Geometry Algorithms

"Geometry Algorithms"
http://people.sc.fsu.edu/~jburkardt/presentations/
asa_geometry_2011.pdf

.....

ISC4221C-01:
Algorithms for Science Applications II

.....

John Burkardt
Department of Scientific Computing
Florida State University

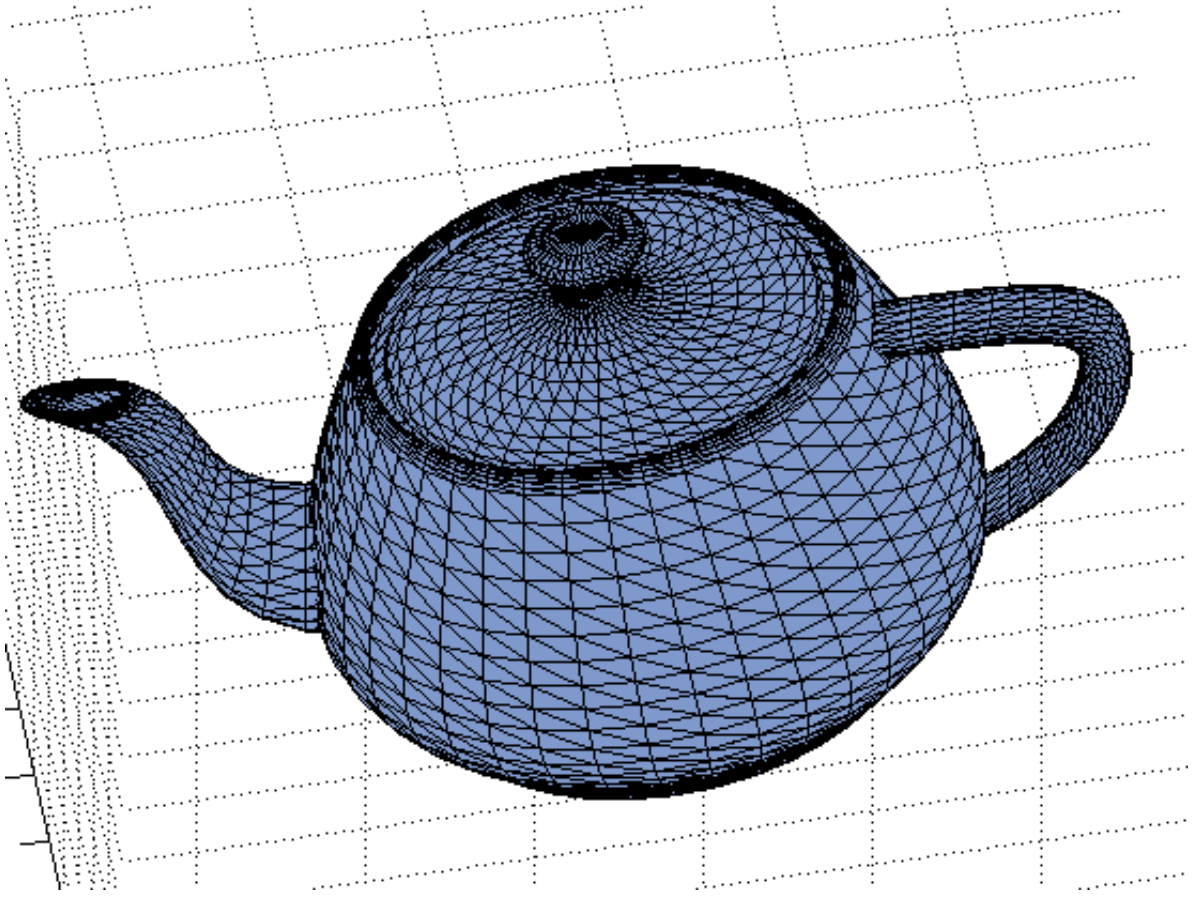
Spring Semester 2011

Geometry Algorithms

- **Overview**
- The Points on a Line
- Points NOT on a Line
- Estimating Integrals over an Interval
- Triangles and their Properties
- Triangulating a Polygon
- The Convex Hull
- Triangulating a Point Set by Delaunay
- Estimating Integrals over a Triangle
- Conclusion

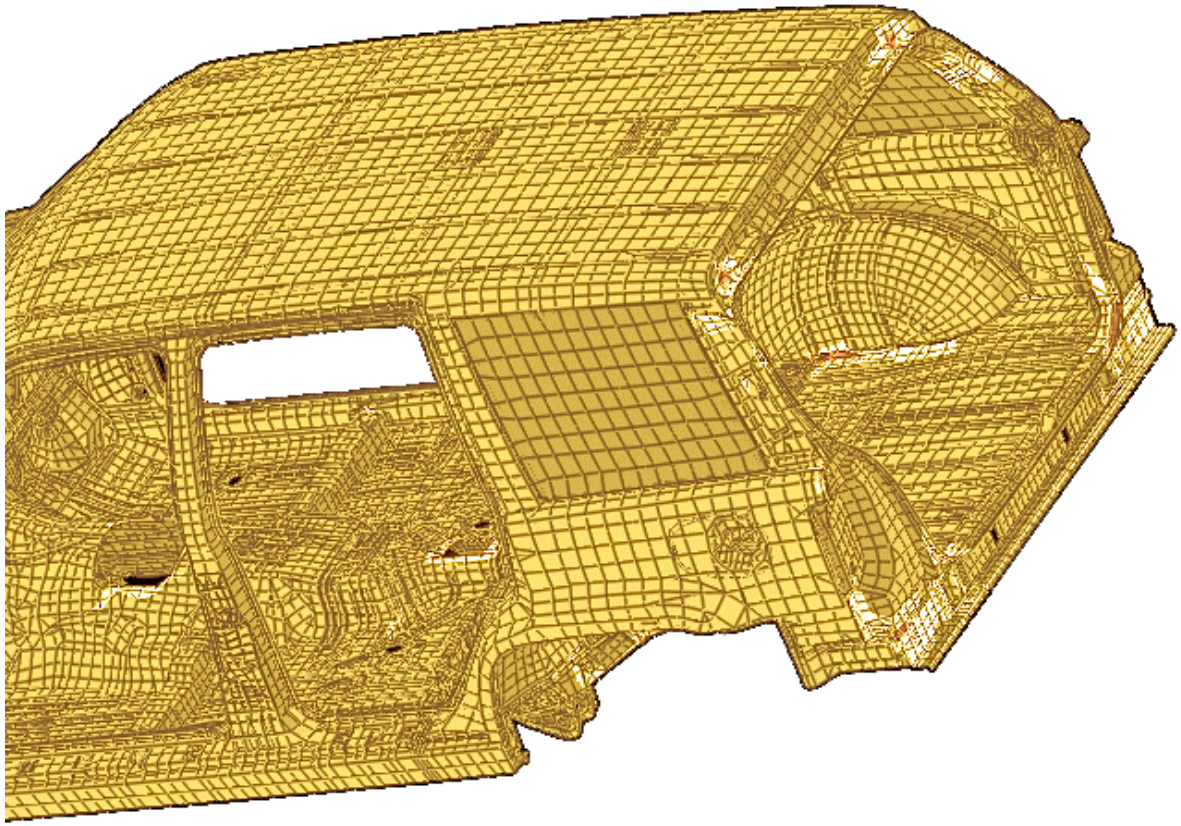
OVERVIEW: Geometry

We use computational geometry to decompose an object into simple shapes that can be displayed in a computer animation.



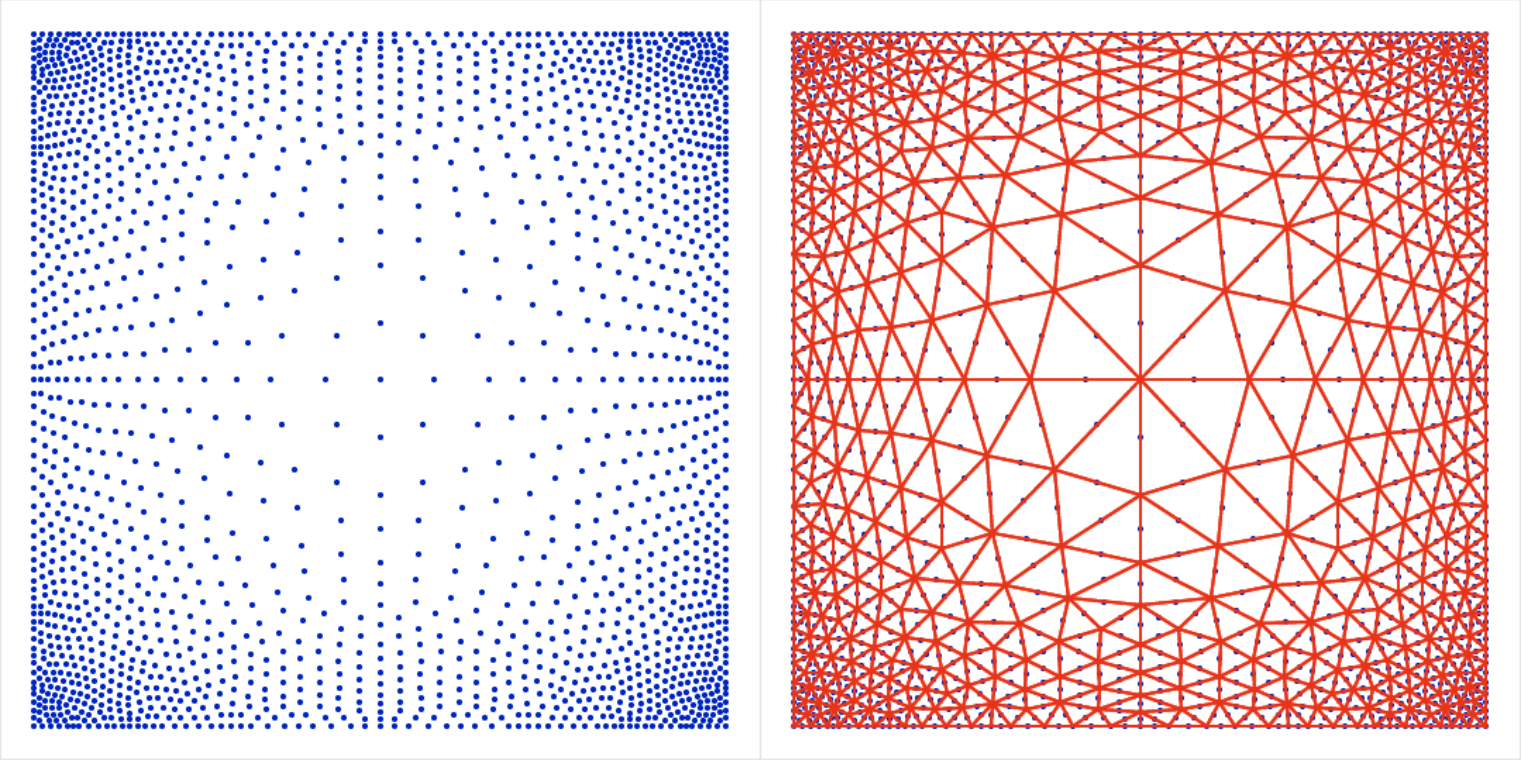
OVERVIEW: Geometry

Computational geometry builds models that can be stressed or crashed for realistic tests.



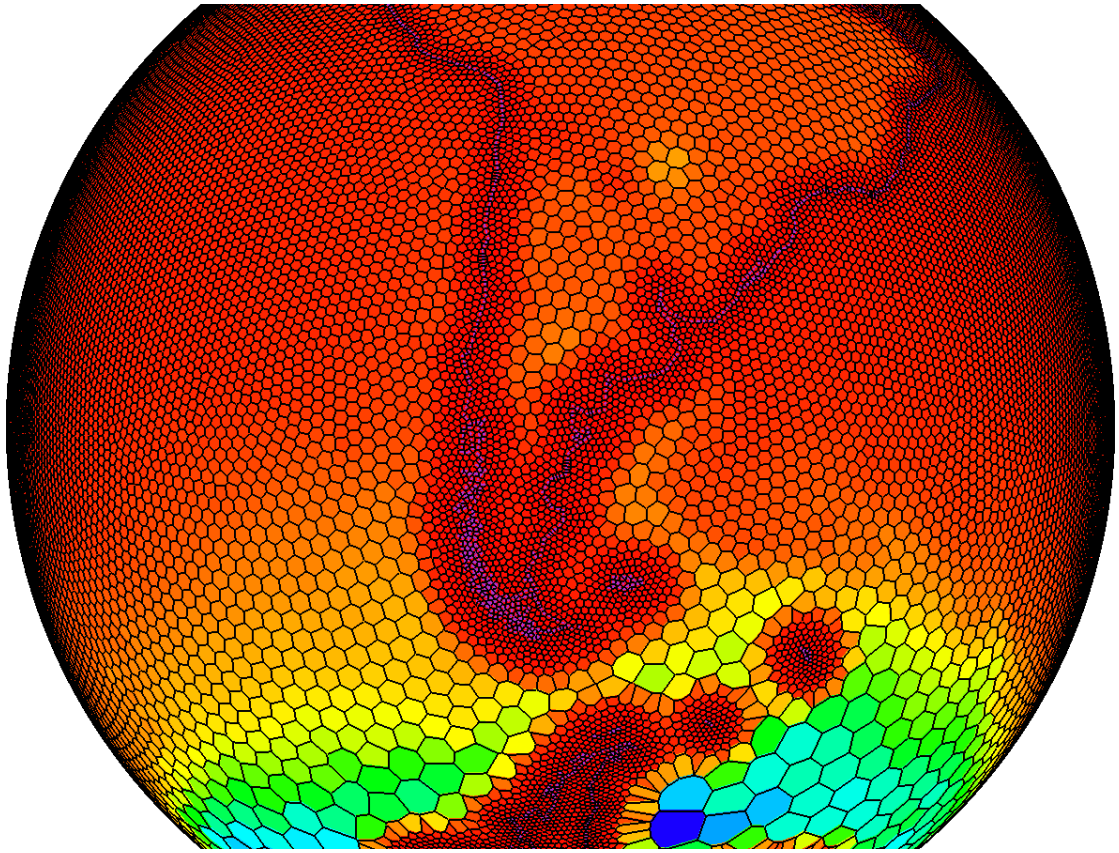
OVERVIEW: Geometry

Computational geometry allows us to control how we choose sample points from a region, and how we combine those points into triangles, in order to carry out an analysis.



OVERVIEW: Geometry

Computational geometry helps us create a model of a curved surface, and shows us how to refine our grid near transition zones.



OVERVIEW: Geometry

The objects we just saw were broken into simpler objects. This is the basis of computational geometry.

To do all these wonderful things with computational geometry, we go back to the basic geometric objects, figure out how to implement them on a computer, and then teach the computer, step by step, the rules that allow us to assemble and understand more complicated objects.

Over two weeks, we will be lucky to study some properties of points, lines, triangles, and triangular meshes.

Computational geometry can easily fill a semester of study.

OVERVIEW: Geometric Tasks

We will sample common computational geometry tasks:

- measurement of length, area, volume, distance, direction, orientation;
- discretization of curves and surfaces;
- indexing or parameterizing the parts of an object;
- 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.

Geometry Algorithms

- Overview
- **The Points on a Line**
- Points NOT on a Line
- Estimating Integrals over an Interval
- Triangles and their Properties
- Triangulating a Polygon
- The Convex Hull
- Triangulating a Point Set by Delaunay
- Estimating Integrals over a Triangle
- Conclusion

POINTS: Locations

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. For example, the coordinates of a 3D point might be described as (x, y, z); If we have several points, we subscript the coordinates, so that z_2 would be the z coordinate of the second point.

Computationally, it is preferable to store the coordinates of a point in a single variable; for instance, $p = [x, y, z]$.

MATLAB allows a vector to be row or column vector. I prefer points to be described as column vectors. In that case, we'd write either $p = [x; y; z]$ or else $p = [x, y, z]'$.

POINTS: What is a Line?

A line is the infinite set of points which...how do we explain it? We know what a line is when we look at one, but that's not good enough.

We know that two points, say p_1 and p_2, determine a line; we might say the line goes through one point and towards the other. So any point on the line might be described by the process of starting at p_1 and going "in the direction" of p_2. What exactly is that direction?

You might recall that a geometric direction corresponds to a mathematical vector, (not a computational one!) and that typically a vector is determined by the *difference* between two points. To get to p_2 starting from p_1 the direction vector is

$$\vec{v}_{p1,p2} = p_2 - p_1$$

POINTS: What is a Line?

Every point p on the line can be found by starting at p_1 and moving in the direction from p_1 to p_2.

Let's use s to indicate *how far* we move along that direction.

Given any value of s, the corresponding point $p(s)$ is:

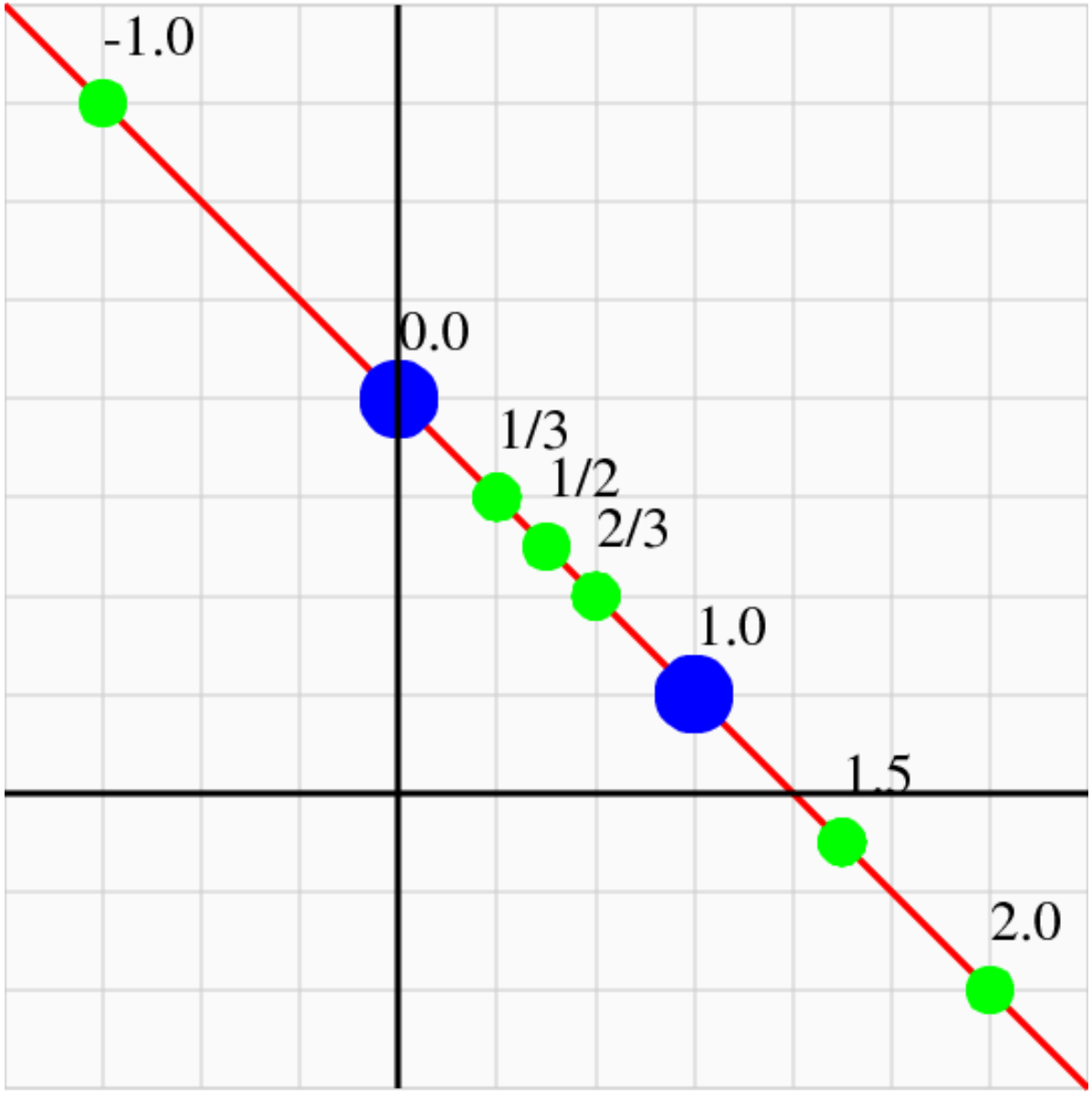
$$\begin{aligned} p(s) &= p_1 + s * \vec{v}_{p1,p2} \\ &= p_1 + s * (p_2 - p_1) \\ &= (1 - s) * p_1 + s * p_2 \end{aligned}$$

POINTS: Examples

Choose $p_1 = (0,4)$, and $p_2 = (3,1)$. Then $\vec{v}_{p1,p2} = (3,-3)$.

```
 s    p1      + s    * p2-p1   =      p
---   ----    ----     ------    ---------
-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  )
```

POINTS: P=P1+S*(P2-P1)



POINTS: Advantages of a Formula

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.

POINTS: The S Coordinate

The value of the s coordinate contains useful information:

- It indicates that every line is really a one dimensional object, because its points can be indexed by a single number;
- Points with $0 \leq s \leq 1$ are between $p1$ and $p2$; if we restrict ourselves to these points, we have a line segment;
- Points with $s < 0$ or $1 < s$ are to the "left" or "right";
- If we know how far p_2 is from p_1, that is, $||p2 - p1||$, then s measures the (signed) distance from $p1$ to the point $p(s)$ in units of that basic distance.

When we get to triangles, we will see a similar coordinate system.

POINTS: Distance

Given points p_1 and p_2 and a value s, it's easy to determine the (x, y) coordinates of the point $p(s)$ from our formula.

But suppose we start with the (x, y) coordinates of a point p that we know is on the line. Can we determine s?

One approach uses the distance function $d(*, *)$ as follows:

The value of s is the (signed) ratio of the distance d from p_1 to p relative to the distance from p_1 to p_2.

$$d(p1, p2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

$$d(p1, p) = \sqrt{(x - x_1)^2 + (y - y_1)^2}$$

$$s = \pm \frac{d(p1, p)}{d(p1, p2)}$$

and we can figure out the plus or minus sign.

POINTS: Dot Product

A better way uses the dot product of two vectors:

$$\vec{v}_1 \cdot \vec{v}_2 = ||\vec{v}_1|| * ||\vec{v}_2|| * \cos(\alpha)$$

where α is the angle between the vectors, and

$$\vec{v}_1 \cdot \vec{v}_1 = ||\vec{v}_1||^2$$

This means that if we have set $\vec{v}_1 = \overrightarrow{p_2 - p_1}$ and $\vec{v}_2 = \overrightarrow{p - p_1}$, then:

$$\begin{aligned} \frac{\vec{v}_1 \cdot \vec{v}_2}{\vec{v}_1 \cdot \vec{v}_1} &= \frac{||\vec{v}_1|| * ||\vec{v}_2|| * \cos(\alpha)}{||\vec{v}_1||^2} \\ &= \frac{||\vec{v}_2|| * \cos(\alpha)}{||\vec{v}_1||} \\ &= s \end{aligned}$$

And now $cos(\alpha)$ gives us the sign of s.

POINTS: Example

With $p1 = (0, 4)$, and $p2 = (3, 1)$, let $p = (-2, 6)$.

Distance method:

$$\begin{aligned} d(p1, p2) &= \sqrt{9 + 9} = \sqrt{18} = 3\sqrt{2}. \\ d(p1, p) &= \sqrt{4 + 4} = \sqrt{8} = 2\sqrt{2} \\ s &= \pm \frac{2\sqrt{2}}{3\sqrt{2}} = -\frac{2}{3} \end{aligned}$$

Dot product:

$$\begin{aligned} v1 &= p2 - p1 = (3, -3) \\ v2 &= p - p1 = (-2, 2) \\ s &= \frac{\vec{v}_1 \cdot \vec{v}_2}{\vec{v}_1 \cdot \vec{v}_1} = \frac{-6 - 6}{9 + 9} = -\frac{12}{18} = -\frac{2}{3} \end{aligned}$$

POINTS: MATLAB Computation of S

```
function s = line_parameter_s ( p1, p2, p )

%  P1 and P2 must be column vectors!
%  P is a column vector.

  v1 = ( p2 - p1 );
  v2 = ( p  - p1 );
  s  = ( v1' * v2 ) / ( v1' * v1 );

  return
end
```

POINTS: Interpretation of S

It's natural to think of s as the distance of the point p from $p1$.

However, that can't be quite right for two reasons:

- **scale**: the s coordinate of $p2$ is 1, no matter how far it is;
- **sign**: some points have negative s. Distance is never negative!

The s parameter is actually an *signed, relative* distance.

To compute the true distance, take the absolute value of s and multiply by the distance from $p1$ to $p2$:

```
distance(p,p1) = | s(p) | * distance (p2,p1)
```

(Of course, if we have the coordinates of $p1$ and p, we can get the distance directly.)

POINTS: MATLAB Computation of S

This works fine if we type

```
p1 = [0;4];  p2 = [3;1]; p = [2;2];
s = line_parameter_s ( p1, p2, p );
```

but it will fail if we try to squeeze in several points at once!

```
  p = [2,2; -3,7; 1.5,2.5; 6,-2]';
??? Error using ==> minus
Matrix dimensions must agree.

Error in ==> line_parameter_s at 22
  v2 = ( p  - p1 );
```

But there's no real reason the function can't do all these calculations at once.

And MATLAB always encourages us to write programs that handle data in batches, rather than one data item at a time.

POINTS: MATLAB Computation of S

```
function s = line_parameter_s ( p1, p2, p )

%  P1 and P2 must be column vectors!
%  P is a column vector or array of columns.

[ m, n ] = size ( p );                  <--Need N.
v1 = ( p2 - p1 );
p1_array = repmat ( p1, 1, n );  <--[ P1 | P1 | ... P1]
v2 = ( p  - p1_array );                 <--Subtract arrays.
s  = ( v1' * v2 ) / ( v1' * v1 );<--S is now a vector

return
end
```

POINTS: Accidentally Try 3D

By the way, why didn't we have to specify that $p1$, $p2$ and p were 2 dimensional points?

Ah, we don't specify vector sizes (or shapes) because MATLAB can figure that out (which leads to bad programming habits when you try to do things in C or Fortran!).

What would happen if we accidentally put in 3D points?

```
s = line_parameter_s ( [1;5;3], [4;2;9], [2;4;5] )
0.3333
```

It still gives an answer...and it's correct! QUIZ: How can we check?

Well, what will break our algorithm?

How about if the point p is not on the line?

Geometry Algorithms

- Overview
- The Points on a Line
- **Points NOT on a Line**
- Estimating Integrals over an Interval
- Triangles and their Properties
- Triangulating a Polygon
- The Convex Hull
- Triangulating a Point Set by Delaunay
- Estimating Integrals over a Triangle
- Conclusion

OFFLINE: Distance for Off-Line Point

We started out by assuming that our point p was already on the line. Presumably, our formulas will "break" otherwise. Let's go ahead, though, and feed in the point $q = (7, 2)$:

Distance method:

$$\begin{aligned} d(p1, p2) &= \sqrt{9+9} = \sqrt{18} = 3\sqrt{2}. \\ d(p1, q) &= \sqrt{49+4} = \sqrt{53} = \\ \text{"}s\text{"} &= \pm \frac{\sqrt{53}}{3\sqrt{2}} \end{aligned}$$

This, it turns out, is the s coordinate of the point on the line which is just as far away as point q. That is, they both lie on a circle of radius s centered at $p1$.

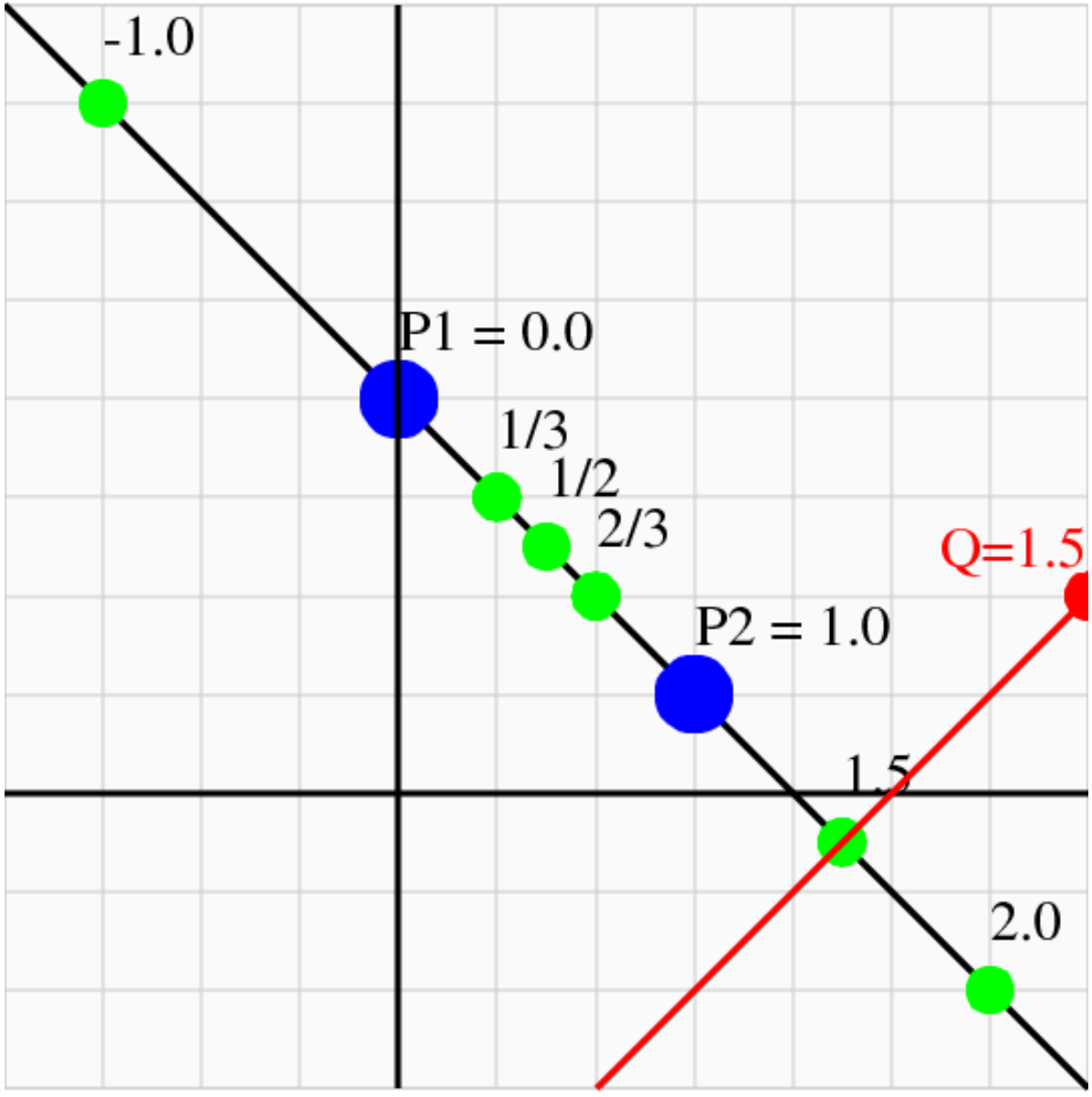
OFFLINE: Dot Product for Off-Line Point

Now let's try the dot product:

$$
\begin{aligned}
v1 &= p2 - p1 = (3, -3) \\
v2 &= q - p1 = (7, -2) \\
\text{"}s\text{"} &= \frac{\vec{v}_1 \cdot \vec{v}_2}{\vec{v}_1 \cdot \vec{v}_1} = \frac{+21 + 6}{9 + 9} = \frac{27}{18} = \frac{3}{2}
\end{aligned}
$$

This is much more interesting. It is the s coordinate of the point on the line that is the nearest to q! That's an interesting geometric computation!

OFFLINE: Dot Product for Off-Line Point



OFFLINE: Distance from Point to Line

Using the dot product, we have been able to compute an s coordinate for all points in the plane, in a way that is very similar to Cartesian coordinates.

Now if we only know the s coordinate of a point, we still have a lot of points to choose from. It sure would be nice to have a t coordinate that would answer that question. Now t should be zero if the point is on the line, and I guess it should be 1 if the point is 1 unit away, and so on.

So one way to go is to start with q, compute s, find the corresponding nearest point p on the line, and then compute t as the distance from p to q.

```
s = line_parameter_s ( p1, p2, q );
p = p1 + s * ( p2 - p1 );
t = norm ( p - q );
```

OFFLINE: A Perpendicular Axis

Now two vectors are *perpendicular* if their dot product is zero.

Given a (nonzero) 2D vector $\vec{v} = (v_x, v_y)$, one vector that is perpendicular to $\vec{v}$ is $\vec{w} = (-v_y, +v_x)$:

$$\vec{v} \cdot \vec{w} = v_x * w_x + v_y * w_y = -v_x * v_y + v_y * v_x = 0$$

Now let $\vec{v}$ be the unit direction vector of the line, that is,

$$\vec{v} = (p2 - p1)/||p2 - p1||$$

and use the formula to define a vector $\vec{w}$ perpendicular to $\vec{v}$. Moreover, let's normalize $\vec{w}$ to have unit length.

The two points on the line already defined the direction vector $\vec{v} = p2 - p1$, which played the role of the x axis.

Now we have a perpendicular direction $\vec{w}$, just like a y axis.

OFFLINE: Distance from Point to Line

A nonzero dot product equals the product of the vector lengths and the cosine of the angle between them.

Let q be a point not on the line, and compute the dot product of $\vec{w}$ with the direction vector $\overrightarrow{q-p}$:

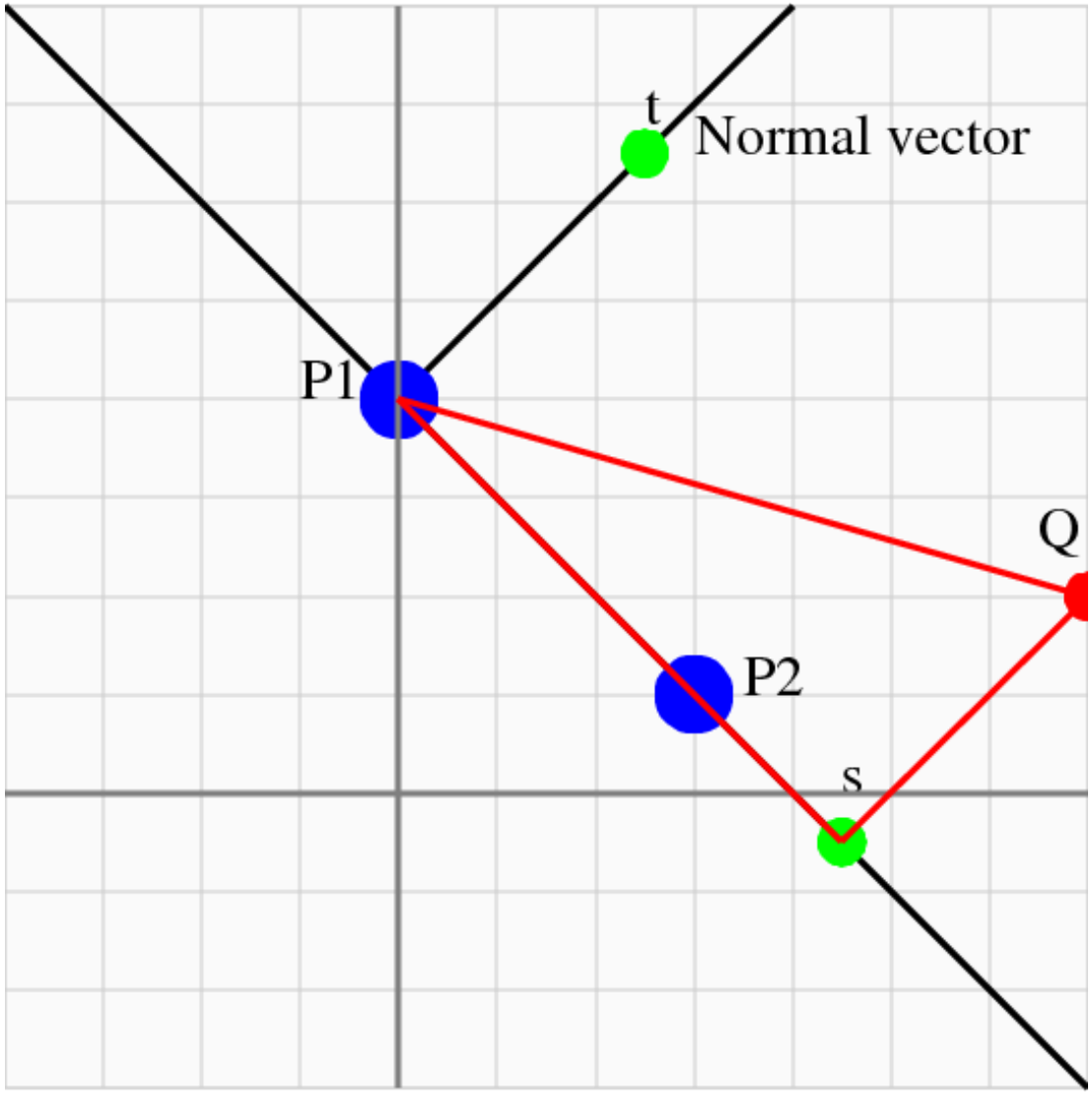
$$\vec{w} \cdot \overrightarrow{q - p1} = ||w||\,||q - p1||\,\cos(\alpha) = ||q - p1||\,\cos(\alpha) = t$$

If q is actually on the line, then t is zero. Otherwise, t is the (signed) length of the side of a right triangle with vertex $p1$, hypotenuse $q - p1$, and adjacent side that is part of the line through $p1$ and $p2$.

Our new S and T axes decompose the vector $\overrightarrow{q-p}$ into parallel and perpendicular parts. They almost behave like x and y axes, except that we never properly scaled distances in the S direction.

So now, let us define a normalized coordinate $\hat{s} = s/||p2 - p1||$.

OFFLINE: Distance of Point to Line



OFFLINE: Distance from Point to Line

Our unit direction vectors $\vec{v}$ and $\vec{w}$ satisfy:

$$\begin{aligned}\vec{v} \cdot \vec{w} &= 0 \\ ||\vec{v}|| &= 1 \\ ||\vec{w}|| &= 1\end{aligned}$$

Any point q defines a vector $\overrightarrow{q - p1}$, and we can compute its $\hat{s}$ and t coordinates using the formulas of decomposition:

$$s = \overrightarrow{q - p1} \cdot \vec{v}$$

$$t = \overrightarrow{q - p1} \cdot \vec{w}$$

and formulas of composition:

$$\overrightarrow{q - p1} = \hat{s} * \vec{v} + t * \vec{w}$$

$$||\overrightarrow{q - p1}||^2 = \hat{s}^2 + t^2$$

so we have an orthonormal $\hat{S}$, T system with origin at $p1$.

OFFLINE: MATLAB Computation of T

```
function t = line_parameter_t ( p1, p2, p )

%  P1, P2, P must be column vectors;
%  The spatial dimension must be 2.

v1 = ( p2 - p1 );
v2 = ( p  - p1 );
nv = [ -v1(2); v1(1) ];
nv = nv / norm ( nv );
t = nv' * v2;

return
end
```

OFFLINE: Check

```
>> p1 = [ 0; 4]; p2 = [3;1]; q = [7;2];
>> t = line_parameter_t ( p1, p2, q )
t = 3.5355

>> s = line_parameter_s ( p1, p2, q )
s = 1.5000

>> p3 = p1 + s * ( p2 - p1 )
p3 =[ 4.5000; -0.5000]

>> norm ( q - p3 )
ans = 3.5355
```

QUIZ:: Can t be negative? What does this mean?

OFFLINE: A MATLAB Function

If the vector $\vec{w}$ is a unit normal vector, then so is $-\vec{w}$, and if we had used that vector instead, all our t values would switch sign.

So the sign of the t parameter is arbitrary (we usually use the right hand rule to pick $\vec{w}$). But whichever normal vector we choose, the sign of t divides all points into three classes: to the left of the line, on the line, or to the right.

It's easy to make a simplified copy of **line_parameter_t(p1,p2,p)** called **line_side(p1,p2,p)** returning:

- $+1$ if p is to the left of the line;
- 0 if p is on the line;
- -1 if p is to the right of the line.

Such an orientation function is handy when we look at triangles!

OFFLINE: Summary

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

- *parameterization*: where is the point p on the line;
- *containment*: is the point p on the line from $p1$ to $p2$?
- *distance*: how far is the point q from the line?
- *nearness*: which point p on the line is nearest the point q?
- *orientation*: which side of the line is offline point q?
- *parameterization*: where is the offline point q?
- *parallel and orthogonal*: how we moved along the line, or searched for nearest points.
- *mapping*: we established a relationship between (x, y) and $(\hat{s}, t)$ coordinate systems.

OFFLINE: Summary

Think about the algorithms we have encountered:

- *distance*: how far is the point q from the line? ($|t|$)
- *containment*: is p on the line through $p1$ and $p2$? (is t 0?)
- *parameterization*: where is p on that line? (use parameter s)
- *containment*: is p between $p1$ and $p2$? ($0 <= s <= 1$)
- *nearness*: which point p on the line is nearest to the offline point q?; (use s coordinate of p)
- *orientation*: which side of the line is offline point q? (sign of t)
- *parameterization*: where is the offline point q; (s and t tell us)
- *parallel and orthogonal*: the (s, t) coordinate system follows the parallel and orthogonal directions of the line.

Geometry Algorithms

- Overview
- The Points on a Line
- Points NOT on a Line
- **Estimating Integrals over an Interval**
- Triangles and their Properties
- Triangulating a Polygon
- The Convex Hull
- Triangulating a Point Set by Delaunay
- Estimating Integrals over a Triangle
- Conclusion

INTEGRALS: Random Samples

Monte Carlo algorithms work by sampling a set of data.

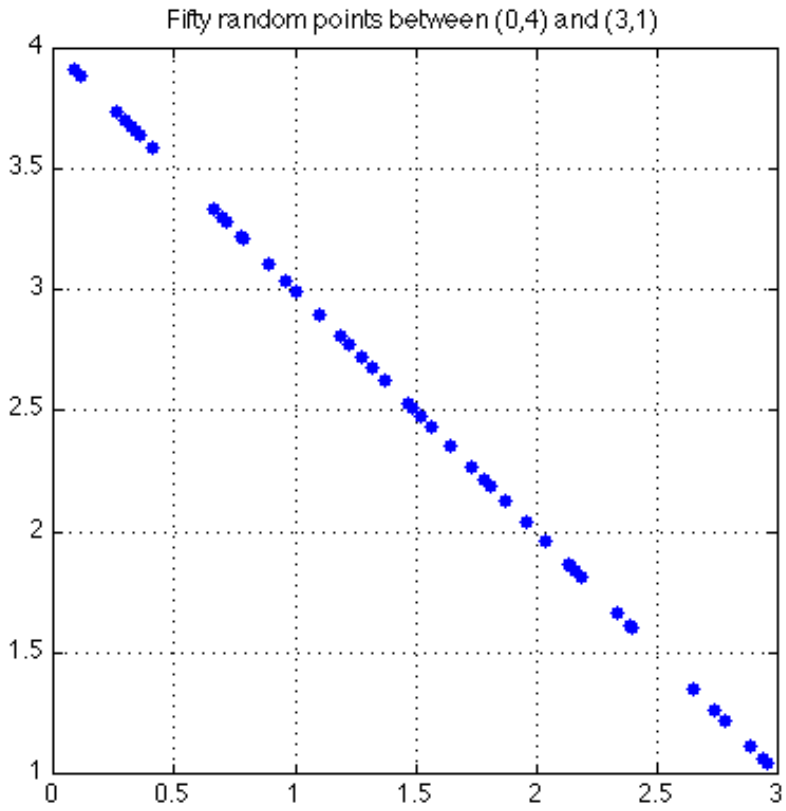
A Monte Carlo method for estimating the integral of a function over the 1D interval $p1 \leq x \leq p2$ would need to pick points randomly over this line segment.

Luckily, our way of describing a line as points $p(s)$ fits this perfectly. Not only do we have a formula for choosing points, but the points between $p1$ and $p2$ are exactly those for which $0 \leq s \leq 1$. That means we can call our random number generator and take its output to be the values of s that determine our points.

This works the same whether our interval is 1D, or for some reason is a line in 2D or higher dimensions.

INTEGRALS: Random Samples in 2D

Fifty random values between P1=[0,4] and P2=[3,1]:



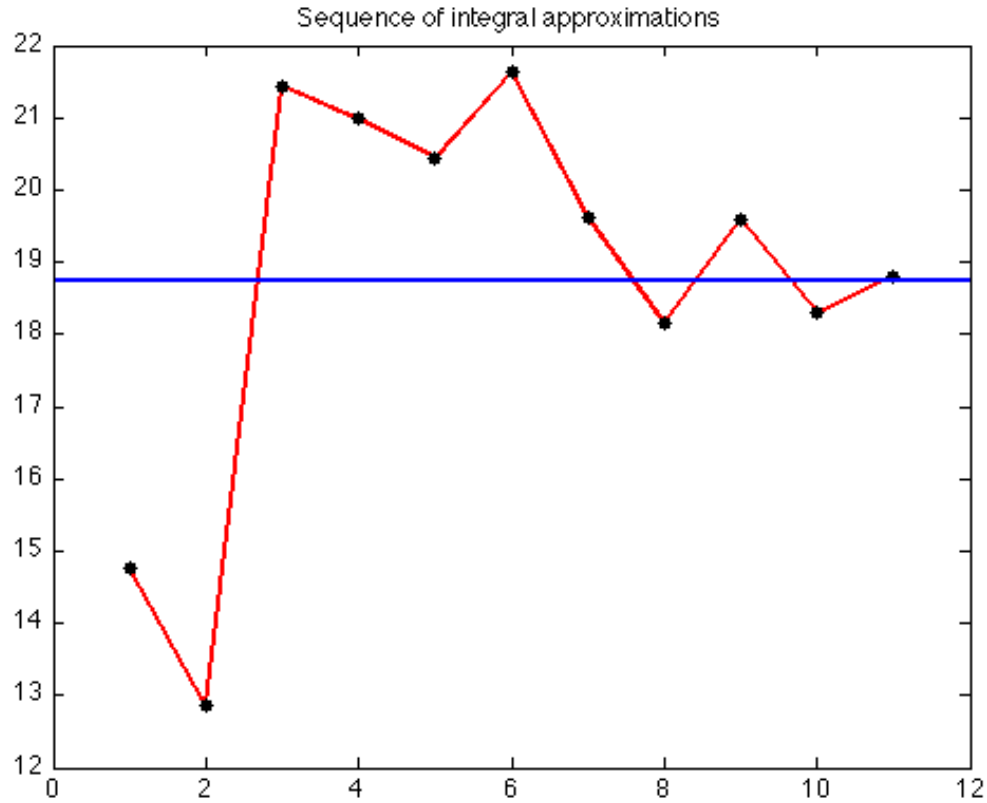
INTEGRALS: Monte Carlo Line Integration

This code can estimate the integral of $f(x)$ from $p1$ to $p2$:

```
function q = line_integral ( n )
  p1 = 0.0;
  p2 = 3.0;
  svec = rand ( n, 1 );
  x = p1 + svec * ( p2 - p1 );
  fvec = x .* x .* ( 4 - x ) + 1;
  q = ( p2 - p1 ) * sum ( fvec ) / n;
  return
end
```

INTEGRALS: Random Samples

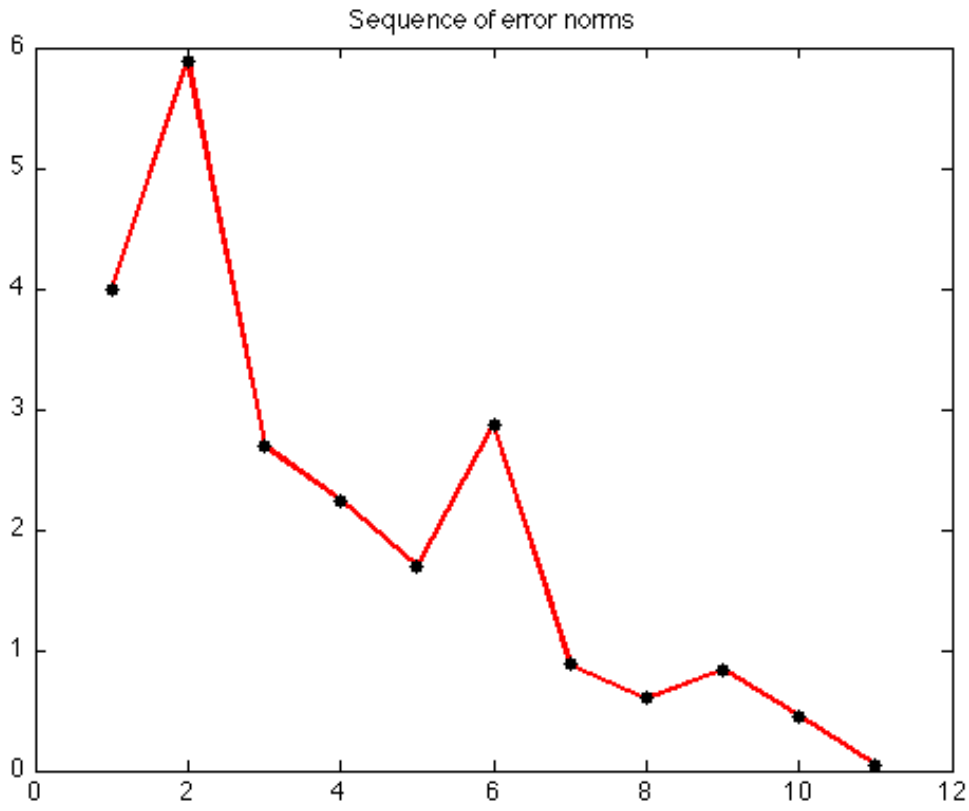
The errors tend to decrease with the sample number.



These estimates were made with 1, 2, 4, 8, ..., 1024 samples.

INTEGRALS: Random Samples

The trend is clearer for the absolute value of the error.



INTEGRALS: Random Samples

The Monte Carlo method works in part by being able to rapidly compute uniform random sample points from the line segment $[p1, p2]$.

It's easy to compare Monte Carlo results with exact integration in this case.

But when the region is a triangle, the interior of an ellipse, or the surface of a sphere...or even a teapot, exact integration techniques are not available. So ideas like the Monte Carlo sampling method will be extremely useful.

This has also been our first example of a computational algorithm that involves sampling.

Geometry Algorithms

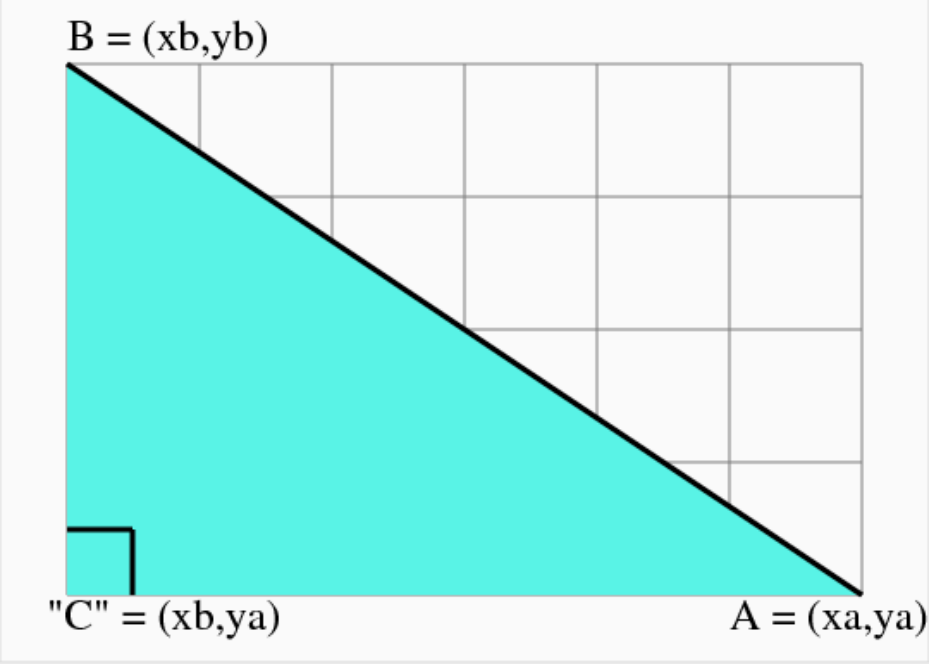
- Overview
- The Points on a Line
- Points NOT on a Line
- Estimating Integrals over an Interval
- **Triangles and their Properties**
- Triangulating a Polygon
- The Convex Hull
- Triangulating a Point Set by Delaunay
- Estimating Integrals over a Triangle
- Conclusion

TRIANGLES: Triangle SIDES?

To measure a right triangle side that goes from A=(xa,ya) to B=(xb,yb), we take advantage of the theorem of Pythagoras:

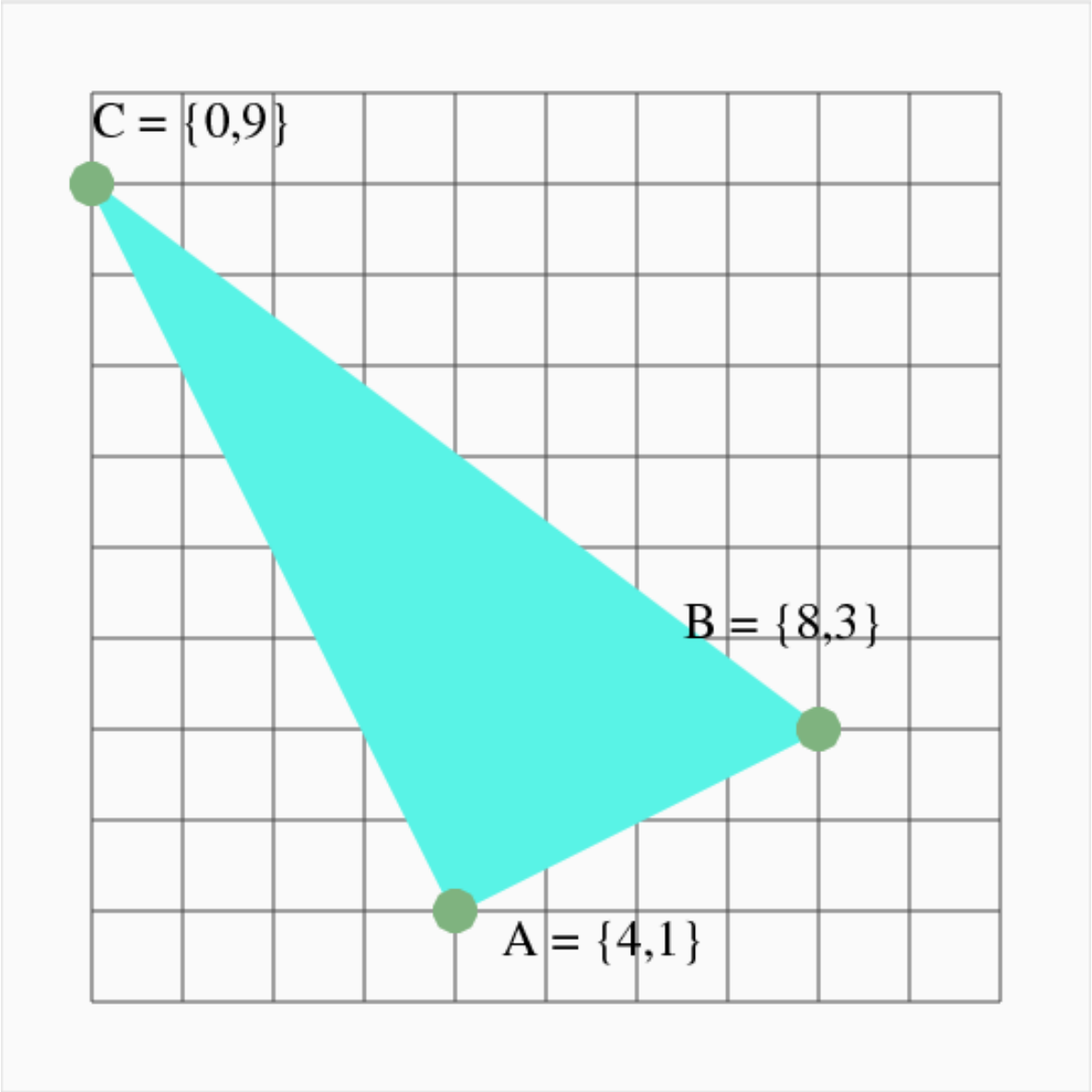
```
AB^2 =          AC^2          + BC^2
     =       ( A - C )^2      + ( B - C )^2
     =       ( xa - xb )^2 + ( yb - ya )^2

AB   = sqrt ( ( xa - xb )^2 + ( yb - ya )^2 )
```



TRIANGLES: Triangle SIDES?

QUIZ: What are the sides of our example triangle?



TRIANGLES: Triangle ANGLES?

Let α, β and γ be the angles at vertices A, B and C. The law of cosines for angle α says:

$$BC^2 = AB^2 + AC^2 - 2 * AB * AC * cos(\alpha)$$

Formulas for all three angles are:

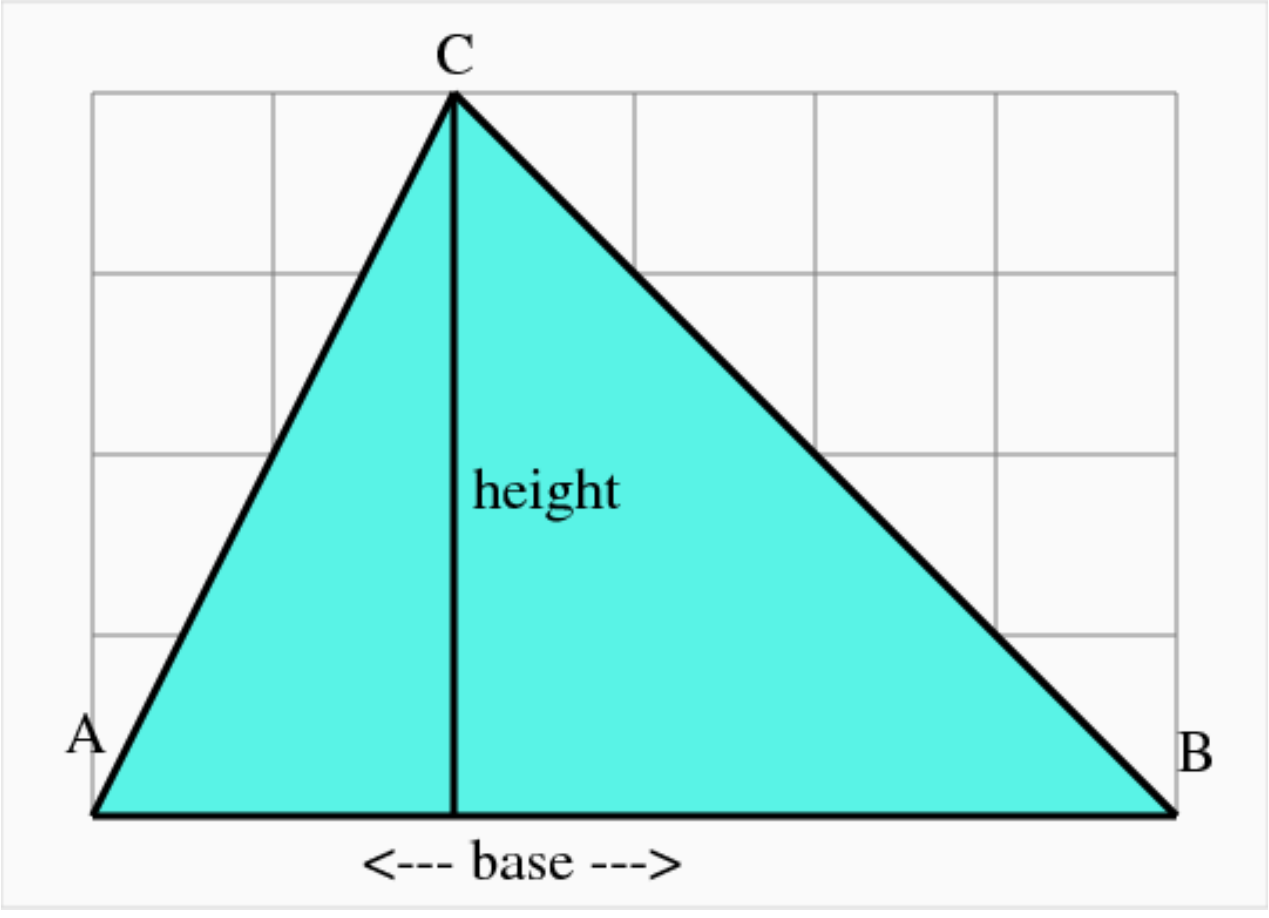
$$\alpha = \cos^{-1}\left(\frac{AB^2 + AC^2 - BC^2}{2 * AB * AC}\right)$$
$$\beta = \cos^{-1}\left(\frac{AB^2 + BC^2 - AC^2}{2 * AB * BC}\right)$$
$$\gamma = \cos^{-1}\left(\frac{AC^2 + BC^2 - AB^2}{2 * AC * BC}\right)$$

QUIZ: What are the angles of our example triangle?

TRIANGLES: Triangle AREA?

In high school geometry, the formula for a triangle is simple:

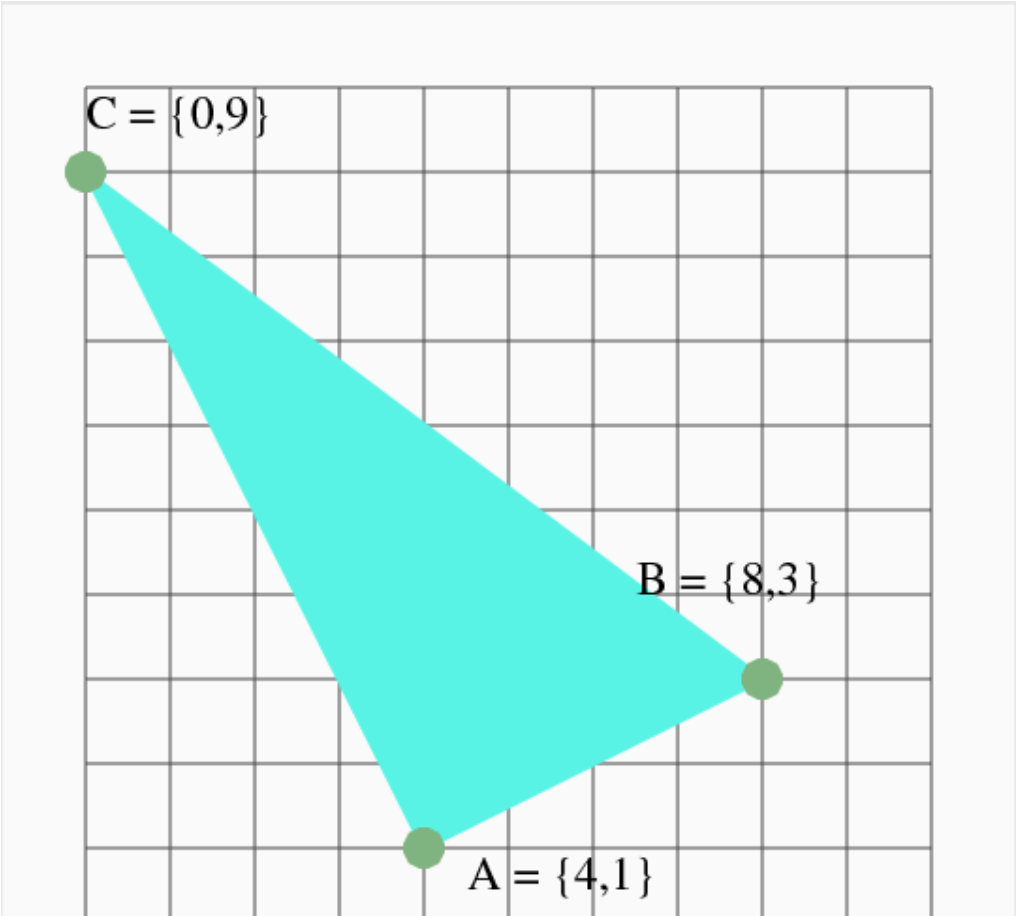
$$\text{Area} = \frac{1}{2} * \text{base} * \text{height}$$



TRIANGLES: Triangle AREA?

Such a formula doesn't help when we don't have a ruler and a protractor, but rather a set of coordinates:

```
T = [ A, B, C ] = [ xa, xb, xc ] = [ 4, 8, 0 ]
                  [ ya, yb, yc ]   [ 1, 3, 9 ]
```



TRIANGLES: Formulas for AREA

$$T = [A, B, C] = \begin{matrix} [xa, xb, xc] \\ [ya, yb, yc] \end{matrix} = \begin{matrix} [4, 8, 0] \\ [1, 3, 9] \end{matrix}$$

$$\begin{aligned} Area = (1/2) * | & (xa * (yb - yc) \\ & - xb * (yc - ya) \\ & + xc * (ya - yb)) | \end{aligned}$$

$$= \frac{1}{2} * | \det \begin{vmatrix} xa & xb & xc \\ ya & yb & yc \\ 1 & 1 & 1 \end{vmatrix} |$$

$$= 1/2 * | (A - C) X (B - C) |$$ *(vector cross product)*

Quiz: What is the area of our example?

TRIANGLES: Signs and Orientation

Why did we need absolute values in each of the area formulas?

It has to do with the *order* of the vertices of the triangle. Given in clockwise order, the basic area formulas give a negative result. Counterclockwise (CCW) order gives a positive result.

It is a mathematical convention to list triangle vertices in CCW order, in which case we can drop the absolute values.

On the other hand, we have also found something interesting:

```
triangle orientation = sign of ( A - C ) X ( B - C )
```

Quiz: What is the orientation of our example?

TRIANGLES: Triangle CONTAINS Point?

We now come to a question that doesn't seem to involve a computation, namely, is a point **P** inside or outside of triangle **T**?

We can answer this question immediately if we draw a picture. However, we need to find a way of answering this question that is an algorithm, that is, *computational* and *automatic*.

If we can answer this question, we will have an idea how to answer the same question in 3D (for a tetrahedron) and higher, abstract dimensions, where drawing a picture would be out of the question.

So even though we think our eye can answer this question, we don't want to have to use our eye to examine a million cases, or to "look" in 4D. A general algorithm will come in very handy!

TRIANGLES: A Walk Around the Block

Our triangle vertices are in clockwise order. Walking from vertex **A** to **B**, **C** and on to **A**, what can we say about our left hand?

It's always pointing into the triangle. Moreover, any point inside the triangle will always be to our left. And any point in the triangle will be "to the left of" the edges **AB**, **BC**, and **CA**.

And if a point is *not* inside the triangle, what can we say? It might be to the left of one or two of the sides, but it will always be to the right of at least one side.

Triangle Contains Point

A point **P** is inside a triangle **T** if, and only if, it is "to the left" of all three sides of **T**.

TRIANGLES: A MATLAB Function

Recall our function for the location of a point relative to a line.

The value returned by **line_side(A,B,P)** is:

- +1 if **P** is to the left of the line through **AB**;
- 0 if **P** is on the line through **AB**;
- -1 if **P** is to the right of the line through **AB**.

Suppose we call this function three times, to check the status of **P** with respect to the lines through the segments **AB**, **BC** and **CA**!

TRIANGLES: QUIZ

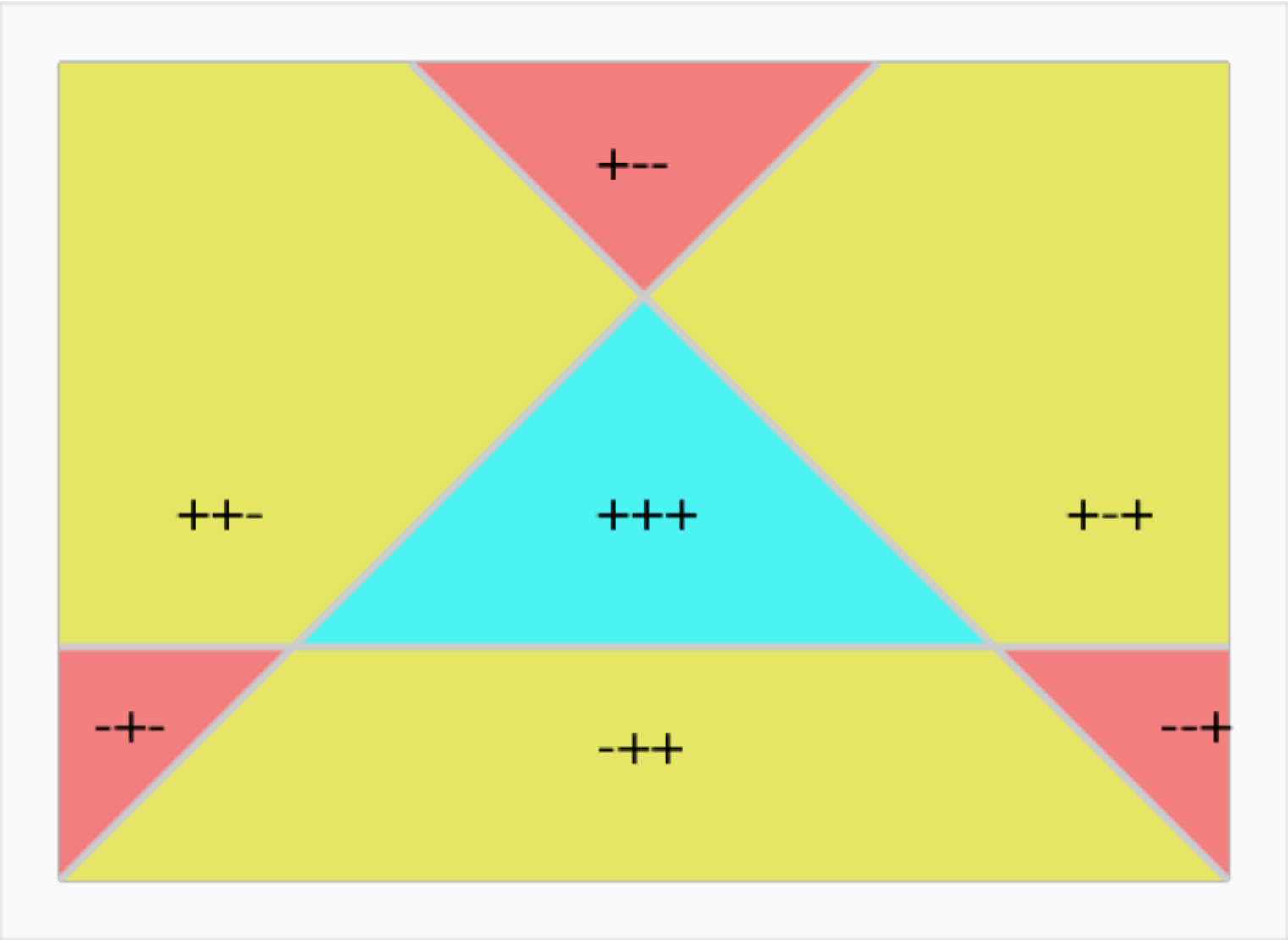
Interpret these results:

	AB	BC	CA	Meaning?
P1	+1	+1	+1	
P2	-1	+1	+1	
P3	+1	-1	+1	
P4	+1	+1	-1	
P5	+1	-1	-1	
P6	-1	+1	-1	
P7	-1	+1	+1	
P8	-1	-1	-1	

There is something a little odd about one result.

TRIANGLES: Triangle Orientation

What we can tell from the +/- signs of the three measurements:

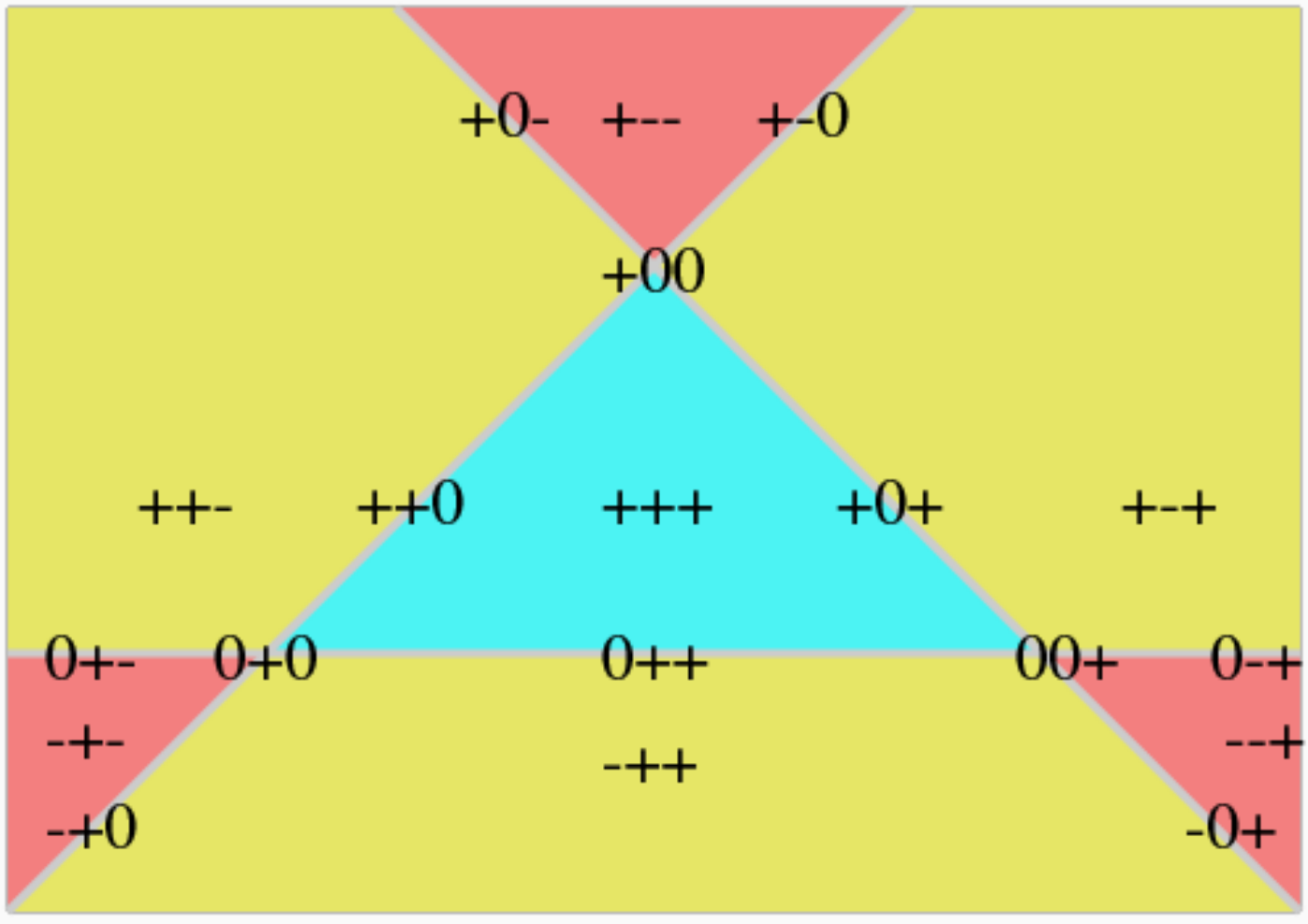


TRIANGLES: We could also check for zero values

	AB	BC	CA	Meaning?
P9	0	+1	+1	inside, but on side AB
P10	+1	0	+1	inside, but on side BC
P11	+1	+1	0	inside, but on side CA
P12	+1	0	0	what is this?
P13	0	+1	0	
P14	0	0	+1	
P15	0	+1	-1	where is this?
...				
P21	-1	0	0	why can't this happen?
...				
P27	0	0	0	how could this be possible?

TRIANGLES: Triangle Orientation

What we can tell if we include 0:



TRIANGLES: MATLAB Line_Side

```
function side = line_side ( p1, p2, p )

  v1 = ( p2 - p1 );
  v2 = ( p  - p1 );
  nv = [ - v1(2); + v1(1) ]; <--counterclockwise!
  t = nv' * v2;  <-- we don't need to normalize nv.

  side = ( 0 <= t );

  return
end
```

TRIANGLES: MATLAB Triangle_Contains

```
function contains = triangle_contains ( t, p )

  contains = line_side ( t(1:2,1), t(1:2,2), p ) && ...
             line_side ( t(1:2,2), t(1:2,3), p ) && ...
             line_side ( t(1:2,3), t(1:2,1), p );

  return
end
```

TRIANGLES: Test Triangle_Contains

```
function triangle_contains_test ( )

  t = [ 4, 0; 3, 4; 0, 1 ]';

  p = [ 1, 4; 2, 1; 2, 3; 3, 2; 3, 4; 3, 5; 4, 1; 4, 5 ]';

  for j = 1 : 8
    c = triangle_contains ( t, p(1:2,j) );
    fprintf ( 1, '  %10f  %10f  %2d\n', p(1:2,j), c );
  end

  return
end
```

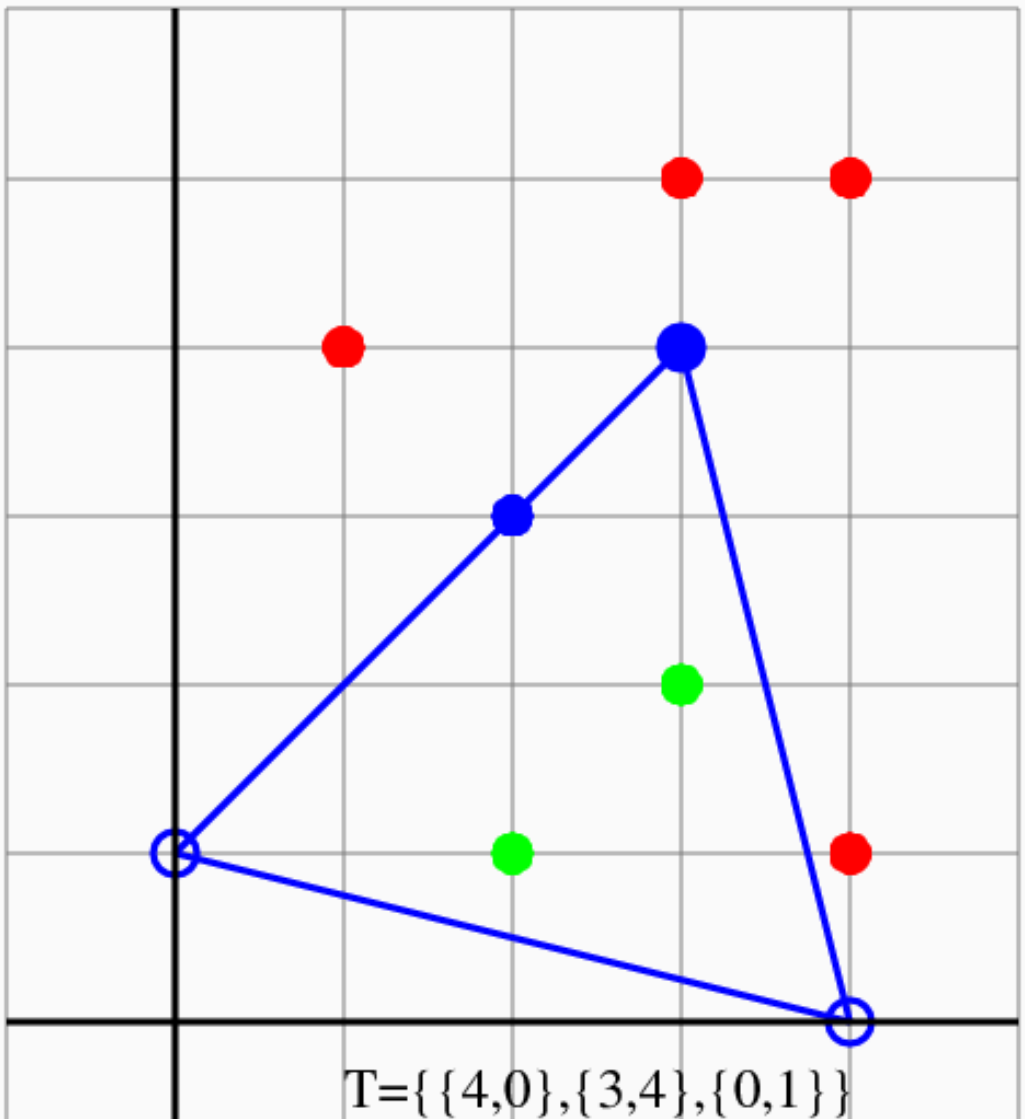
TRIANGLES: Test Triangle_Contains

```
triangle_contains_test ( )

     X             Y          C

  1.000000      4.000000     0
  2.000000      1.000000     1
  2.000000      3.000000     1
  3.000000      2.000000     1
  3.000000      4.000000     1
  3.000000      5.000000     0
  4.000000      1.000000     0
  4.000000      5.000000     0
```

TRIANGLES: A Picture of Our Test



TRIANGLES: Centroids

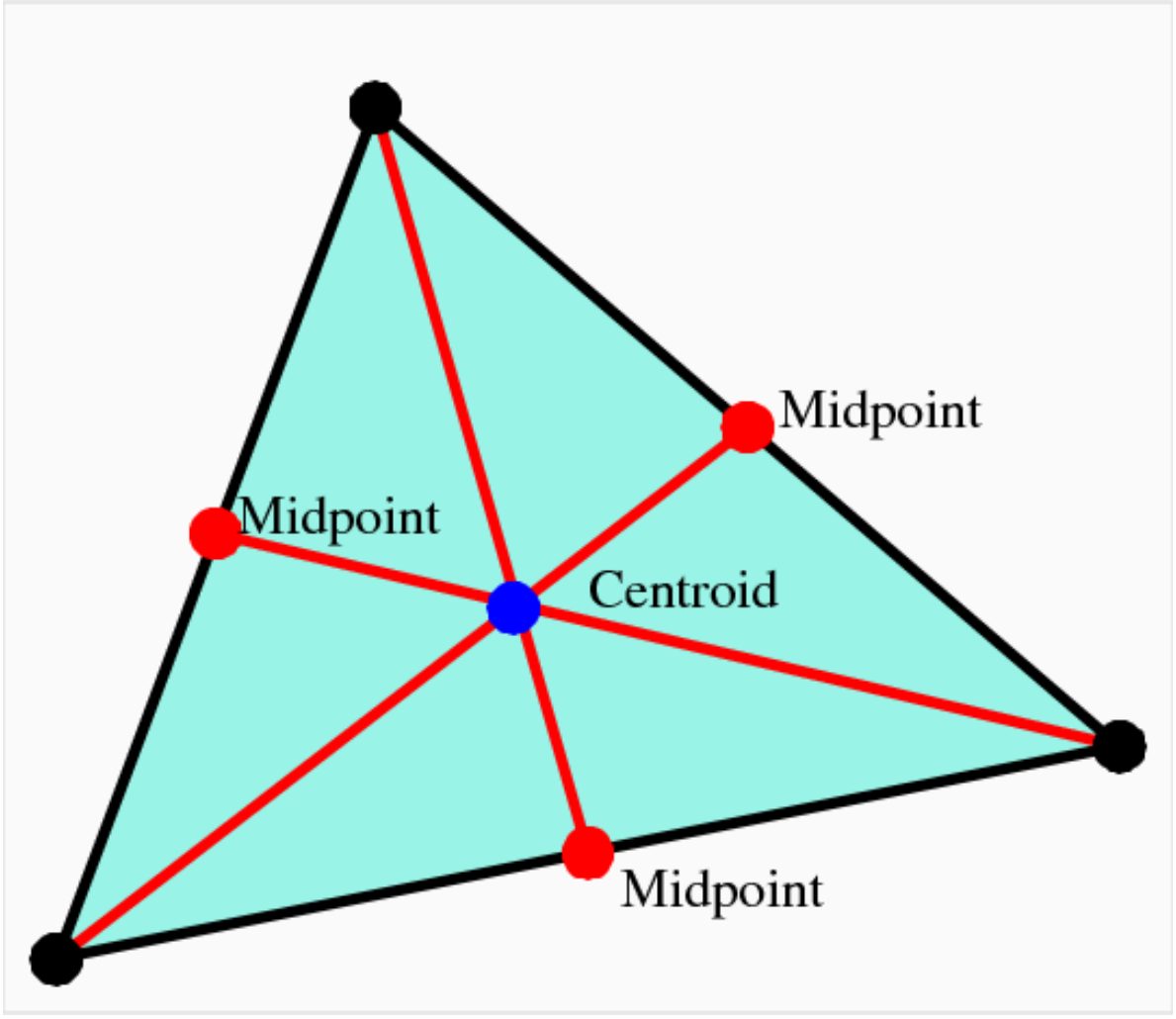
The centroid or, loosely speaking, the "center of mass", of a triangle, is the unique point **CM** with the property that *any* line through **CM** divides the triangle into two pieces of equal area.

You can locate the centroid in a drawing of a triangle very easily: Connect each edge midpoint to the opposite vertex. All three lines intersect at the centroid.

It sounds like computing the centroid will be hard (determine formulas of lines, compute intersections), but for a triangle with vertices **A**, **B**, and **C**, the formula for the centroid is amazingly simple:

$$CM = \frac{A + B + C}{3}$$

TRIANGLES: A Picture of the Centroid Calculation



TRIANGLES: The Centroid Calculation

```
A = ( 0,  0)
B = (10,  2)
C = ( 3,  8)

Mid(AB) = (A+B)/2   = (  5,   1)
Mid(BC) = (B+C)/2   = (13/2,  5)
Mid(CA) = (C+A)/2   = ( 3/2,  4)

CM      = (A+B+C)/3 = (13/3, 10/3) = (4.33, 3.33)
```

and we don't really need to compute the midpoints to get the centroid!

TRIANGLES: Random Samples

The key to sampling is a parameterization of the points.

Triangle points are linear combinations of vertices A, B and C.

We require the coefficients α, β, γ be nonnegative, and $\alpha + \beta + \gamma = 1$.

To pick a random point p in the triangle ABC, set:

$$
\begin{aligned}
r1 &= \text{rand}(); \\
r2 &= \text{rand}(); \\
\alpha &= 1 - \sqrt{r1} \\
\beta &= \sqrt{r1} * r2 \\
\gamma &= \sqrt{r1} * (1 - r2) \\
p &= \alpha A + \beta B + \gamma C
\end{aligned}
$$

TRIANGLES: Distance

For our last "trick" with triangles, let's ask the simple question, how far is a point p from a triangle?

To answer this question, we begin by computing the quantities **sAB**, **tAB**, **sBC**, **tBC**, **xCA** and **tCA**, the s and t parameters for the point relative to the lines AB, BC, and CA.

If all three t parameters are nonnegative, the point is in or on the triangle, and so the distance is zero.

If just one value of t is negative, then the nearest point is on the line, but the triangle includes just a segment of that line. If the corresponding value of s is between 0 and 1, then the nearest point is part of the line segment, and the distance is $|t|$. But if $s < 0$, the nearest point is the first vertex, and if $1 < s$, the nearest point is the second.

TRIANGLES: Distance

The last possibility is that two values of t are negative.

This means that the nearest point is on one of the two sides, or their common vertex.

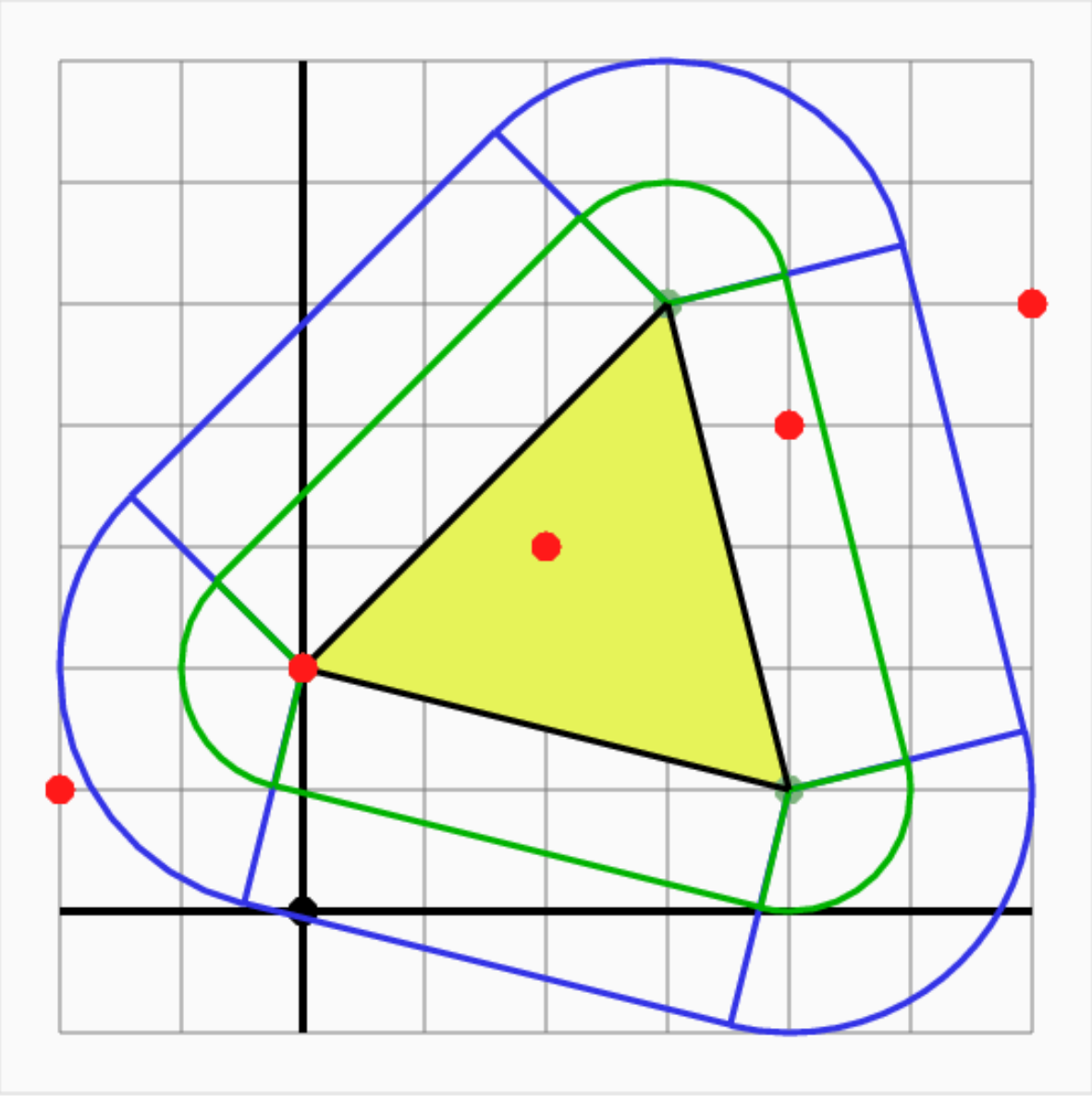
We work by checking the two s values.

If both s values are "out of range" (that is, outside of [0,1]), then the vertex is the closest point.

If just one s value is in range, then the corresponding side is the closest, so the distance is the corresponding value of $|t|$.

And it cannot be the case that both s values are in range.

TRIANGLES: A Picture of the Distance Calculation



TRIANGLES: Summary of Algorithms

We have seen many triangle algorithms:

- **triangle_angles()**;
- **triangle_area()**;
- **triangle_centroid()**;
- **triangle_contains_point()**;
- **triangle_distance_to_point()**;
- **triangle_orientation()**;
- **triangle_perimeter()**, *(You can figure this one out!)*;
- **triangle_sample()**;
- **triangle_side_lengths()**;

Many of these are building blocks for more complicated problems.

Geometry Algorithms

- Overview
- The Points on a Line
- Points NOT on a Line
- Estimating Integrals over an Interval
- Triangles and their Properties
- **Triangulating a Polygon**
- The Convex Hull
- Triangulating a Point Set by Delaunay
- Estimating Integrals over a Triangle
- Conclusion

POLYGONS: Let's Not Start Over!

Triangles are the simplest of the **polygons**, which include squares, pentagons, hexagons, and so on.

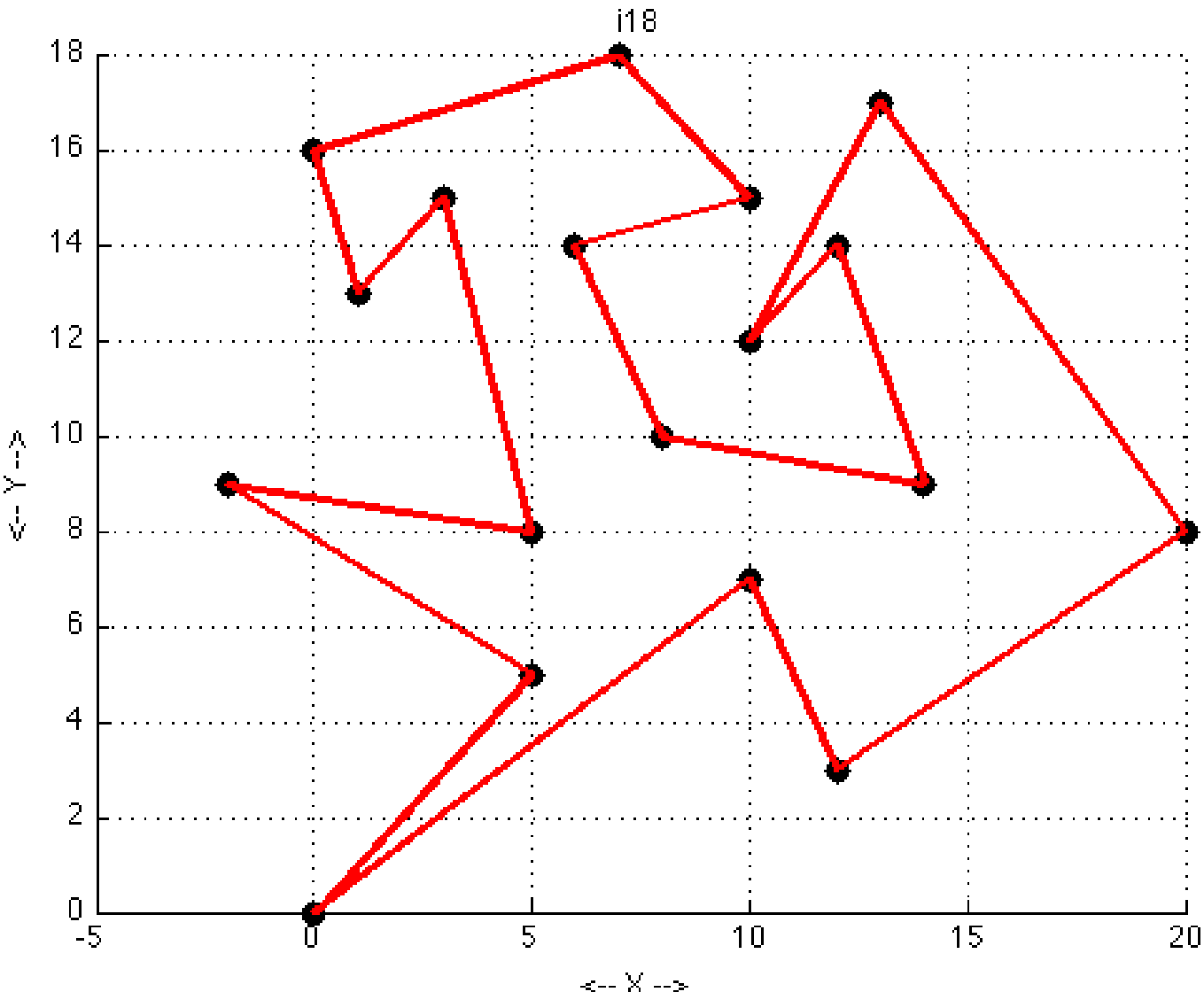
Unlike triangles, however polygons with $3 < \mathbf{N}$ vertices:

- are "bendable"; knowing the sides doesn't tell us the shape.
- might not be convex; there may be "bites" in the shape.
- can cross edges, if we allow "misbehaving" (we won't!);

However, every well-behaved polygon of **N** sides can be triangulated, that is, it can be decomposed into **N**-2 triangles, simply by drawing **N**-3 non-intersecting "diagonals", that is, by connecting pairs of vertices of the polygon.

POLYGONS: A Polygon with 18 Vertices

There are many ways to triangulate this polygon. They all end up with 16 triangles!



POLYGONS: Definition of CONVEX

I used the word convex a moment ago, and it will come up again from time to time.

In the plane, a convex polygon is one that has no "dents".

The strict definition: The object C is convex if and only if, whenever two points $p1$ and $p2$ are elements of the object, so is every point p that lies on the line segment between $p1$ and $p2$:

$$p = t * p1 + (1 - t) * p2$$
$$0 \leq t \leq 1.$$

Circles, squares, regular pentagons are examples of convex shapes.

A star is not convex, nor are the letters of the alphabet, with the possible exception of lowercase **l** and uppercase **I** when drawn in a sans serif font!

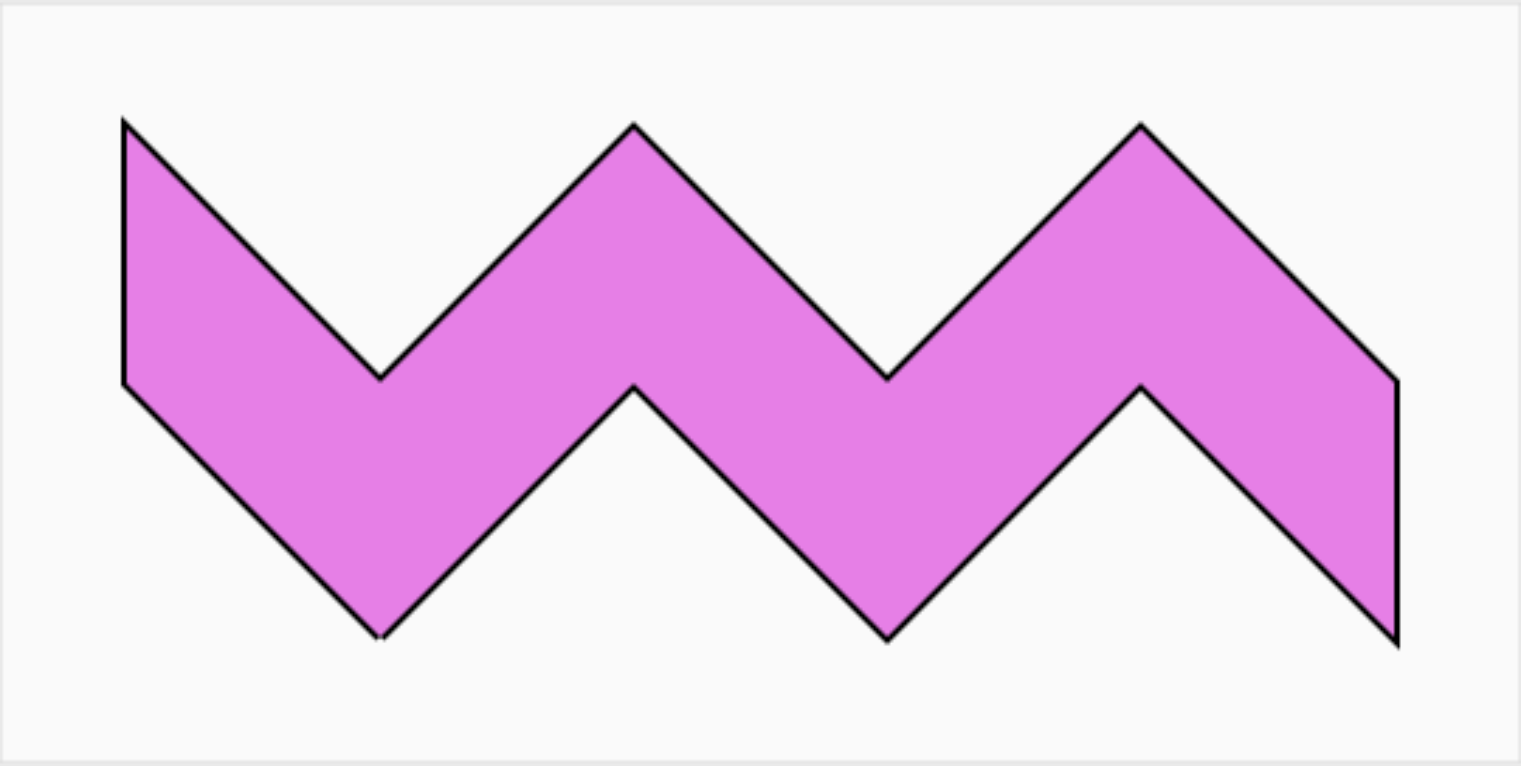
POLYGONS: Let's Not Start Over!

Rather than try to come up with new formulas for the analysis of polygons, it makes sense to triangulate a polygon, apply our formulas to the triangles, and then figure out how to put together those results to say something about the original polygon.

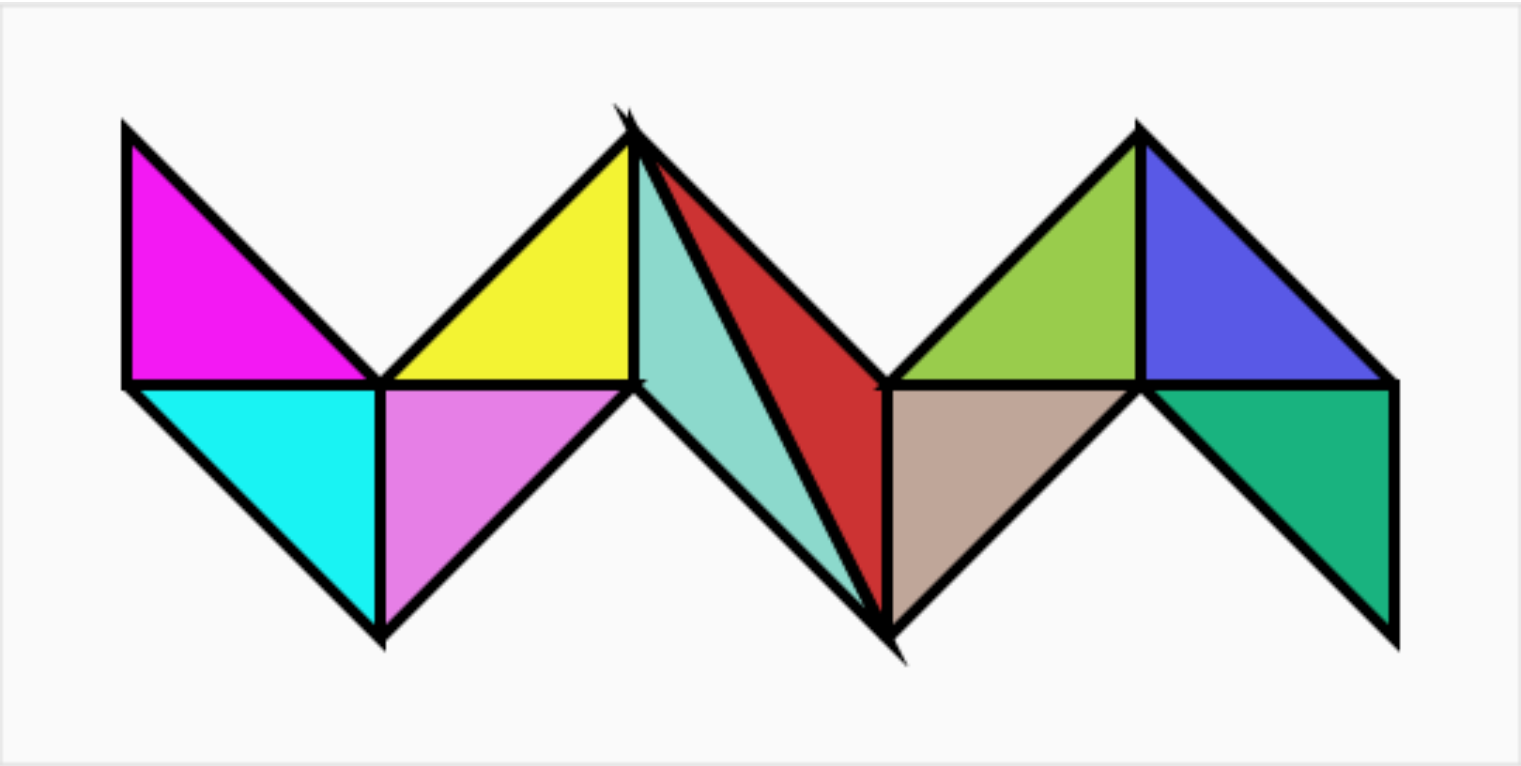
If we can determine a triangulation of a polygon, then

- the area is the sum of the triangle areas;
- a point is in the polygon if it is in a triangle;
- the distance to the polygon is the minimum of the distances to any triangle;
- the centroid of the polygon is computable;

POLYGONS: A "Snake" Polygon



POLYGONS: A Triangulated "Snake" Polygon



POLYGONS: Ideas for Triangulation

Once again, it's easy to see how to do something, but really hard to create an algorithm that works.

One approach starts with the idea of *"Decrease and Conquer"*; that is, we replace the original problem with a partial answer and a smaller problem.

Imagine the polygon was a kind of cake (with lots of corners!). Shouldn't it always be possible to slice one triangular piece off? Doesn't that reduce the number of vertices by one?

QUIZ: May there be some vertices we can't immediately eliminate? Must there always be at least one that we can?

If the polygon is convex, can we cut off any slice (vertex) we like?

POLYGONS: Polygons Have Ears

An ear of a polygon is a triangle that can be formed by three consecutive vertices of the polygon in such a way that two edges of the triangle are edges of the polygon, and the third edge is completely contained inside the polygon.

An ear can be sliced off a polygon, reducing the vertices by 1.

We have the following very useful theorem:

If **P** *is a simple polygon (no internal holes and no edge crossings) with at least 4 vertices, then* **P** *is guaranteed to have at least two distinct ears.* [Meisters, 1975]

Moreover if **P** is convex, every 3 consecutive vertices form an ear.

Can you spot the ears on our "snake" polygon?

POLYGONS: Ear Slicing

The ear theorem means that *every polygon can be triangulated*.

We start with a polygon of **N** vertices, and 0 diagonals.
We locate an ear, defined by the consecutive vertices **B**, **C**, **D**.

We add the diagonal from **B** to **D** to our list.

We also remove **C** from the polygon, and decrease **N** by 1.

Now we have a polygon of **N**-1 vertices, and 1 diagonal.
If **N** is still greater than 3, we repeat the process.

After slicing off **N-3** ears, we have 3 vertices left, forming a single triangle, which is our last ear.

Thus we end up with **N-2** triangles by creating **N-3** internal diagonal lines.

The list of diagonals is our triangulation of the original polygon.

POLYGONS: Ear Detection

We need to be clear about the "find an ear" step!

Suppose we have vertices **A**, **B**, **C**, **D** and **E**, which are consecutive in the counterclockwise ordering. Then **B**, **C** and **D** form an ear if:

- triangle **BCD** has positive area (= counterclockwise);
- the (open) line segment **BD** doesn't intersect any edge;
- the line segment **BD** is "between" **BA** and **BC**;
- the line segment **DB** is "between" **DC** and **DE**.

POLYGONS: Representing a Polygon

We represent a polygon as a linked list. We can delete any vertex so that the remaining data represents the simplified polygon.

We must set up an array of things of **vertex** type.

Each **vertex** has fields called **index**, **next**, and **prev**.

We might store the polygon $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 1$ by:

```
mypolygon
  prev index  next  ear   x     y
   4     1      2    1   0.0   0.0
   1     2      3    1   1.0   0.0
   2     3      4    1   1.0   1.0
   3     4      1    1   0.0   1.0
```

Here the **ear** field is 1 if the vertex defines an ear.

POLYGONS: Removing an Ear

If we decide to remove the ear represented by vertices 2, 3 and 4, then we must remove vertex 3 from the polygon so we have $1 \rightarrow 2 \rightarrow 4 \rightarrow 1$ remaining.

Essentially, all we need to do is change the pointers:

mypolygon

prev	index	next	ear	x	y
4	1	2	1	0.0	0.0
1	2	4	1	1.0	0.0
0	0	0	0	1.0	1.0
2	4	1	1	0.0	1.0

And now suddenly we have a triangle rather than a square!

POLYGONS: The Ear Removal Tasks

If we have decided that **B**, **C** and **D** form an ear then we need to

- add diagonal **B**, **D** to our list;
- remove vertex **C** from the polygon (reset **C.index** to 0);
- reset **B.next** to **D** and **D.prev** to **B**;
- reset **B.ear** if **A**, **B**, **D** has become an ear;
- reset **D.ear** if **B**, **D**, **C** has become an ear.

POLYGONS: The Slicing Code

```
i2 = first;
while ( diagonal_num < n - 3 )
  if ( vertex(i2).ear )
    i3 = vertex(i2).next;  i4 = vertex(i3).next;
    i1 = vertex(i2).prev;  i0 = vertex(i1).prev;
    vertex(i1).next = i3;
    vertex(i2).index = 0;
    vertex(i3).prev = i1;
    vertex(i1).ear = diagonal ( i0, i3, vertex );
    vertex(i3).ear = diagonal ( i1, i4, vertex );
    diagonal_num = diagonal_num + 1;
    diagonals(diagonal_num,1) = i1;
    diagonals(diagonal_num,2) = i3;
  end
  i2 = vertex(i2).next;
end
```

POLYGONS: An Example Implementation

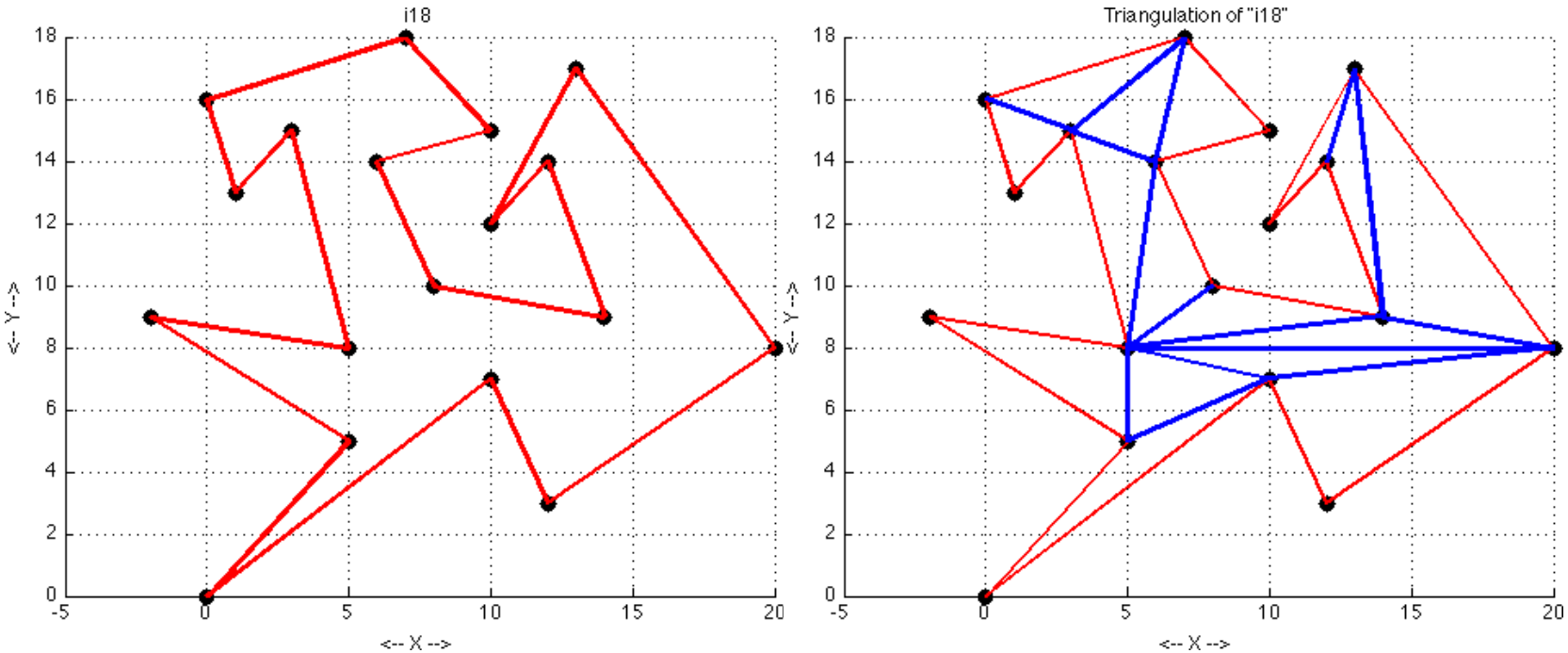
An example of a MATLAB code for carrying out the ear-slicing algorithm for polygon triangulation is available at:

http://people.sc.fsu.edu/~jburkardt/m_src/triangulate/triangulate.html

and a C version is available at

http://people.sc.fsu.edu/~jburkardt/c_src/triangulate/triangulate.html

POLYGONS: Ear Slicing Example



If there's time, let's watch this happen as an animation!

POLYGONS: Computing Polygon Properties

Once we have the triangulation of a polygon, we can apply some of the triangle algorithms to problems about polygons.

To do this, we have to save the triangles instead of just the diagonals. This requires two changes:

- Each time we find a diagonal I1, I3 by slicing off node I2, we must add triangle I1, I2, I3 to the triangle list.
- After the last diagonal is computed, we have a triangle remaining, which we must append to the triangulation.

POLYGONS: Triangles, with diagonals in black

2	18	1
4	2	3
7	5	6
8	5	7
12	10	11
15	13	14
18	16	17
2	16	18
4	16	2
8	4	5
15	12	13
8	16	4
9	16	8
10	16	9
15	10	12
16	10	15

POLYGONS: Computing Polygon Properties

Some polygon algorithms don't require a triangulation.

The *side lengths* can be computed simply by taking the norm of the vector from one vertex to the next.

The *perimeter* is just the sum of the side lengths.

The *angles* can be computed in a straightforward manner. If we have consecutive vertices A, B, and C, then recall the formula

$$\begin{aligned} v1 \times v2 &= ||v1|| \cdot ||v2|| \cdot \sin(\theta) \\ v1 \cdot v2 &= ||v1|| \cdot ||v2|| \cdot \cos(\theta) \\ \theta &= \text{atan2}(v1 \times v2, v1 \cdot v2) \end{aligned}$$

Letting $v1 = (C - B)$ and $v2 = (A - B)$, the formula gives the angle at vertex B.

POLYGONS: Computing Polygon Properties

The triangles making up a polygon can be used to compute some geometric quantities:

1. the **area** of the polygon is the sum of the areas of the triangles;
2. the **centroid** of the polygon can be computed as the sum of the triangle centroids, each multiplied by its area, and then divided by the total area;
3. a polygon **contains a point** if and only if one of the triangles contains the point;
4. the **distance from a point** to a polygon is the minimum of the distances to the triangles;

POLYGONS: Random sampling

Can our triangle algorithms do random sampling of a polygon?

If we have triangulated the polygon, we can sample the polygon by sampling one of the triangles.

If one triangle has twice the area of another, then we should probably sample it twice as often.

So the correct procedure is as follows:

1. let A(I) be the area of the I-th triangle, and ATOTAL the total area;
2. choose a random value r1, and consider $B = r1 * ATOTAL$;
3. Using no triangles, we have 0 area, and using them all we have ATOTAL; therefore, there is some triangle J so that the sum of areas 1 to J just reaches or exceeds B.
4. pick a random point from that triangle J;

POLYGONS: Summary of Algorithms

Our collection of polygon algorithms now includes:

- **polygon_angles()**;
- **polygon_area()**;
- **polygon_centroid()**;
- **polygon_contains_point()**;
- **polygon_distance_to_point()**;
- **polygon_perimeter()**;
- **polygon_sample()**;
- **polygon_side_lengths()**;

Geometry Algorithms

- Overview
- The Points on a Line
- Points NOT on a Line
- Estimating Integrals over an Interval
- Triangles and their Properties
- Triangulating a Polygon
- **The Convex Hull**
- Triangulating a Point Set by Delaunay
- Estimating Integrals over a Triangle
- Conclusion

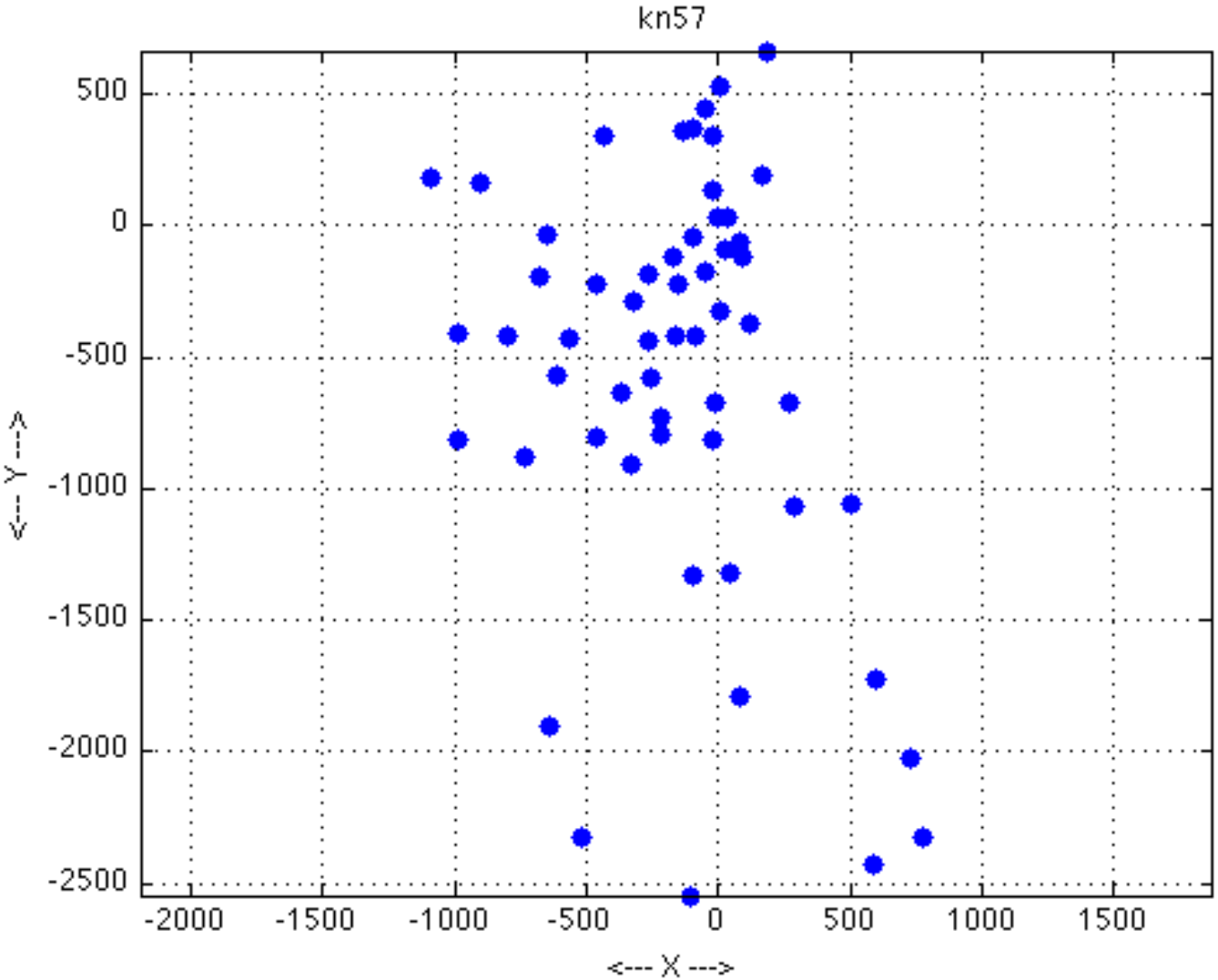
HULL: The "Shape" of a Set of Points

In many scientific applications, data is given at a scattered set of points. To organize this data, one need is to try to describe the geometric location of the points. Unless we are very lucky, the points will not lie on the vertices of a square, or in a straight line, or any other regular shape.

The first thing we might think to report is the range of the data.

Assuming the geometric data is two dimensional, then we could report the minimum and maximum x and y values. This is like putting a rectangle around the data. One thing that does is give us a feel for the "area" covered by the data. However, since our rectangle must line up with the x and y coordinate axes, it's almost surely not the smallest rectangle that could box in the data!

HULL: 57 Points



HULL

So we could imagine that we're trying to **fence in** the data using a rectangle that we can turn at an angle. If we're paying for the fence, we'd like to use the least length possible.

What if we allowed ourselves to use pentagons instead? The minimum fence length (the perimeter) over all possible pentagons must be as low, or lower, than what we can get with rectangles.

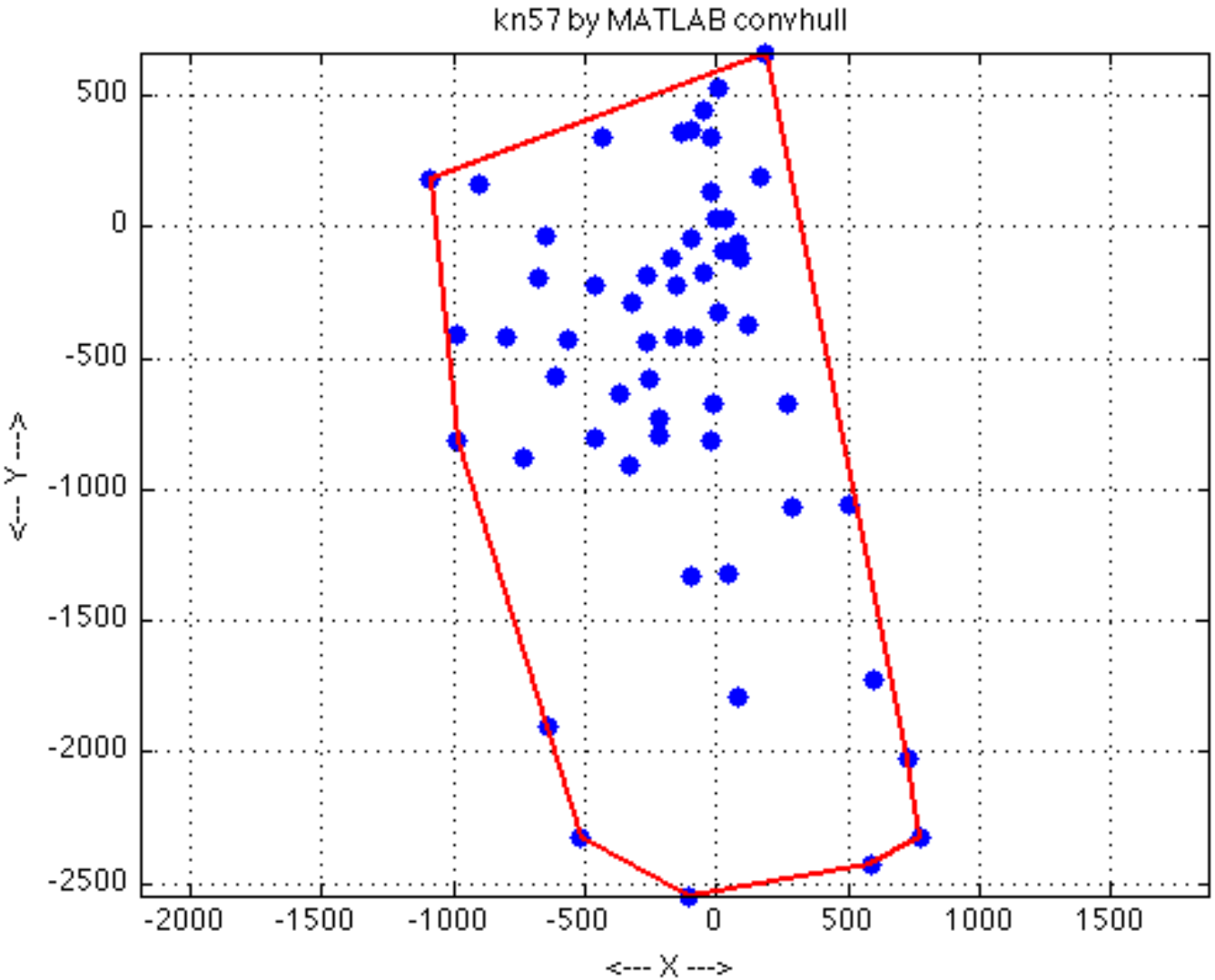
To drive the perimeter length as low as possible, we should consider every possible polygon that encloses the data.

It's clear that the "fence posts" will always occur at data points.

It should also be clear that the smallest perimeter requirement implies that the hull will be convex.

The convex hull problem: *find the convex polygon of smallest perimeter which contains a given set of points.*

HULL: 57 Points Fenced In



HULL: Things to Note

The convex hull of this data was computed by MATLAB's **convhull** command.

The convex hull of a *finite* set of points is a polygon: every vertex is a data point, and every edge is a straight line segment.

In fact, if you had to build the cheapest fence that contained the data, the bends in the fence would come at data points, just as we see in the picture.

Looking at the picture, you can also imagine it to be a kind of "wrapping" problem, that is, we could imagine a group of trees that we are going to surround with a rope fence. When we pull the rope as tight as possible, we get the convex hull.

In fact, the idea of wrapping the data will lead us to an algorithm.

HULL: Convexity

Although polygons are allowed to have dents or wiggles, the convex hull of our data has no "dents" (and no internal holes either).

Quiz: Can you explain why this should be true?

The hull is called convex because if **p** and **q** are any points in the shape, so is every point on the line between **p** and **q**.

Although a convex shape can be described by its boundary, it actually is not just the boundary, but all points contained inside.

Thus, the letter "**I** is a convex shape, the letter "**O**" is not, but it is the boundary of a convex shape, while the letter "**A**" is neither a convex shape nor the boundary of one.

HULL: Convexity

A second way to define the convex hull of a set of points is that it is the smallest convex shape that contains the points.

If a shape is convex, it is its own convex hull (this **must** be true!)

But wait a minute...a circle is convex, so it's a convex hull. But we said a minute ago that the convex hull is a polygon, made with straight line segments. *Is there a contradiction here?*

The definitions of a convex shape and a convex hull can be extended to higher dimensions.

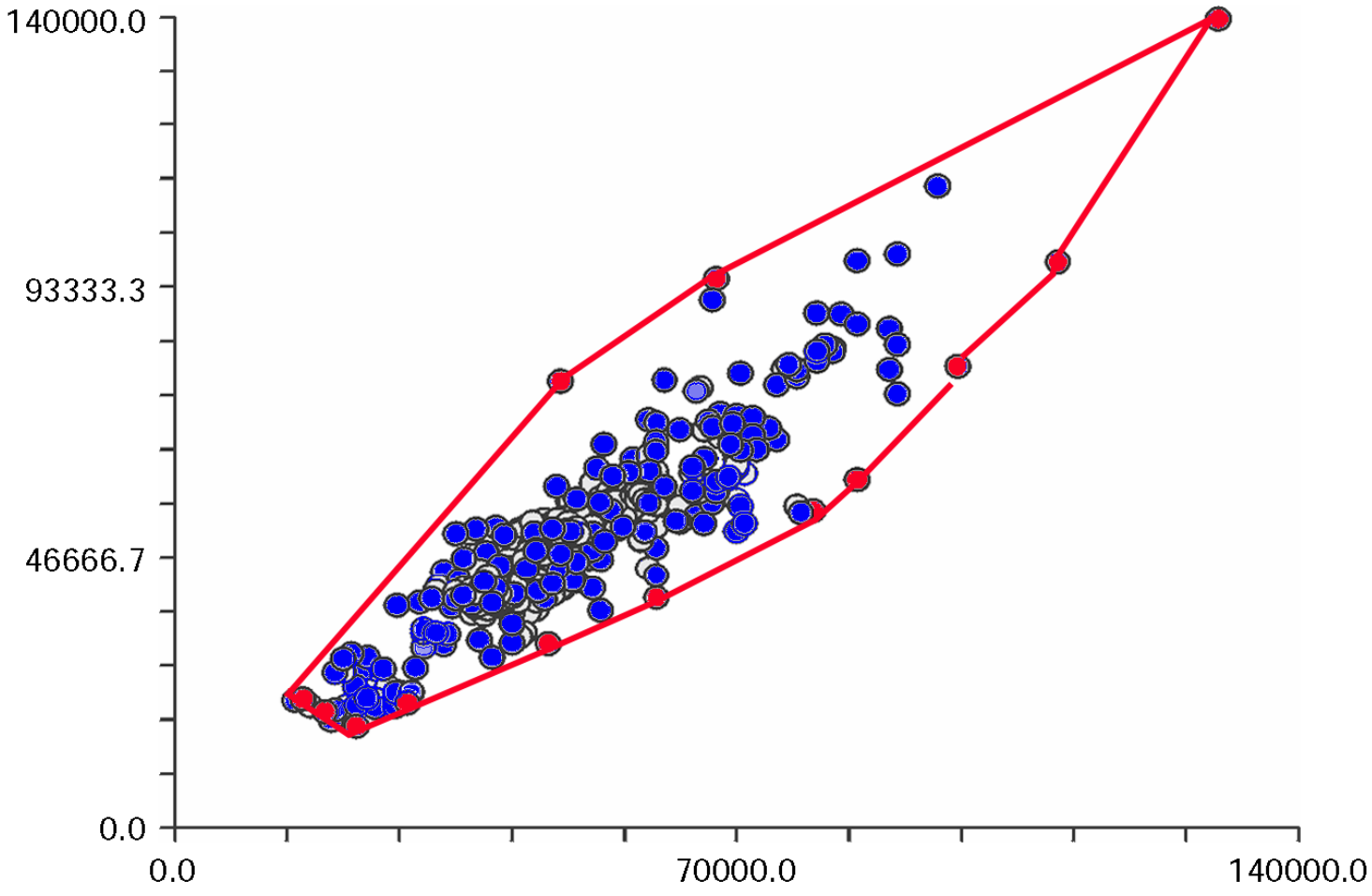
HULL: Not just geometry!

Like many things in mathematics, the convex hull is not simply a question about geometry!

We often have cases where we have a great deal of data available, which we think of as samples from some larger set of possibilities.

One way to sample that larger set begins by constructing the convex hull of the data. The convex hull is a polygon, so we can triangulate it, and hence take sample values. By sampling within the convex hull, our "simulated" data stays within the range of the original data, but can return completely new values within that range.

HULL: A data hull for sampling



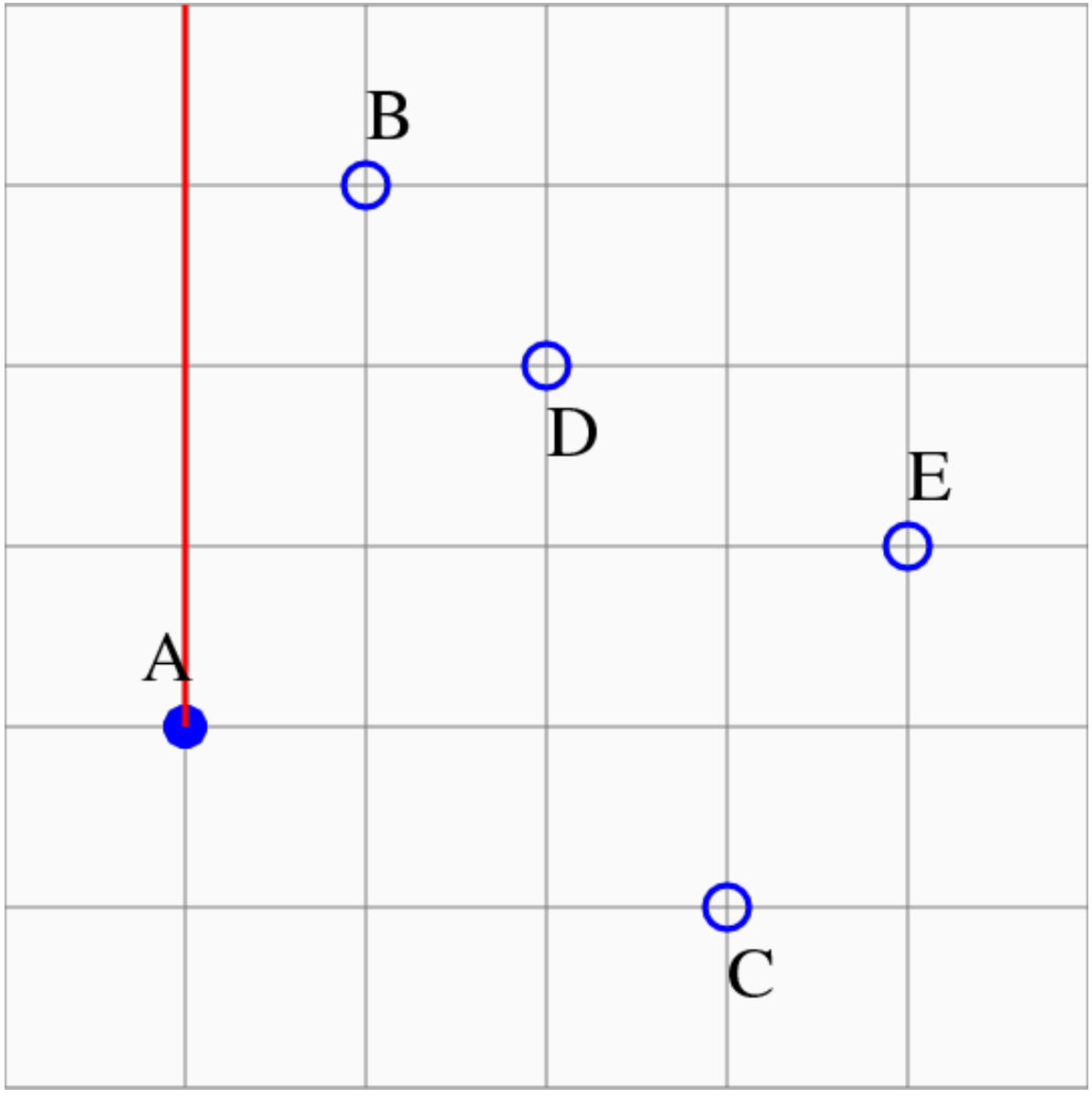
HULL: The Wrapping Algorithm

We can always find one point on the convex hull: simply find the point with the minimum x component. If there are several points with the same minimal x component, choose the one with smallest y component. That gets us started, with a data point we will call vertex **H1**.

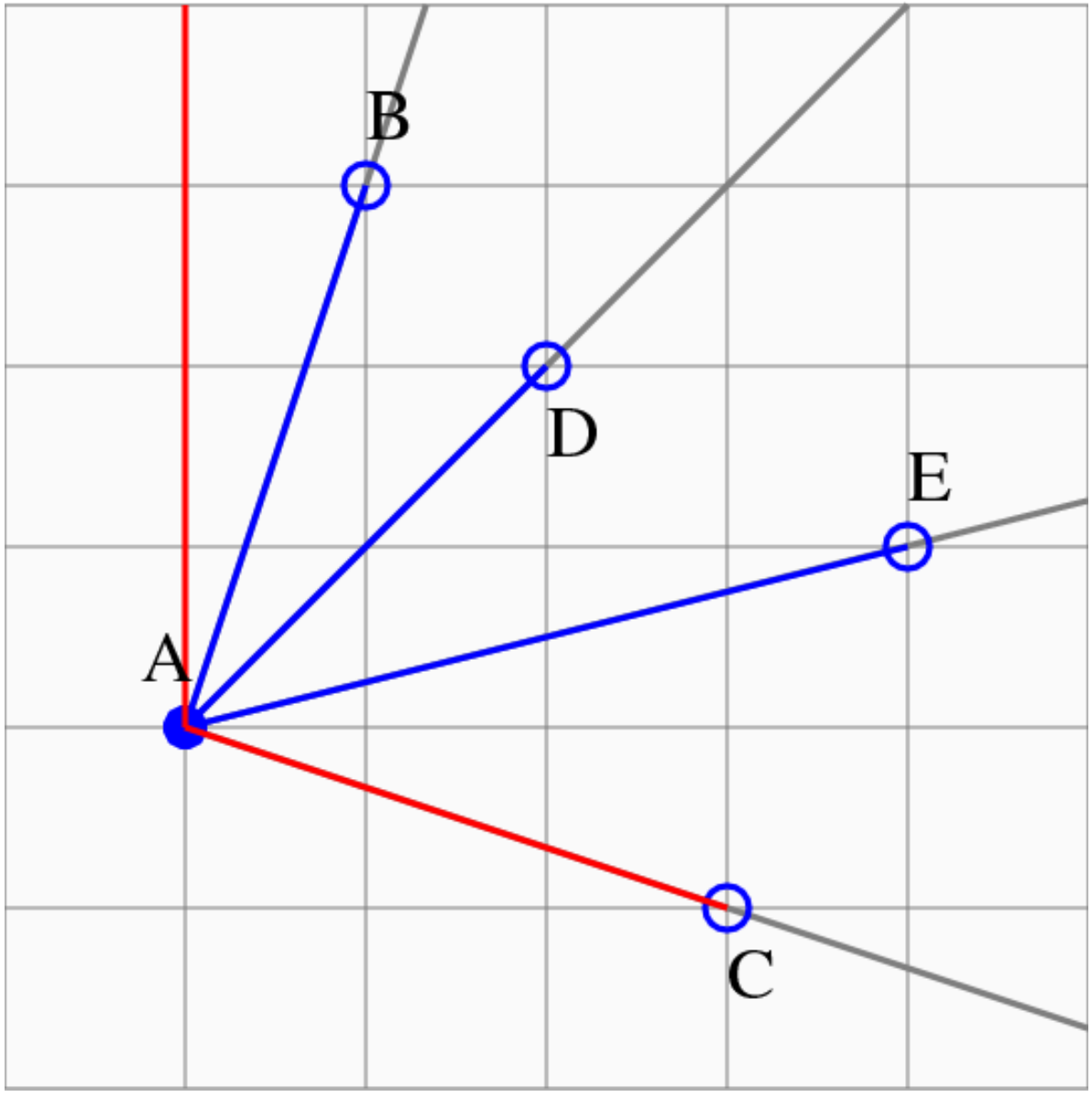
Now we have to determine the next vertex of the polygon. We have **N**-1 datapoints to choose from. Let's assume we are trying to build the polygon by following the vertices in the counterclockwise direction. So we have **N**-1 possible edges, from **H1** to each of the unused datapoints.

The correct edge **(H1,H2)** *is the unique line through* **H1** *to some vertex* **V** *with the property that all data points lie to the left of it.*

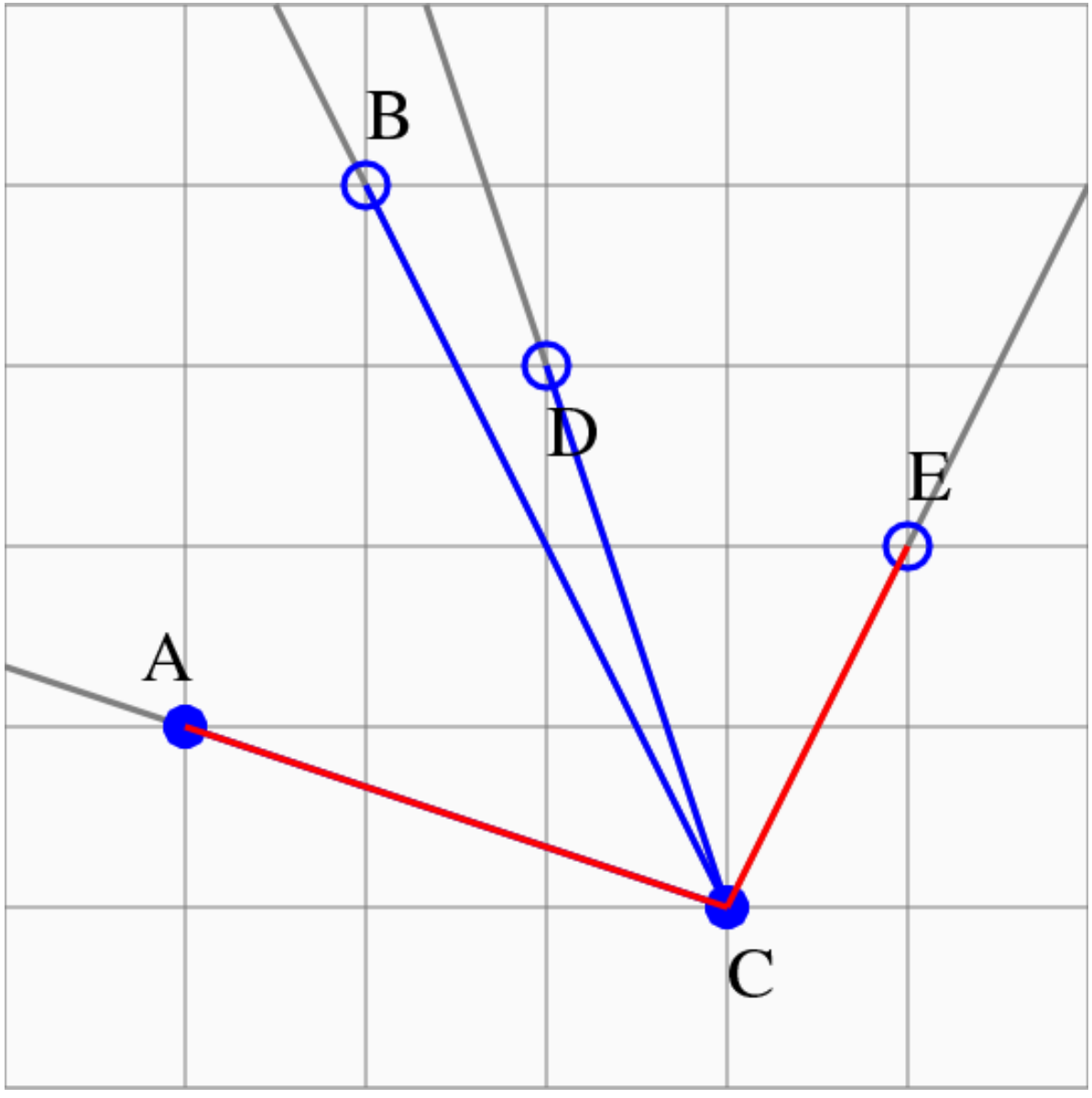
HULL: Find Edge 0



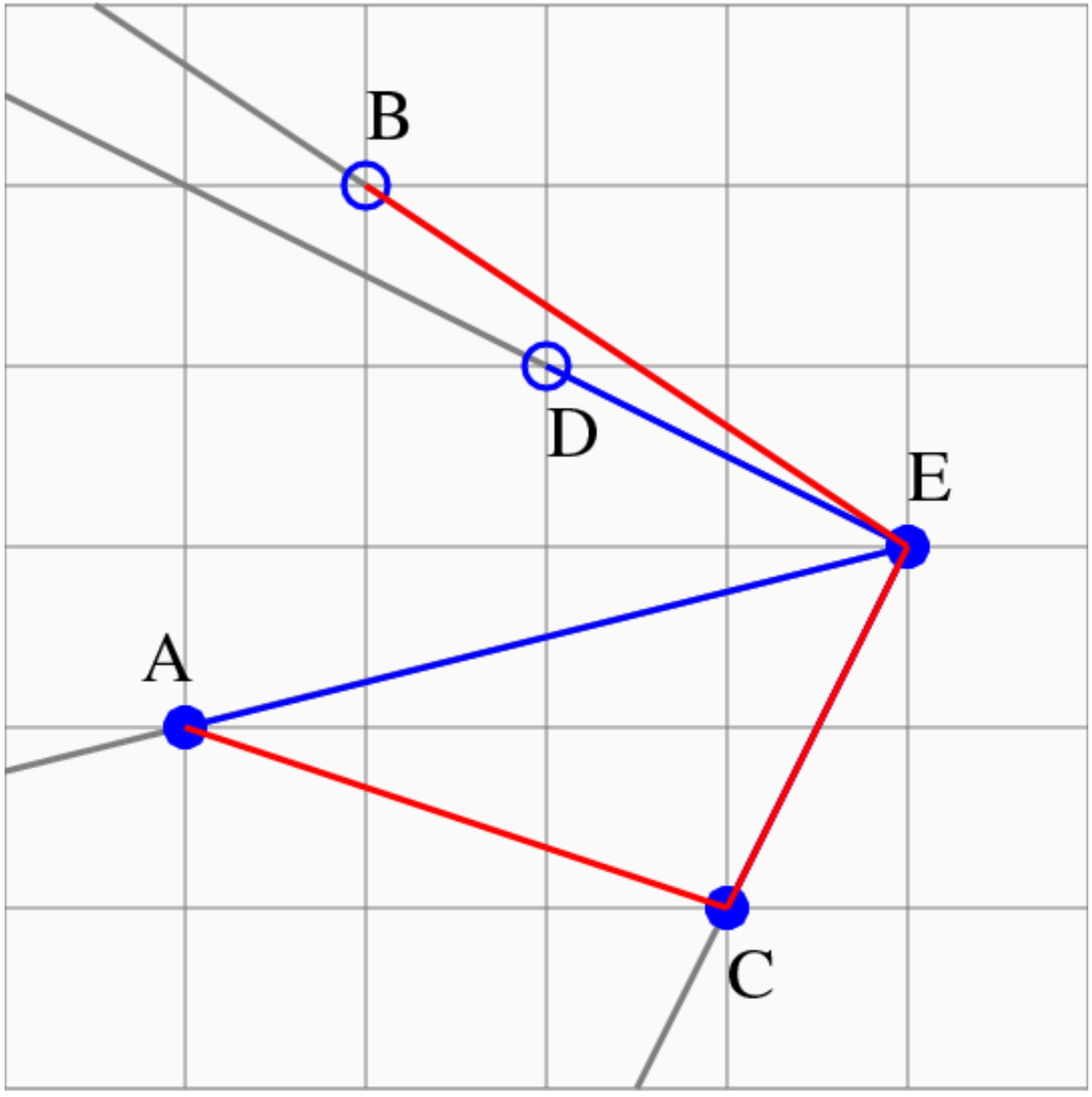
HULL: Find Edge 1



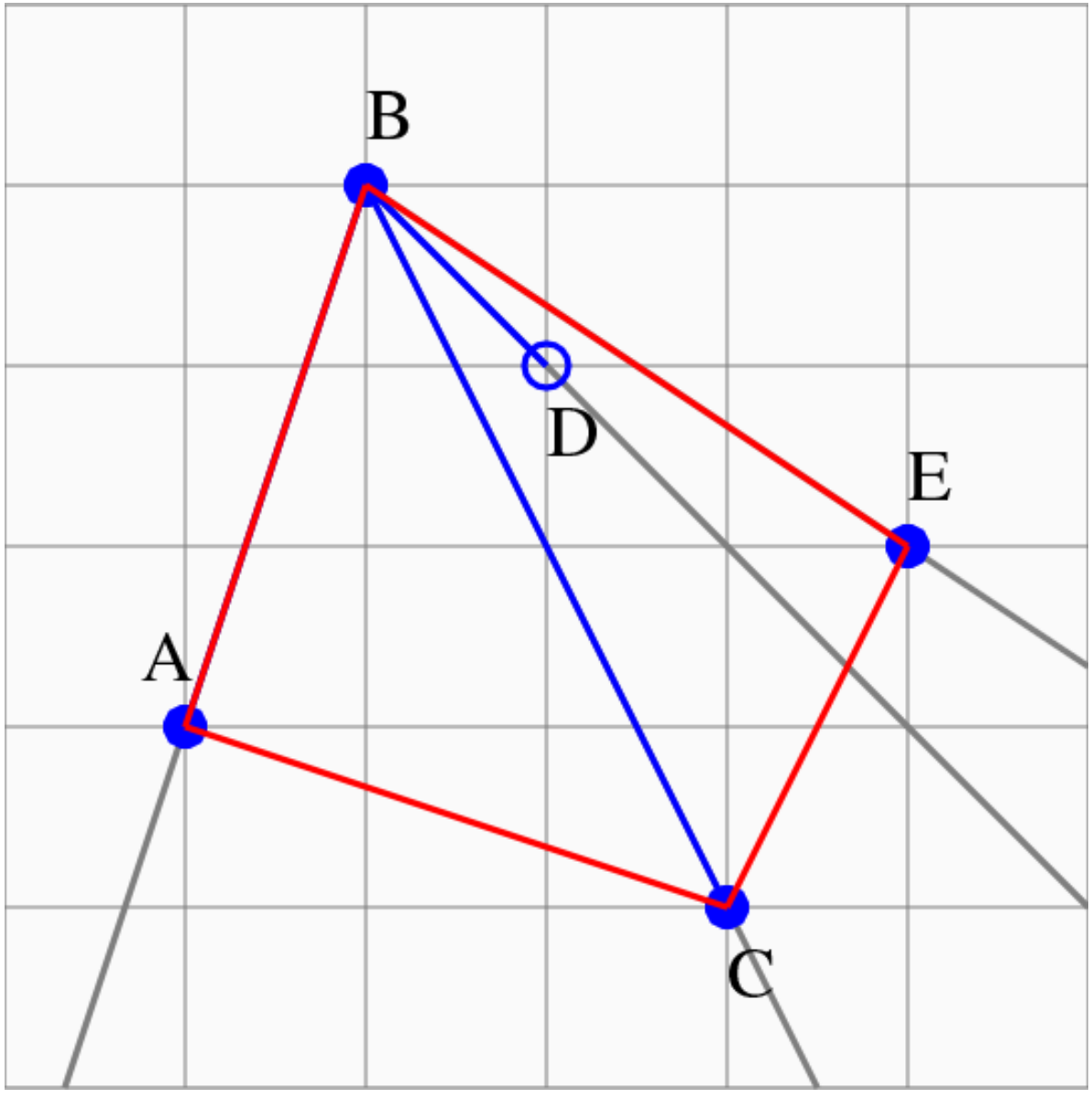
HULL: Find Edge 2



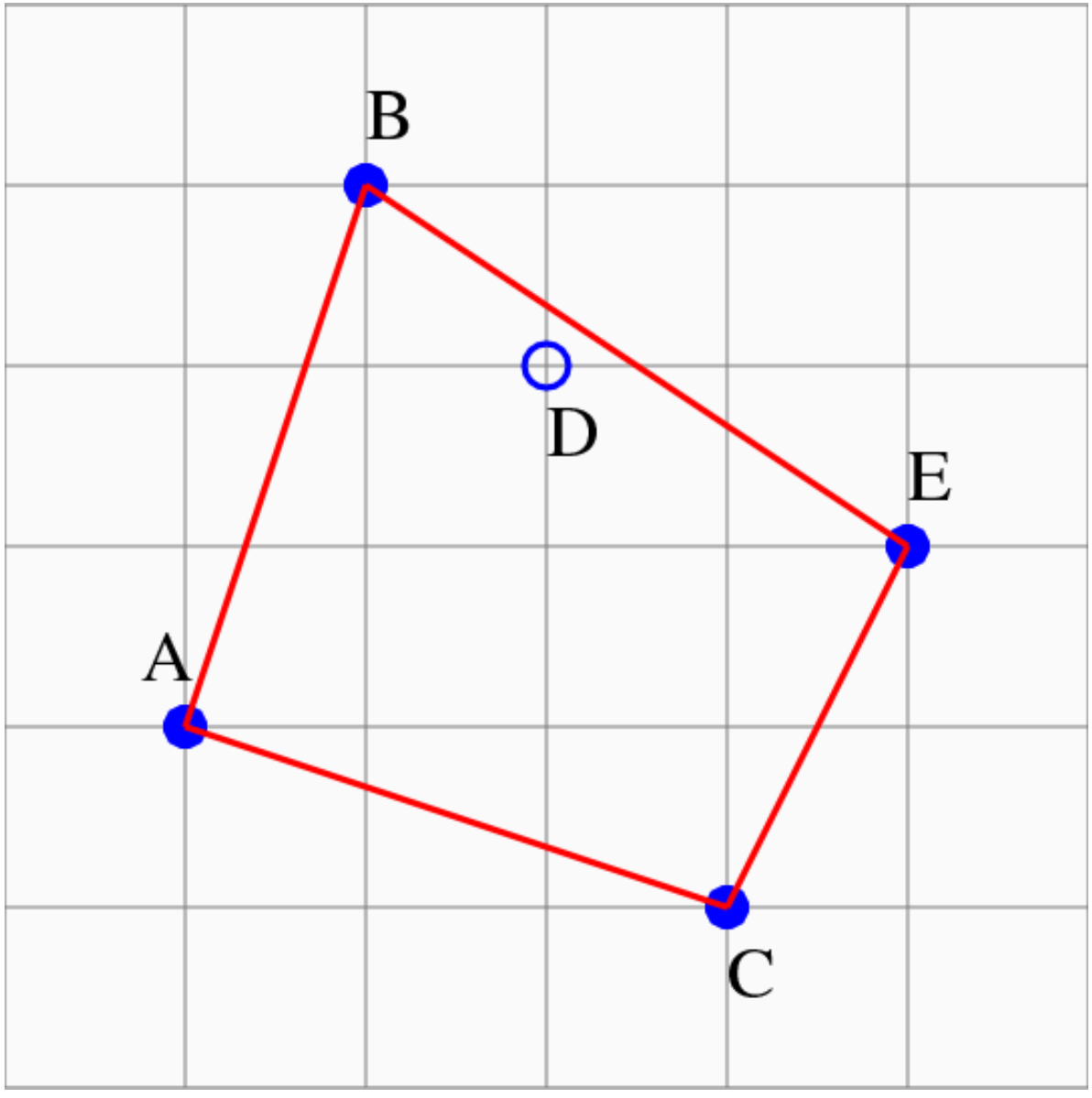
HULL: Find Edge 3



HULL: Find Edge 4



HULL: Finished



HULL; Summary

For a convex hull calculation in MATLAB, use a command like

k = convhull **(x, y)**

k indexes the points which form the convex hull. From **k** we have the polygon containing the points, hence the area, the perimeter, the ability to sample, and so on.

To see a plot of the data points and their hull, use:

```
plot ( x, y, '.' );
k = convhull ( x, y )
hold on
plot ( x(k), y(k), '-r' )
hold off
```

The command **convhulln()** is available for higher dimensions.

Geometry Algorithms

- Overview
- The Points on a Line
- Points NOT on a Line
- Estimating Integrals over an Interval
- Triangles and their Properties
- Triangulating a Polygon
- The Convex Hull
- **Triangulating a Point Set by Delaunay**
- Estimating Integrals over a Triangle
- Conclusion

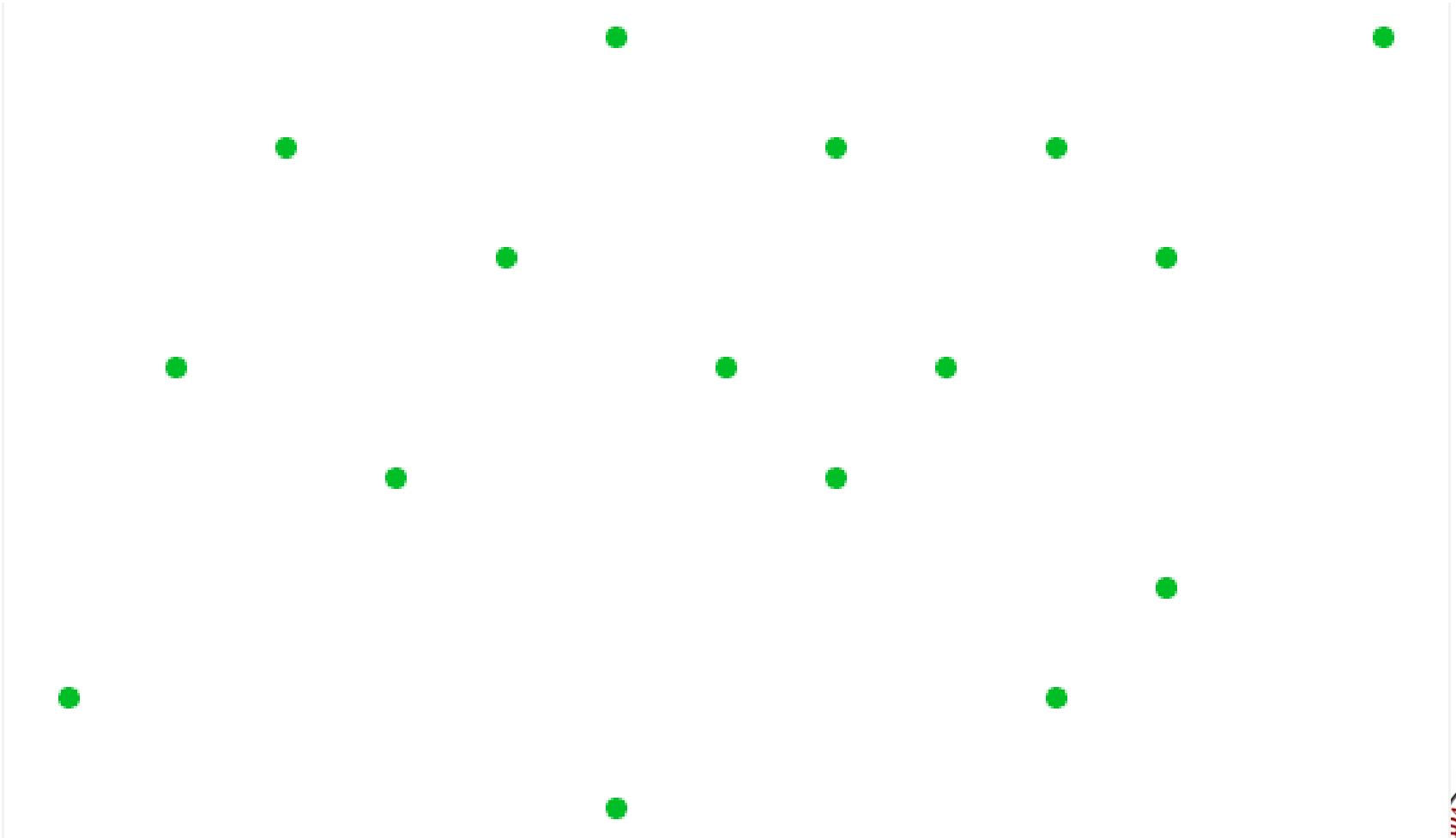
DELAUNAY: The Point Set Problem

Suppose we draw a number of points on the plane. It is certainly possible to draw a line connecting two of the points. Now let's draw another line...except that we are not allowed to cross the first line. That should still be easy. Surely we can draw many lines, but just as surely, there will come a point when we cannot draw any more lines without crossing one we already drew.

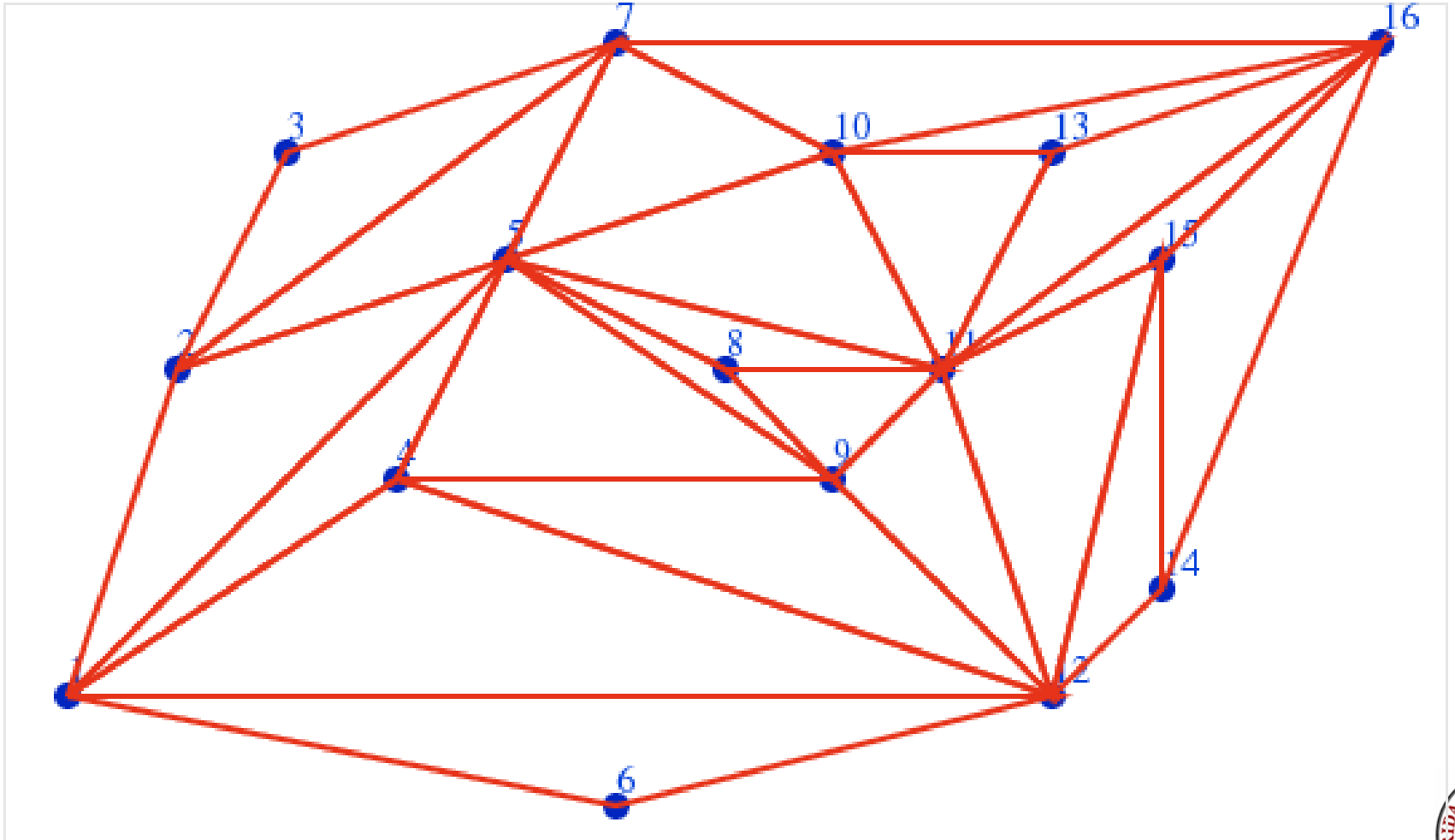
It may surprise you to realize that, for a given set of points, there are many final results possible, but in every case, if we cannot draw any more lines, then all the points on the plane are now vertices of triangles.

We have now triangulated a set of points. This is similar to the problem of triangulating a polygon, but now we do not start with a bounding polygon, and we imagine we might have many points do deal with, so efficiency is important.

DELAUNAY: A Set of 16 Points



DELAUNAY: A Triangulation of 16 Points



DELAUNAY: What is a "good" triangulation?

We drew the lines of our triangulation at random. If we tried a second time, we'd get a different picture. There are actually many ways to triangulate a set of points in the plane. Given that fact, it's likely that some triangulations are "better" than others, but that depends on what we want to do with our triangulations!

If we think about the connecting lines as "roads", we might prefer a triangulation that uses the shortest total length.

If we think about the triangles as representing patches of territory, we might dislike triangles that have a very small angle.

For graphics applications, and for many computational purposes, *the avoidance of small angles* is a very common criterion.

DELAUNAY: What is a "good" triangulation?

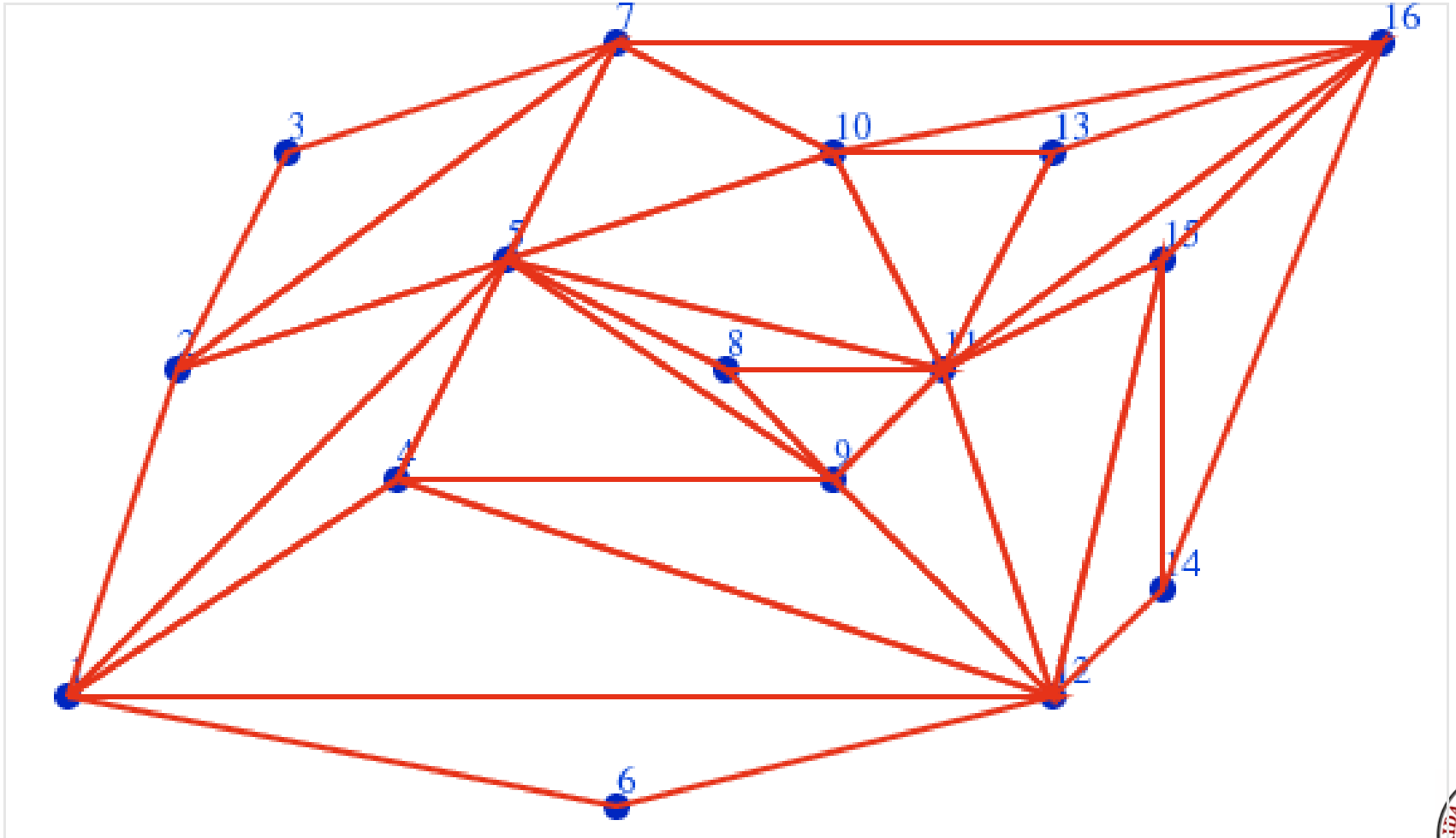
The Delaunay triangulation of a set of points is the (usually unique) triangulation which does the best job of avoiding small angles.

More strictly speaking, consider all possible triangulations of a set of data points. For each triangulation T, let $\theta(T)$ be the smallest angle that occurs in any triangle of that triangulation. Then a triangulation T^* is a Delaunay triangulation if

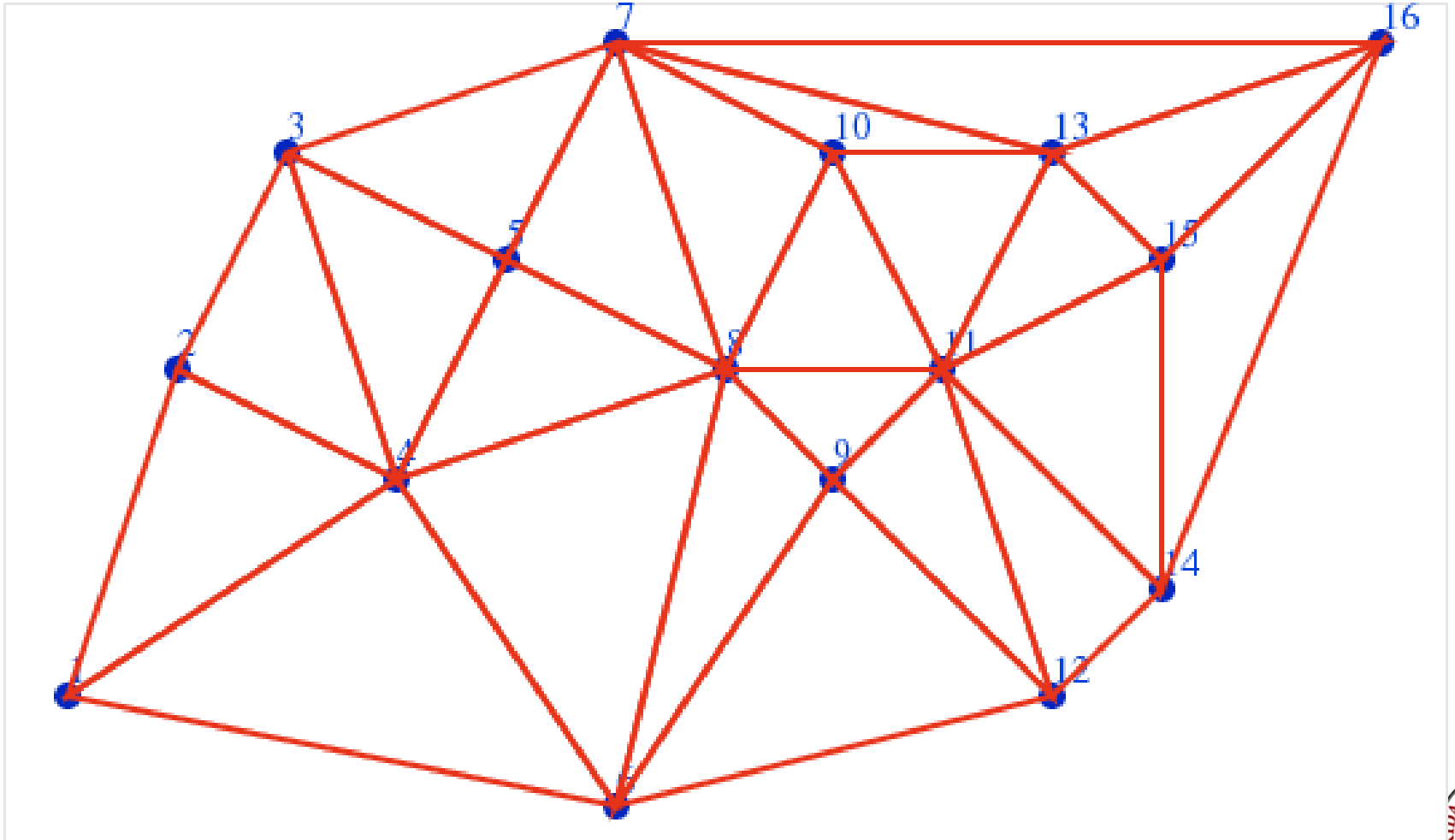
$$\theta(T) \leq \theta(T^*)$$

for all triangulations T.

DELAUNAY: A Triangulation of 16 Points



DELAUNAY: A Delaunay Triangulation of 16 Points



DELAUNAY: MATLAB Calculation

To compute the triangles that form a Delaunay triangulation of a set of data points, use the MATLAB command

tri = **delaunay** (x, y)

To display the triangulation,

tri = **delaunay** (x, y)
triplot (tri, x, y)

DELAUNAY: 3D Surfaces

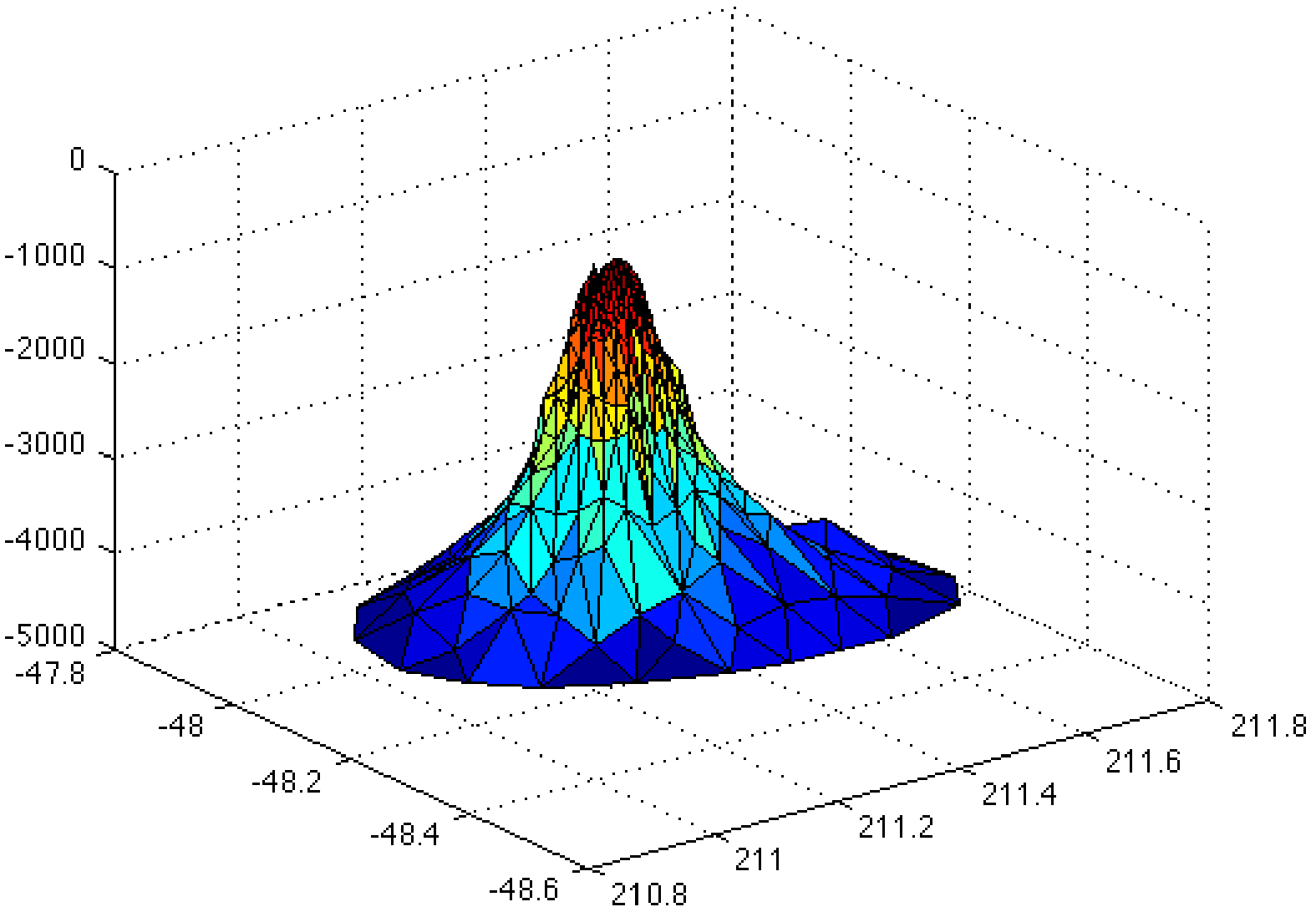
Often, measurements of a 3D surface are available at an irregularly scattered set of points.

If we can organize the (X,Y) data into a triangular mesh, then over each triangle, we can draw a flat surface defined by the three corresponding Z values.

Doing this for each triangle in the mesh, we can create a 3D image of the surface, where before we simply had point data.

load seamount (...a built in XYZ dataset)
tri = **delaunay** (x, y)
trisurf (tri, x, y, z)

DELAUNAY: A 3D Image from XYZ Point Data



Geometry Algorithms

- Overview
- The Points on a Line
- Points NOT on a Line
- Estimating Integrals over an Interval
- Triangles and their Properties
- Triangulating a Polygon
- The Convex Hull
- Triangulating a Point Set by Delaunay
- **Estimating Integrals over a Triangle**
- Conclusion

INTEGRALS: Now the Region is the Problem!

When you learned integration in Calculus, you probably began with the idea that it was simply "the inverse" of differentiation. Why someone would want to differentiate a formula and then integrate it back was not clear.

Perhaps later you were told a much more suggestive interpretation of integration: integration carries out the summation of "very many" "very small" terms.

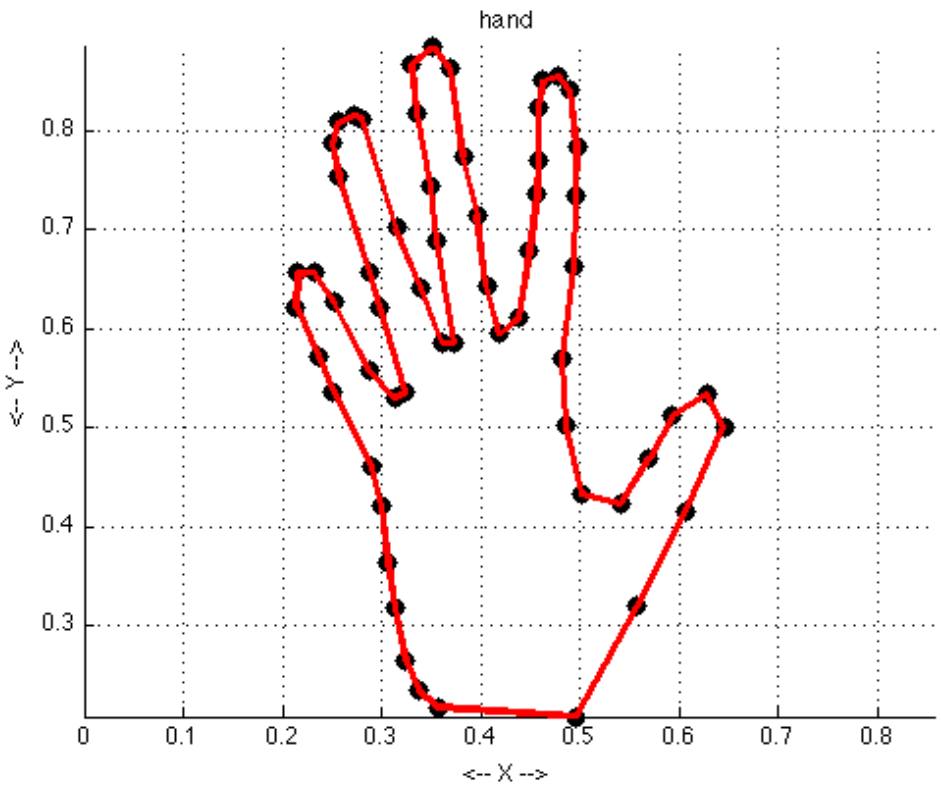
Thus, we approximated the integral of a function (the area under its curve) by a sum over many subintervals, with the true integral being the limit as these subintervals become arbitrarily small.

We saw many unusual functions to integrate, but the integration region itself was usually simple: an interval, or perhaps a box, or very occasionally a surface of rotation...which turned out to be just an interval plus a "twist".

INTEGRALS: Simple Function, Weird Geometry!

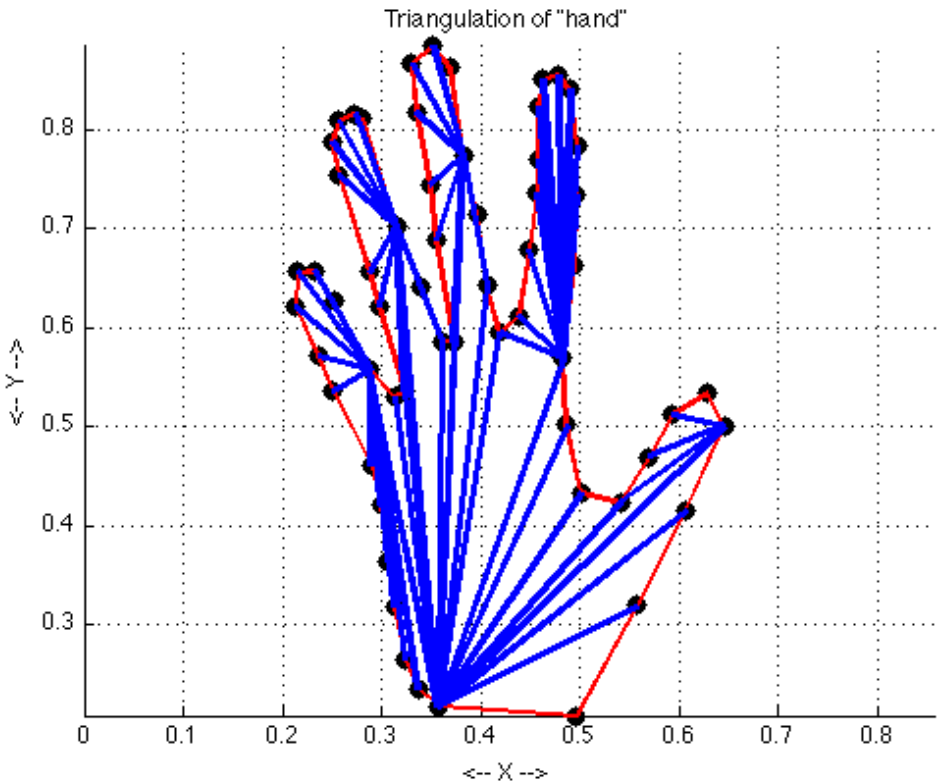
Real life problems require us to use Calculus in new ways.

Computing the area of this hand is the same as integrating $f(x, y) = 1$. The difficulties come not from the function, but the unusual region.



INTEGRALS: Simple Function, Weird Geometry!

Keeping in mind that we're really integrating $f(x, y) = 1$, we figure we can estimate the area by triangulating the region, computing the area of each triangle, and taking the sum.



INTEGRALS: Integrating F(X,Y)=1 is Easy

But now suppose that I wanted to compute the volume of this hand, and that for any point (x, y) I can measure the height or thickness of the hand, which I can regard as the function $f(x, y)$.

$$\text{Volume} = \int_{\text{hand}} f(x, y)dx\,dy$$

An accurate estimate of this integral will approximate the volume of the hand.

I knew how to integrate the function $f(x, y) = 1$, because that was just the area. But now I have an integral of a complicated function $f(x, y)$ over a complicated region.

What do I do?

INTEGRALS: Approximate Integration over Triangles

When we computed the area, we broke the region up into triangles. That idea is still the right way to go, because it reduces the complicated geometry to a sum of simple geometries.

We are left with the problem of approximating the integral of a function over a general triangle.

For special cases where the triangle has two sides aligned with the x and y axes, we can come up with exact integration formulas. But for the hand volume problem, this is not realistic to expect.

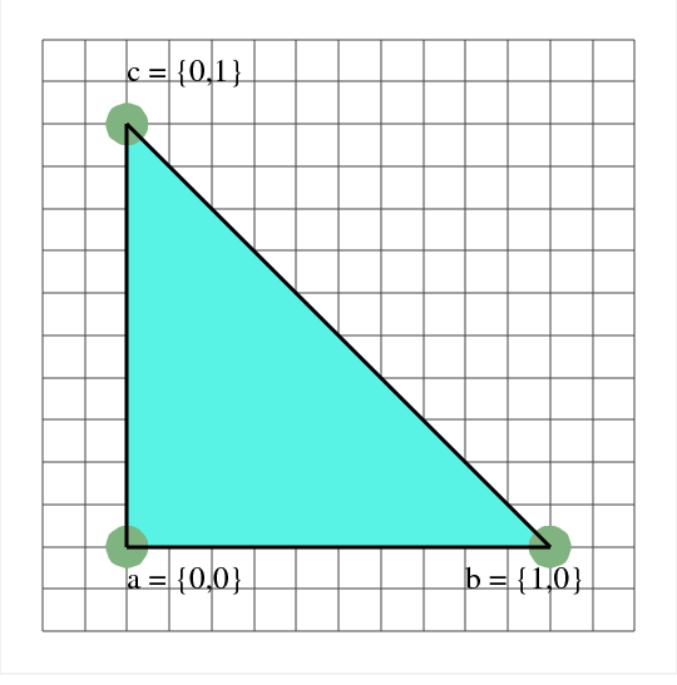
Instead, we will turn to some very useful formulas which allow us to estimate the integral of a function over *any triangle* as a weighted average of its value at a few prescribed points.

This technique is known as quadrature.

INTEGRALS: Quadrature Rules

A quadrature rule is a set of n points (x_i, y_i) and weights w_i which can be used to estimate the integral of a function $f(x, y)$ over the unit triangle T01:

$$\int_{T01} f(x, y)\, dx \approx \text{Area(T01)} \cdot \sum_{i=1}^{n} w_i \cdot f(x_i, y_i)$$



INTEGRALS: A Rule of Precision 1

By averaging the function value at the vertices, we get a rule which approximate the integral, and is actually exactly right if $f(x, y)$ is equal to a constant, or a linear function.

```
Points:
-----------------------
x = [ 1, 0, 0 ];
y = [ 0, 1, 0 ];
```

```
Weights:
-----------------------
w = [ 1/3, 1/3, 1/3 ];
```

INTEGRALS: A Rule of Precision 1

Another nice thing about the *vertex rule* is that it's obvious how to use the same rule on a general triangle.

Since the vertex rule is only "precise" for constants and linears, it is natural to seek rules that are "more precise", that is, which get the exact answer when the function is any polynomial up to some given degree.

Using just 6 function values, we can devise a rule through degree 4, so that it can integrate exactly any polynomial like:

$$\begin{aligned} f(x,y) = & a \\ & + bx + cy \\ & + dx^2 + exy + fy^2 \\ & + gx^3 + hx^2y + ixy^2 + jy^3 \\ & + kx^4 + lxy^3 + mx^2y^2 + nxy^3 + oy^4 \end{aligned}$$

INTEGRALS: A Rule of Precision 4

```
Points:                          Weights:
-----------------------          -----------------------
a = 0.816847572980459;
b = 0.091576213509771;
c = 0.108103018168070;           u = 0.109951743655322;
d = 0.445948490915965;           v = 0.223381589678011;

x = [ a, b, b, c, d, d ];        w = [ u, u, u, v, v, v ];
y = [ b, a, b, d, c, d ];
```

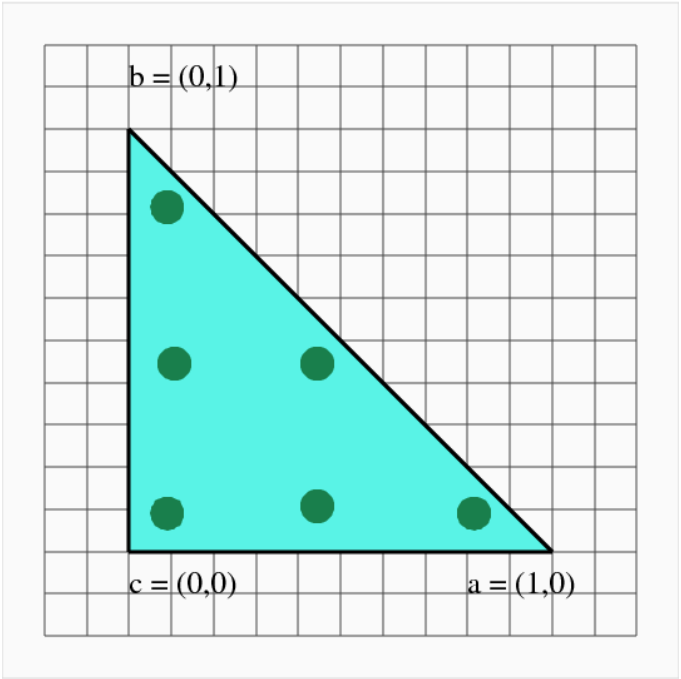
INTEGRALS: A MATLAB Code for the Unit Triangle

```
a = 0.816847572980459;
b = 0.091576213509771;
c = 0.108103018168070;
d = 0.445948490915965;
u = 0.109951743655322;
v = 0.223381589678011;
xvec = [ a; b; b; c; d; d ];
yvec = [ b; a; b; d; c; d ];
wvec = [ u; u; u; v; v; v ];
fvec = f ( xvec, yvec );
area = 0.5;
q = area * wvec' * fvec;
```

INTEGRALS: A Rule of Precision 4

Here is a plot of the quadrature points as they are arranged in the unit triangle T01.

This means we can now approximate integrals over the unit triangle, but what do we do for integrals over an arbitrary triangle ABC, which is what we actually need?



INTEGRALS: Linear Map to a General Triangle

We've solve the problem on the unit triangle T01, but of course our actual region is made up of all kinds of general triangles, a typical one being **TABC** with vertices A, B and C.

Luckily, we can translate our results by using the following linear map, which maps each $(x, y) \in T01$ to $(X, Y) \in TABC$:

$$(X, Y) = A * x + B * y + C * (1 - x - y)$$

To see that the formula works, check these results:

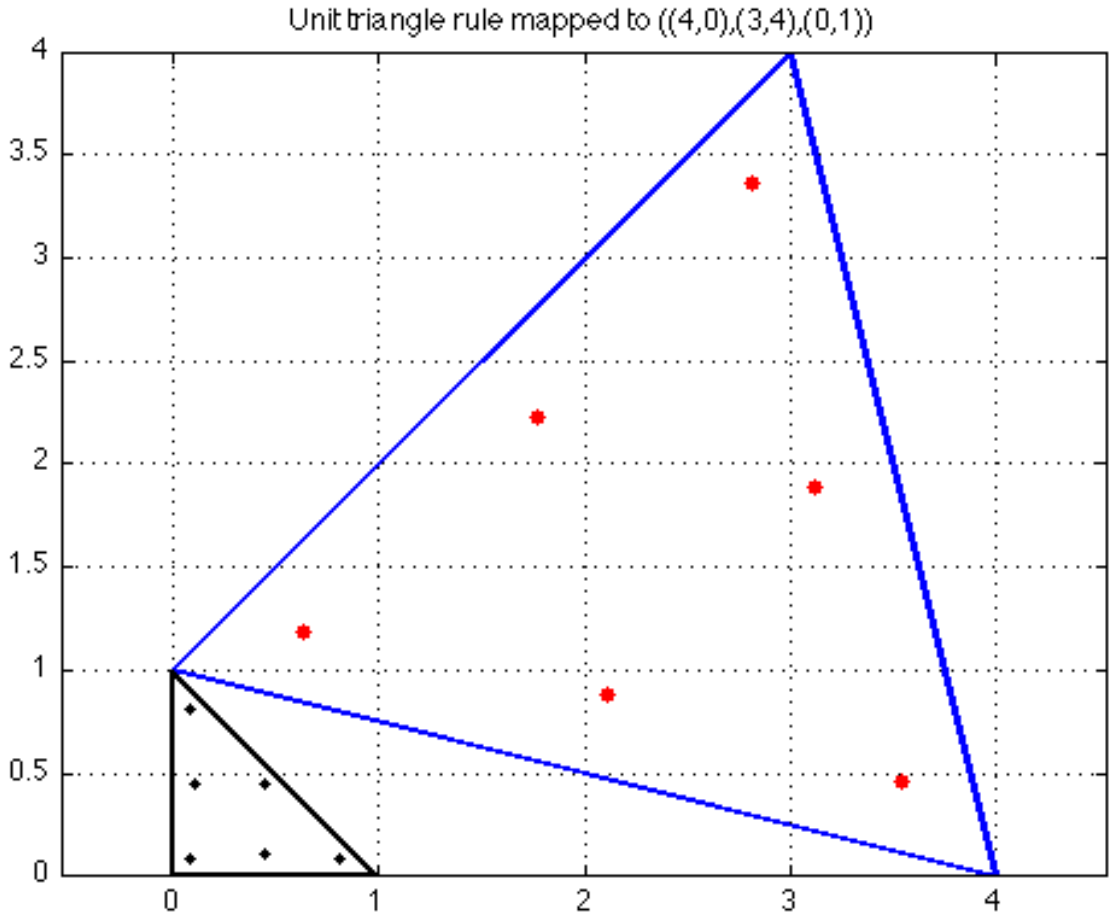
(x,y)	(X,Y)
(1,0)	A
(0,1)	B
(0,0)	C
(1/3,1/3)	(A+B+C)/3 (the centroid)

INTEGRALS: A MATLAB Code for the General Triangle

```
A = [ ax; ay ];
B = [ bx; by ];
C = [ cx; cy ];
Xvec = ax * xvec + bx * yvec + cx * ( 1 - xvec - yvec );
Yvec = ay * xvec + by * yvec + cy * ( 1 - xvec - yvec );
Fvec = f ( Xvec, Yvec );
Area = triangle_area ( A, B, C );
q = Area * wvec' * Fvec;
```

INTEGRALS: The Precision 4 Rule in a General Triangle

TABC = { (4,0), (1,3), (0,1) }:



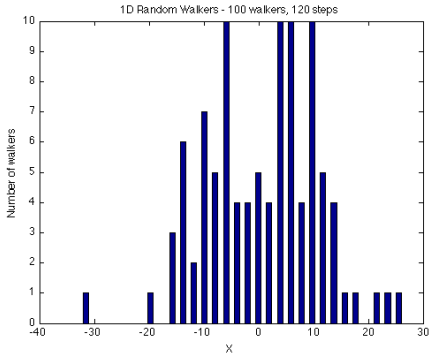
Geometry Algorithms

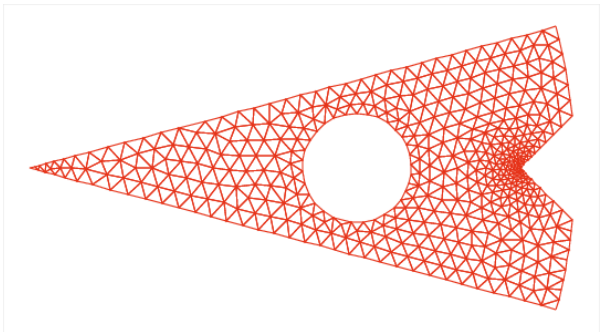
- Overview
- The Points on a Line
- Points NOT on a Line
- Estimating Integrals over an Interval
- Triangles and their Properties
- Triangulating a Polygon
- The Convex Hull
- Triangulating a Point Set by Delaunay
- Estimating Integrals over a Triangle
- **Conclusion**

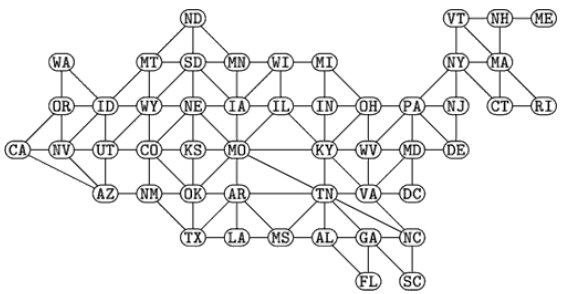
CONCLUSION: We've Shown You the Mountains

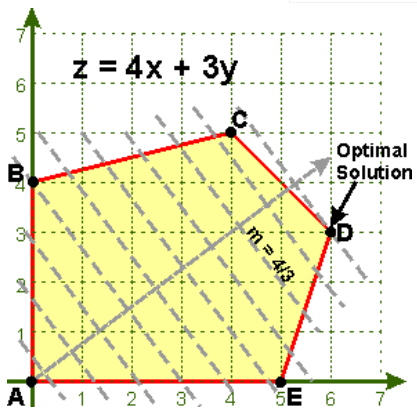
We have come a long, long way this semester, though we've only guided you to the the foothills of many interesting mountains.

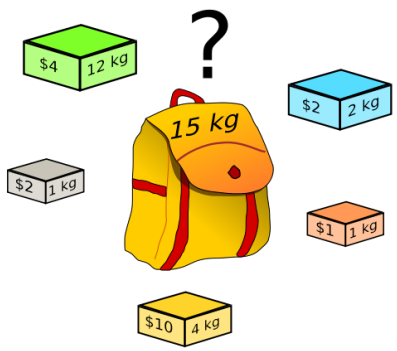
CONCLUSION: Remember Some of those Mountains!











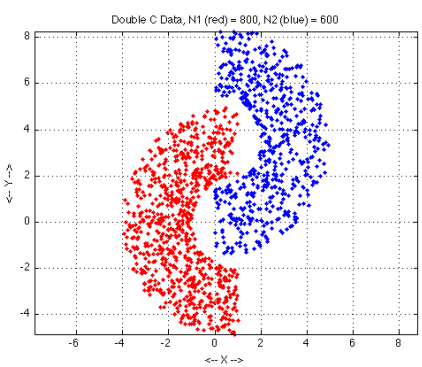


Image using 3 shades

CONCLUSION: The Summit is Up to You

We hope the knowledge and tools we've shown you will enable you to climb the mountains you choose to challenge!