest ( p0, p1, q ):
  s = parameter_from_point ( p0, p1, q )
  q2 = point_from_parameter ( p0, p1, s )
  return q2
```




In other words, we have now found that the vector $(q - p0)$ can be decomposed into two parts

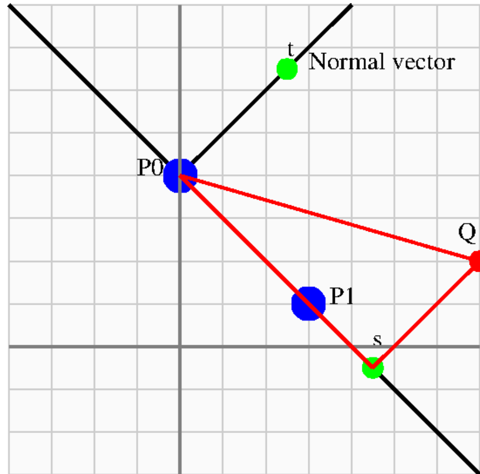
$$q - p0 = (q - q2) + (q2 - p0)$$

and these two components are perpendicular:

$$(q - q2) \cdot (q2 - p0) = 0$$

By Pythagoras, we have

$$||q - p0||^2 = ||q - q2||^2 + ||q2 - p0||^2$$




## 7 Distance from a point to a line

If we know how to find the nearest point on a line, then we can compute the distance $t$ to the line.

```
def line_distance ( p0, p1, q ):
  s = parameter_from_point ( p0, p1, q )
  q2 = point_from_parameter ( p0, p1, s )
  w = q - q2
  t = np.linalg.norm ( w )
  return t
```

The vectors $v$ and $w$ are very similar to Cartesian coordinate directions, and the values $s$ and $t$ to Cartesian coordinates. In other words, we can write

$$q = s * v + t * w$$

The vectors $v$ and $w$ do not have unit length, so certain tools don't work. If we replace the vectors by unit versions

$$\hat{v} = v/||v||$$
$$\hat{w} = w/||w||$$

then we have to recompute the values of $s$ and $t$ as well:

$$\hat{s} = s * ||v||$$
$$\hat{t} = t * ||w||$$

after which we can make some useful formulas. In particular, we have

$$r = ||q - p0|| = \sqrt{\hat{s}^2 + \hat{t}^2}$$

and other results that we are familiar with in normal Cartesian coordinates.

## 8 Distance from a point to a line segment

Two points $p_0$ and $p_1$ can also be regarded as defining a line segment, that is, the set of points on the line that are actually between the two endpoints. Given a point $q$, we can ask for its distance to this shorter

object. We can still compute the $s$ parameter that corresponds to $q$, and identifies the closest point on the full line. If that point is also on the line segment, then the distance to the line segment is the same as to the line. If $s < 0$, then $p_0$ is now the closest point, and the distance is $||q - p_0||$, while if $1 < s$, the distance is $||q - p_1||$.

```
def line_segment_distance ( p0, p1, q ):
  s = parameter_from_point ( p0, p1, q )
  if ( s <= 0.0 ):
    t = np.linalg.norm ( q - p0 )
  elif ( 1.0 <= s ):
    t = np.linalg.norm ( q - p1 )
  else:
    q2 = point_from_parameter ( p0, p1, s )
    w = q - q2
    t = np.linalg.norm ( w )
  return t
```

We will find this computation useful when we ask for the distance from a point $q$ to a triangle; then, we want the minimum of the distances from $q$ to each of the three line segments that are the sides of the triangle.

# 9 Which side are you on?

Suppose a line is defined by pair of points $p0$ and $p1$. If we walk from $p0$ towards $p1$, then the line defines two half-planes, on our left and our right. Suppose we are given the coordinates of a point $q$. Can we determine whether it is to our left or our right?

We are very lucky that this is so. We can do this thanks to the *vector cross product*, which is written $v0 \times v1$. The vector cross product is defined for both 2D and 3D vectors, but we will only work with the 2D case.

- If $v0$ and $v1$ are vectors, and $z$ is the point $(0,0)$ then the vector cross product returns the signed area of the parallelogram with vertices $0, v0, v0 + v1, v1$.
- If the vertices occur in counterclockwise order, the area is positive. Otherwise it is negative.
- The cross product has the form
$$v0 \times v1 = ||v0||||v1|| \sin(\theta)$$
- The formula for the cross product is
$$v0 \times v1 = (v0x * v1y - v1x * v0y)$$
- In Python, the cross product is computed by

```
      cp = np.cross ( v0, v1 )
```

  Use (`float()` to turn this into a number!

Now let the line be defined, as usual, by $p0 = (0,4)$ and $p1 = (3,1)$ and consider the points $q1 = (7,2)$. Construct the vectors v0 = p1 - p0 v1 = q1 - p0 As we are walking along the line from p0 to p1, the point q1 is to the left of us. That means the vertices of the parallelogram (z,v0,v0+v1,v1) are listed in counterclockwise order. And that means the signed area will be positive:

```
  q1 = np.array ( [ 7, 2] )
  v0 = p1 - p0
  v1 = q - p0
  v0xv1 = float ( cross ( v0, v1 ) )
```

returns the value `v0xv1 = 15`. If we repeat this calculation, but with $q2 = (2, -3)$, we get `v0xv1 = -15`, because $q2$ is to the *right* of the line. And if we try $q3 = (4.5, -0.5)$, we get `v0xv1 = 0`, because $q3$ is exactly on the line.

This means we have a computational procedure for determining where any point $q$ is relative to a given line:

```
def line_side ( p0, p1, q ):
  import numpy as np
  v0 = p1 - p0
  v1 = q1 = p0
  v0xv1 = np.float ( np.cross ( v0, v1 ) )
  return v0xv1
```

We will soon make heavy use of this function when we consider how to determine whether a point is inside a triangle.

# 10 Random sampling

If $r$ is a random number $0 \leq r \leq 1$ then a random point between $p0$ and $p1$ can be defined, again by a convex combination:

$$q = (1 - r)\, p0 + r\, p1$$

which is essentially the formula we found for defining points on the line. This formula can be used for a line in any dimension.

Random sampling will become much more challenging when looking at geometric shapes in 2D.

# 11 Integral estimation

A Monte Carlo method for estimating the integral of a function $f(x)$ over the line $p0 \leq x \leq p1$ is

$$I(f) = \int_{p0}^{p1} f(x) dx \approx \frac{p1 - p0}{n} \sum_{i=0}^{i<n} f(x_i)$$

where $x_i$ is one of $n$ randomly generated points in the interval.

Analogously, if $p0$ and $p1$ are points in 2D, we can approximate an integral of $f(x, y)$ over the the line between these points by

$$I(f) = \int_{p0}^{p1} f(x, y) dx\, dy \approx \frac{||p1 - p0||}{n} \sum_{i=0}^{i<n} f(x_i, y_i)$$

where each point $(x_i, y_i)$ is a random point on the line. A similar procedure can be adopted for a line in any higher dimension.

For example, we can estimate

$$I(f) = \int_{0}^{2} \frac{100}{(10x - 3)^2 + 1} + \frac{100}{(10x - 9)^2 + 4} - 6$$

whose exact value to 20 digits is $I(f) = 29.326213804391148551$

It may seem strange to estimate an integral this way, since there are many good integration programs available already; for example, we know about `scipy.integrate.quad()`. Really, we are only introducing the Monte Carlo approach for the line because later we will recognize and understand it when we want to estimate integrals over triangles, polygons, and other shapes for which there are not so many prewritten algorithms available.

## 12 A note about vectors

Geometric points and vectors might seem to be roughly the same thing, since in 2D we can describe either object by specifying a pair of values $(x, y)$. However, logically, we really think of a point as a position, whereas a vector is a direction; in this case, it is the direction suggested by starting at the origin and moving to $(x, y)$.

Vectors have two important properties:

- the length: $||v|| = \sqrt{x^2 + y^2}$
- the angle: $\theta(v) = \tan^{-1} \frac{y}{x}$

The preferred computation of $\theta(v)$ uses an alternate inverse tangent function, $\theta(v) = \tan 2^{-1} \frac{y}{x}$ which in `numpy` would be expressed by

```
theta = np.arctan2 ( y, x )
```

The `arctan2()` function is able to return an angle in all four quadrants, whereas the `arctan()` function only returns angles in the first or fourth quadrants. This is because it uses the ratio of $y$ and $x$, and hence can't distinguish an angle and its negative.

Unlike points, it is possible to do arithmetic with vectors. Given a vector $v$, we can increase its length by a factor $\alpha$:

$$w = \alpha v$$

Given two vectors $u$ and $v$, we can form the sum

$$w = u + v$$

The diagram illustrating this process shows how $w$ is the diagonal of a parallelogram whose two sides are $u$ and $v$. In general, we can form a linear combinations of vectors.

$$w = \alpha u + \beta v$$

Linear algebra, in fact, is the study of these linear transformations of vectors.

## 13 Summary of point discussion

We've run across some examples of classic geometric tasks:

- parameterization: where is the point $p$ on the line;
- distance: how far is the point $q$ from the line?
- parameterization: where is the offline point $q$?
- parallel motion: how we moved along the line.
- orthogonal: how we looked at points off the line.
- containment: is the point $p$ on the line from $p0$ to $p1$?
- nearness: which point $p$ on the line is nearest the point $q$?
- mapping: between $(x, y)$ and $(s, t)$ coordinates
- orientation: which side of the line is offline point $q$?
- sampling: picking representative points from the line
- quadrature: estimating integrals