"To ensure the vitality of all its core institutions, the United States must make it a priority of national policy to improve the quality of primary and secondary education, particularly in mathematics and the sciences".
U.S. Commission on National Security in the 21st Century

Algorithmic Geometry

Introducing high school math students to modern software geometry

Pierre Bierre

computer vision

structural biology

CAD

robotics

aerospace

animation

software

GPS

nanotechnology

global competitiveness

Abstract

Math educators and policymakers face an epochal decision – whether the advanced math skills that undergird high tech society will be passed along to the few, or to the many. The answer is a harbinger of our nation's future economic wellbeing and global security.

Geometric problem solving has undergone a quiet revolution over the past generation, in the hands of people who solve spatial problems by writing software. The new geometry is *algorithmic*. The theory foundation has morphed significantly for ease in *thinking and writing algorithms*. This paradigm shift is poised to impact 9-12 math education, with the piloting of algorithmic geometry courseware designed for 11-12 graders. Students learn explicitly how to represent geometric objects and properties in software. Each student writes their own geometry software library in Java, operationalizing and automating their problem-solving knowledge as they acquire it. Problem challenges are explored in STEM topics ranging from robotics to GPS to molecular mechanics.

Angles and slope are passed over as computationally inferior, supplanted by the more elegant *direction vector* (a unit vector representing spatial direction). As a consequence of sidestepping angle, trigonometry is receding in importance. Coordinate rotations are specified in terms of a new set of axes (called a *rotator*). In 3D, direction vectors and rotators alleviate the confusion associated with polar angles and roll-pitch-yaw angle combinations. These aren't mere preferences – direction-vector-based representations are arguably superior both mentally and computationally to angular representations. This white paper explains why 21st century students deserve to be taught this new approach.

Algorithmic geometry also introduces math students to the power of *problem-solving automation*. They learn the discipline of thinking through *fully-generalized* representations and problem solutions (ones able to handle all cases). The payoff is that previous solutions may be reused in any context, opening the door to *layered problem solving*. Layering and reuse make it possible during a 145-hour course for students to ascend to 3D challenges in robot arm motor coordination, wireframe graphics, GPS positioning, molecular mechanics, optics, computer vision, and interstellar navigation.

A two-year public school pilot (n=28 students) was undertaken in 2010-12. The results of these courses are presented. Examples of instructional materials are given. Preliminary answers are offered on how many students can learn to do geometry this way, what level of teaching resources are required, and how best to teach teachers. Pending grant support, development opportunities are expected to surface in the next few years. The purpose of this white paper is to introduce 9-12 math-CS educators to the basics of the algorithmic geometry approach, and begin building an alliance of participating high schools.

Geometry as a Human-Machine Partnership

What is Algorithmic Geometry? Where does it come from? Why is it emerging now? How does it differ from analytic geometry? How does it fit into existing curricula?

The starting point is the assumption that basic geometry constructs be well suited to *thinking and writing software algorithms*. This is a very recent concept, one bound to reshape the set of thinking tools considered central to solving spatial problems.

Our rich geometric heritage, developed and handed down over millennia, builds in the unquestioned assumption of *human* computation. With the advent of software technology, a more powerful set of geometric thinking tools becomes possible.

The driving question is: How relevant are traditional, pre-computational geometry concepts when doing geometry in partnership with software? This question boils down to how nimbly representations originally optimized for brains translate over into software. With two generations of experience, answers are coming into focus. These answers are nuanced.

Points and distances translate easily into software. Accordingly, in our first lab, the student implements 2D Cartesian point vectors and Pythagoras' distance formula.

At ease with these familiar concepts, the student learns how to implement them in Java. Though language details take some explaining, the overall gist is fairly intuitive, and students can absorb much through mimicry.

To represent a Cartesian 2D point as a vector quantity, first, a template for objects of this class is created, saying what information is needed to represent any 2D vector:

```
public class Vec2 {
      double x;
      double y;
}
```

Double precision real variables will house the x and y values (accurate to 16 decimal digits). The distance unit 1.0 represents one pixel width on the screen.

Then, students create and name instances of 2D points, and call a built-in function to draw them on the screen:

```
      Vec2 p1 = new Vec2(100, 200);  // inits p1 to [100, 200]
      p1.draw();                     // draws point on screen
```

Near the end of the first lab session, students will have absorbed enough rudimentary Java to write their own Pythagorean distance function:

```
      public static double distance (Vec2 p1, Vec2 p2) {
        double dx = p2.x – p1.x;
        double dy = p2.y – p1.y;
        double dist = Math.sqrt(dx*dx + dy*dy);
        return dist;
        }
```

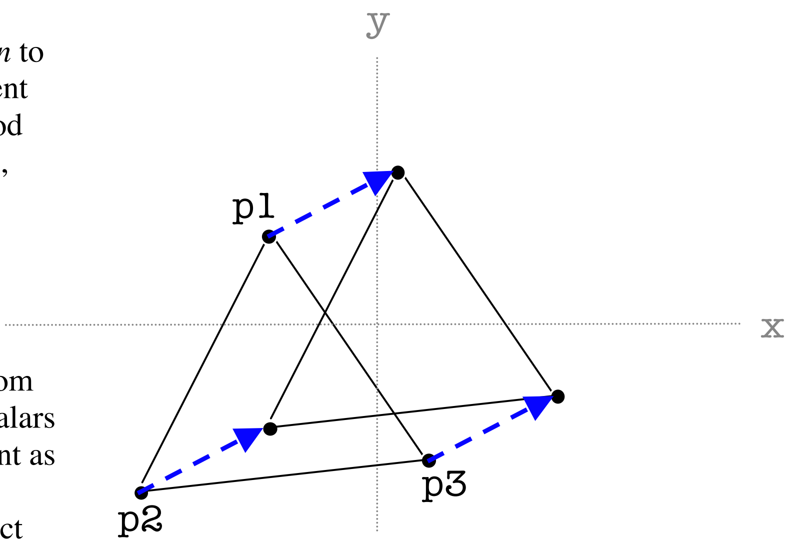
Java is well suited to programming mouse-interactive 2D graphics, and there is probably no more engaging, rigorous and fun way to learn 2D algorithmic geometry. Half an hour into the course, the student is putting dots on the screen, and learning how to make them mouse-editable. This progresses quickly to mouse-editable line segments, leading up to the problem of programming a mouse-draggable triangle.

The challenge given is to move the triangle to a new location by clicking and dragging near its center of gravity. That is, to write a program supporting this GUI behavior.

The elegance of using *vector addition* to translate the vertices becomes apparent (blue arrows). Calling upon a method they've written that adds two vectors,

```
Vec2 c = Vec2.add(a,b);
```

the student is thenceforth freed up from having to deal with the minutia of scalars x and y. She is able to think of a point as a *unitary* object, and operate on it as such. She processes points and object motion using vector operations.



Making the Departure from Angle

After points and distance, the third most basic concept in geometry is *direction.* In classical plane geometry, angle and slope serve this purpose. Neither angle nor slope is ideally suited to computation. The ancients can hardly be faulted for this oversight.

Computers do not "think" the same way as we humans. Back in the '80s, I was a software engineer at Stanford Univ. Neuropsychology Lab, studying feline vision. One day it dawned to me that, as a consequence of representing edge orientation with angle θ, I was having to write exception code. In the computer, θ is just a number, and the first letdown is that > 6.2834, the edge directions represented begin to repeat over again.

The computer does not handle ambiguity skillfully as we do. It's preferable to have a *1:1 relationship* between the representation and the thing it stands for. As a band-aid for θ, we can insist that directions be represented within the bounded range:

$$0 \le \theta < 2\pi$$

After every addition and subtraction of angles, the result value has to be inspected and, if necessary, brought back into standard range. But another problem is looming....

we are forced to swallow a *discontinuity* at 2π. This will complicate deciding when two directions are almost equal. We cannot merely subtract their values.

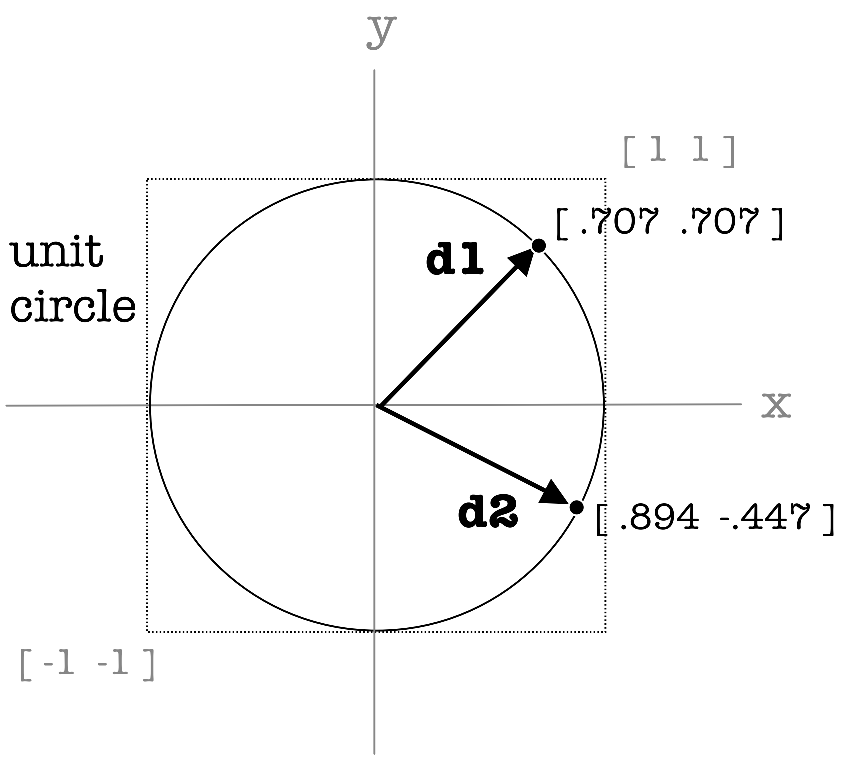
Software writing is better off when spatial direction can be represented with the property of *continuity*....i.e., nearby directions can always be assumed to have nearby values.

After Stanford, it took me another 15 years to clearly articulate the problem with using angle computationally to represent direction:

> There is not quite enough information capacity in a single scalar number to smoothly and unambiguously represent pointing direction in 2D space.

The solution is to use two numbers.....to use *direction vectors* instead of angles.

Any direction in 2D space can be represented as a vector of length = 1, where the tail is at the origin and the head falls on the unit circle. Here are examples of two direction vectors.



Direction vectors (used in place of angles)

Direction vectors are an ideal choice computationally for three reasons:

<u>1:1 Representation</u>: Each possible direction has one and only one direction vector.

<u>Continuity</u>: The value of direction vectors change very smoothly going around the unit circle…there are no discontinuities anywhere.

Scalability: When we get to 3D, the same concept can be applied – instead of a unit circle, 3D direction vectors will fall on the unit sphere.

These representational properties streamline the thought process when creating algorithms. Direction vectors give us a means of representing spatial direction free of messy exceptions. Once you get used to working with direction vectors, the mental process becomes similarly streamlined. Algorithmic geometry is multimedia, straddling the brain, paper and pencil drawing, writing source code, and the computer running it producing graphic results. Compatibility across *all four media* is the major consideration shaping the theory and methodology.

Direction vectors are too hard to calculate in your head (even with the help of paper and pencil). This explains why they receive perfunctory mention in high school texts (if at all). For instance, the direction from point **p1** to **p2** is:

$$\mathbf{d}_{\mathbf{p1}\to\mathbf{p2}} = (\mathbf{p2} - \mathbf{p1})_{\text{norm}} = [\ (\text{p2.x-p1.x})/\ \text{dist}(\mathbf{p1}, \mathbf{p2}) \quad (\text{p2.y-p1.y}) / \text{dist}(\mathbf{p1}, \mathbf{p2})\ ]$$

Angles are much better suited to brain computation. But direction vectors are superior when the computer is crunching the numbers. They are a nimble fix to the overcompression of information in angle, and require no more effort than to once and for all write source code to compute the direction from **p1** to **p2**:

```
public DirVec DirVecOf(Vec2 p1, Vec2 p2)  {//p1→ p2
     if  (identical(p1, p2)) return null;
     return new DirVec(normalize(diff(p2, p1)));
```

Don't expect the elegance of this approach to jump out of the page and seduce you on the spot. It takes some experience applying direction vectors algorithmically before your allegiance to angle softens up. Where we want to help you get to (and your US students) is a new level of representational sophistication where angles and direction vectors coexist along a human-machine continuum, each playing to their strengths, and avoiding their respective limitations. A student straddles this continuum by writing conversion functions for going back and forth between θ radians and direction vector **d** = [x y].

Reinventing the 2D line equation

The next problem the student is confronted with is referred to in computer graphics as *line hit testing* -- Write an algorithm that decides when the mouse is being clicked on a visible line on the screen. The algo must be written to work with *any possible line*.

When *drawing* a line segment, the system's graphics software takes care of filling in all the pixels between the endpoints. Line hit testing is an *input processing* problem, and requires the student's program know something about where the line is. How shall a line be represented? For simplicity, we consider the line to extend infinitely. The following question provides the starting point for thinking about how best to represent lines (and other geometric objects):

What is invariant numerically about the points on the line?

In our formative years, we're taught that a straight line is represented mathematically as the set of points [x y] satisfying the equation:

$$y = m x + b$$

where m is the slope (dy/dx), and b is the y-intercept.   What's wrong with this theory?

The problem is that it cannot represent *vertical* lines.   For such lines, the above equation seriously bombs, first because calculating the slope will force a divide by zero, which generates a computer error.   Second, where does a vertical line intercept the y-axis?

The slope-intercept formula isn't a problem for humans doing pre-computational, paper and pencil geometry because we are good linguistic exception handlers, and adapt to vertical lines as a special case, by writing:

$$x = 7$$

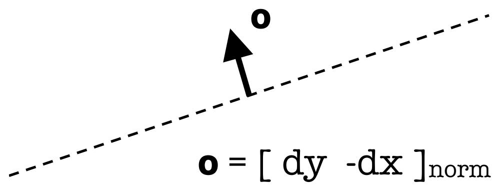
From an algorithmic standpoint, changing the *form* of representation to handle special cases is a sign that the choice for representation is not general enough.   The ideal is to represent all cases in the same manner with no exceptions.

The problem with slope (dy/dx) is that a single *scalar* number does not have quite enough *information capacity* to comfortably represent all possible line orientations.   A similar problem arose trying to shoehorn 2D direction into scalar angle θ radians. In computation, the resulting singularities posit ugly exceptions that are better avoided.

The remedy is to allow the use of more than one number in representing line tilt. We use a *direction vector* to solve this problem.

We use the direction *perpendicular* to the line to represent its *orientation* **o**:

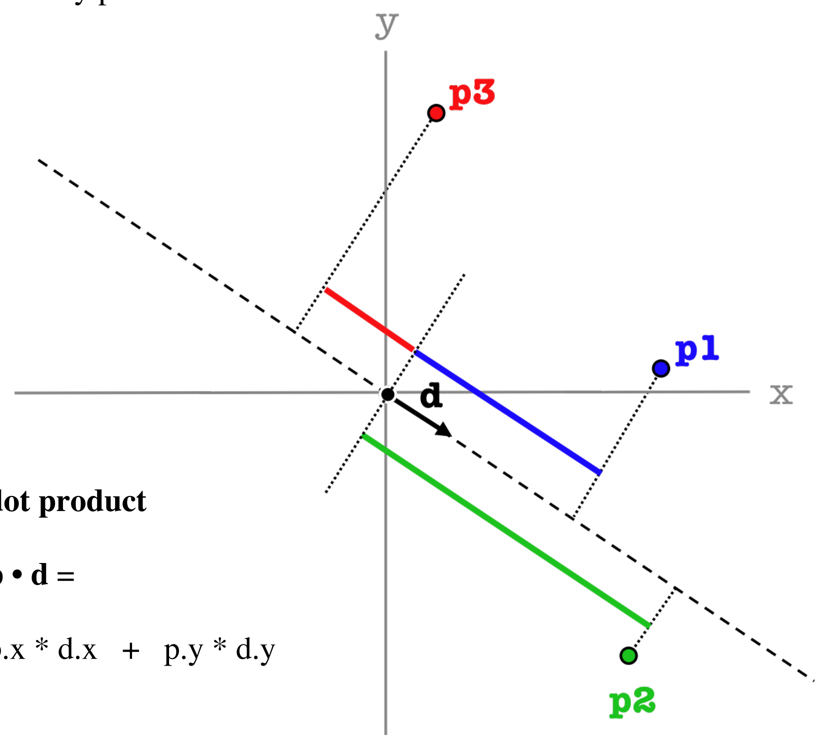
By allocating *two* real numbers to represent line orientation, the exception for vertical lines is eliminated, and all line orientations can be represented in a uniform manner.



Line orientation o (replaces slope)

As can be seen, dx and dy are still the key pieces of information needed to compute line orientation. The *faux pas* we avoid is collapsing dx and dy into single number dy/dx. Division destroys information. ∞ works fine for brains, but in the computer, there is no way to represent such an indistinct quantity. ∞ just doesn't compute. This is one of the adjustments we make in the switch to algorithmic math.

Next, the student learns about the dot product of two vectors. We introduce the dot product non-traditionally, by studying the dot product of a direction vector and an arbitrary point:



dot product

p • d =

p.x * d.x + p.y * d.y

We let direction vector **d** play the role of an alternative x-axis.

The dot product

p • d

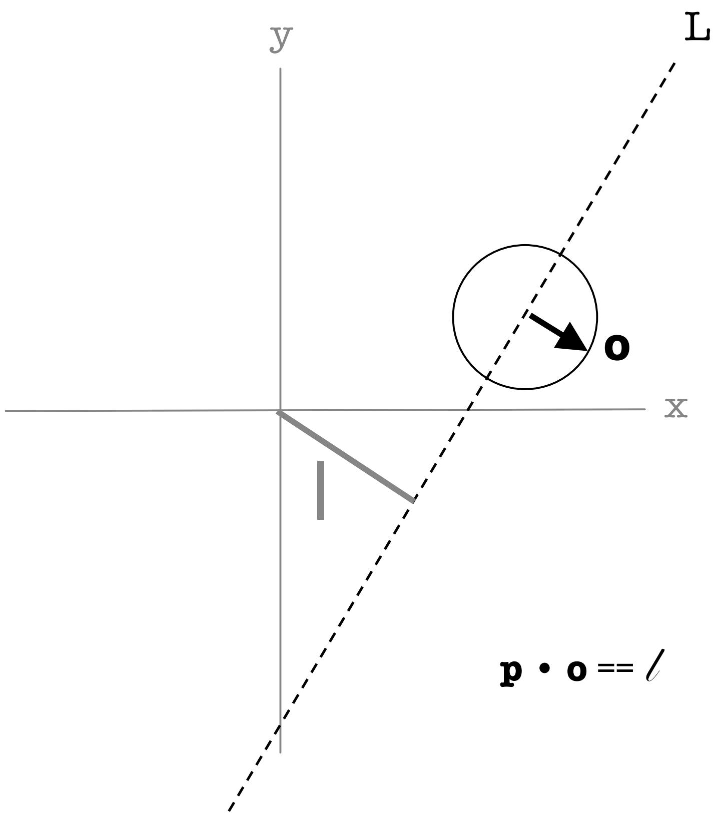
serves to compute the new coordinate of any point **p** along this new axis **d** (colored distances indicate new coordinate lengths).

Equipped with line orientation and the dot product, we may proceed to the vector-based invariance formula for points on an infinite straight line.

If **o** is the line's orientation, and | is its signed distance from the origin along direction **o**, then for every point **p** on line L:

p • o == ℓ

Graphically, think of line L's orientation **o** as an alternative x-axis. Every point on line L projects onto the same coordinate (ℓ) along this new axis direction **o**.



Invariance of points on a line

This *vector line predicate* is a close cousin to the standard form (and Hessian), but surpasses any purely algebraic formulation in that its numeric features have clear spatial meanings that the student can latch onto. The two features needed to represent an infinite 2D line are its orientation **o**, and its signed distance from the origin ℓ. This leads to the following software representation:

```
public class Line {
     DirVec o;   // orientation (perp. direction)
     double l;   // signed distance from origin (along o)
}
```

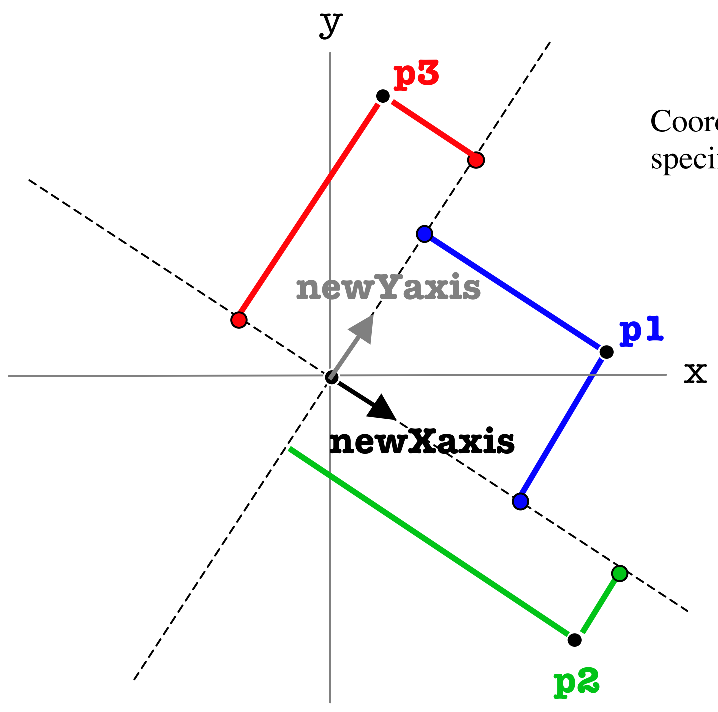
Using this representation, the line hit testing problem may be solved in a few lines of code. The scalar result $\mathbf{p} \bullet \mathbf{o} - \ell$ gives the *signed proximity* of a point to a line (how close is it, and on which side it falls):

```
public double signedProximity (Line L, Vec2 p) {
     return dotProd(p, L.o) – L.l;
}
```

Hit testing the mouse-click for proximity to a line falls out naturally at this point. The student merely needs to pick a reasonable pixel distance forgiving pecking inaccuracy, and then decide if abs(signed proximity(Line, mousePos)) falls within that many pixels.

Coordinate Rotation using **newXaxis** instead of angle θ

Applying the 2D vector line predicate, a student begins to appreciate the power of the dot product to transform coordinates for an arbitrary new axis direction. This sets the stage for introducing generalized 2D coordinate rotation, specified by **newXaxis**:



Coordinate rotation specified by **newXaxis**

Students should already be familiar with coordinate rotation as the process of adopting a new coordinate system, transforming points into *new* coordinates. In algorithmic geometry, the "amount" of rotation is specified *by simply saying where you want the new x-axis to fall*. You specify **newXaxis** as a direction vector. The advantage is that a new axis direction may be obtained from points without invoking angles and trig. The inverse transcendentals arccos, arcsin, arctan are messy computationally (mathematically, they are not even functions). Direction vectors are always well-behaved computationally.

Having already automated the dot product function, the student is able to write a workhorse method to rotate point **p** for **newXaxis**. This method is highly streamlined compared to traditional coordinate rotation specified via angle (and requiring trig).

```
public Vec2 rotate (DirVec newXaxis,  Vec2 p )  {
   DirVec newYaxis = rot90CCW(newXaxis);
   return new Vec2(dotProd(p, newXaxis), dotProd(p, newYaxis));
}
```

Another way to think of 2D point rotation working, using matrix operations, is to conceive of a 2x2 matrix **R**, having as its column vectors the new x and y axis directions:

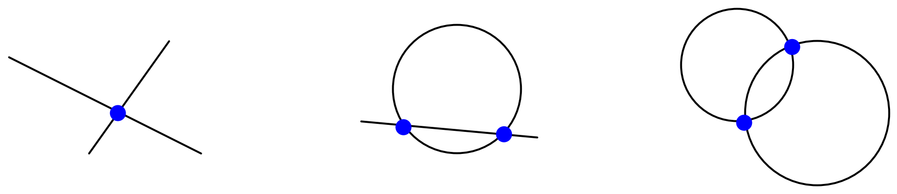
$$\mathbf{R} = \begin{bmatrix} \mathbf{newXaxis} & \mathbf{newYaxis} \end{bmatrix}$$

and implementing point rotation as a vector **p** dotted with **R**:

$$\mathbf{p'} = \mathbf{p} \bullet \mathbf{R}$$

The role that **R** plays in software invites us to refer to such an object as a *rotator*. This concept will become very potent when the student segues into 3D. At that point, she will arrive prepared mentally to rapidly accept the use of 3D rotators. These rotators make mental mincemeat out of working with freely rotating bodies. They represent a *quantum leap* in mental clarity and elegance over roll-pitch-yaw angles, while putting to bed all the computational and representational hassles associated with the latter.

At this point in the course, the 2D representational foundation is built, and attention turns to solving the basic intersection algorithms involving lines and circles:



2D intersection problems solved algorithmically

Typically, these intersection problems will each require algorithms stringing together 3-5 mental steps and 5-15 new lines of code. Considering that the algo must handle all possible cases, a degree of mental parsimony is being harnessed in these solutions.

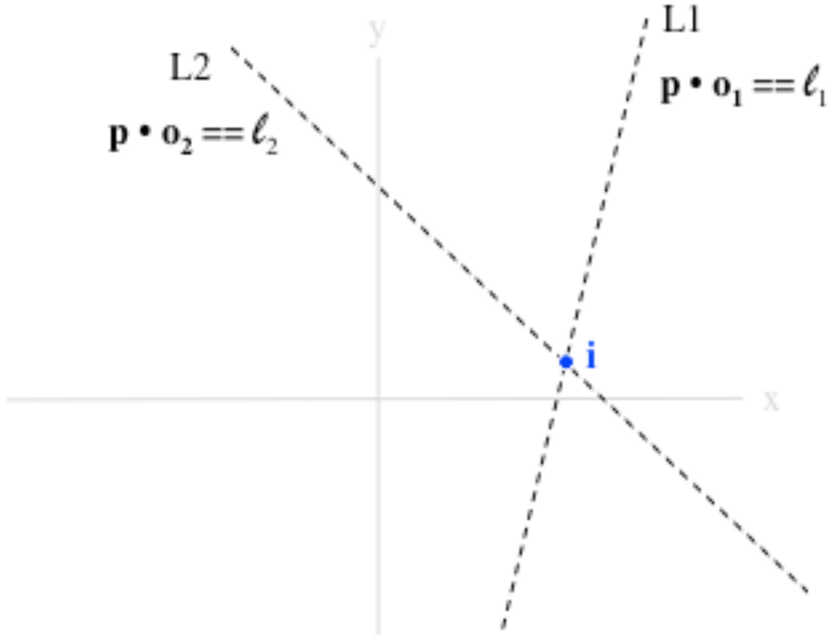
Let's look at **the intersection of two lines** as an example. Problem statements are graphical. Students have to process the lines using their *vectorized* representations.

Given:

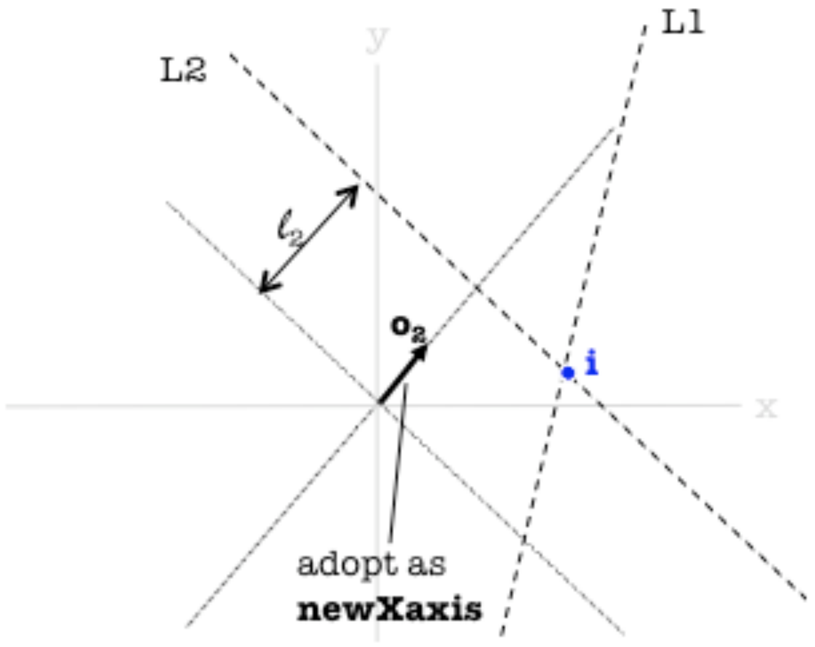
L1 = [$\mathbf{o_1}$ ℓ_1]

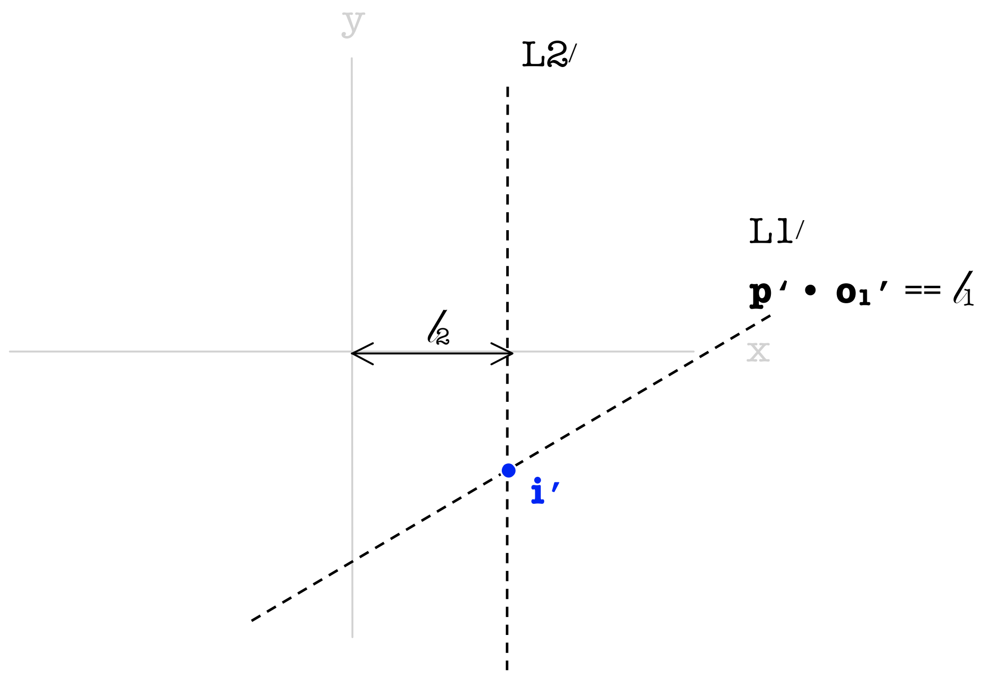
L2 = [$\mathbf{o_2}$ ℓ_2]

Solve for: **i**


L1
$\mathbf{p} \bullet \mathbf{o_1} == \ell_1$
L2
$\mathbf{p} \bullet \mathbf{o_2} == \ell_2$
y
x
i

Students are coached to think of a special case that would be easier to solve. Wouldn't it be easier to solve if one line were parallel to an axis? Coordinate rotation can be used to morph any problem into this special case. Let's pick one of the lines, and rotate coordinates under its influence to make that happen....how about L2? What if we use L2's orientation to define the **newXaxis**:





Line intersection problem "rotated" into easier special case

Rotating coordinates under the influence of L2 makes L2' stand *perfectly vertical.*

Rotation does not change a line's signed distance from the origin. So, in the rotated space, the point **i'** lies on the vertical line having x' coordinate ℓ_2:

$$i'.x = \ell_2.$$

We rotate L1 → L1' computationally to make use of its information. Its distance from the origin is preserved, so all we need to do is rotate its orientation:

$$\mathbf{o_1}' = \text{rotate (newXaxis, } \mathbf{o_1}\text{) = rotate (} \mathbf{o_2}, \mathbf{o_1}\text{)}$$

All we are missing now is i'.y. The line equation of L1', given an x' coordinate, determines a y' value. A little algebra solves for y':

$$[\ \ell_2 \quad i'.y\ ] \ \bullet\ \mathbf{o_1'} \quad = \ell_1$$ (line equation where $i'.x = \ell_2$)

$$\ell_2 \quad o_1'.x \quad + \quad i'.y \quad o_1'.y \quad = \ell_1$$ (expand the dot product)

$i'.y = (\ell_1 - \ell_2 \, o_1'.x) \, / \, o_1'.y$ (rearrange, solve for i'.y)

We now have both coordinates for intersection point **i'**. After that, we just need to unrotate **i'** → **i** so that the result comes out in original coordinates. A sibling function to `rotate` called `unrotate` will exactly undo the transform when handed the same value of **newXaxis**.

The mental steps captured in the sketches and algebra serve as the *specification* for the source code algorithm. Each student translates their mental steps and sketch into Java:

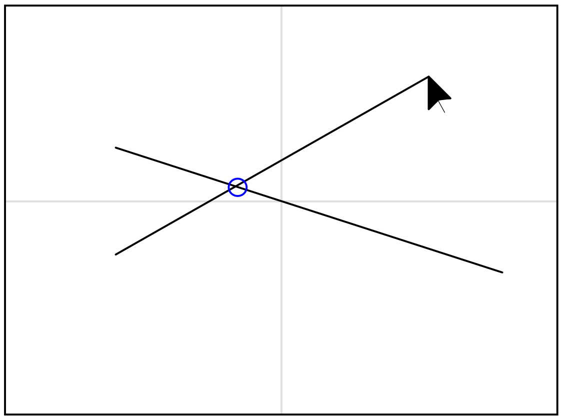
```
public static Vec2 IntersectPtOf(Line L1, Line L2) {
          if (parallel(L1,L2)) return null;

          // rotate coords so that L2' is vertical
          DirVec newXaxis = L2.o;
          DirVec L1_o_prime = Vec2.rotate(newXaxis, L1.o);

          // find i_p, the intersect pt in rotated coords
          Vec2 i_p = new Vec2 ();
          i_p.x = L2.l;
          i_p.y = ( L1.l - L2.l * L1_o_prime.x) / L1_o_prime.y;

          // unrotate i_p back to original coords
          Vec2 i = Vec2.unrotate(newXaxis, i_p);
          return i;
          }
```

Notice that the degenerate case is handled up front. A null result signifies no intersection. The student solves `parallel(L1, L2)` as a separate algorithm.

The final step is to test-run 2D line-intersection using on-screen graphics. A pair of lineSegs whose endpoints are mouse-editable serves as the testbed. The student programs in a small circle to highlight her algo's computed intersection point.

As the student drags one lineSeg around, the circle tracks with the "X" (unless there is a bug).

Interactive graphics testing throws *dozens* of test cases at the algorithm per second. It is a very efficient way for the student to become confident a general, automated solution has been achieved.



Test of line intersection algorithm

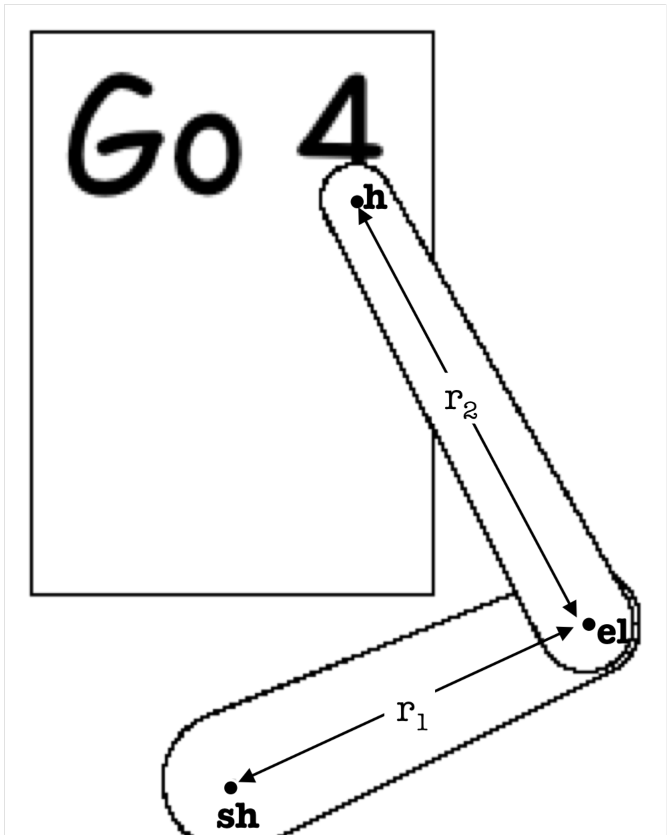
The intersection of *two circles* is approached in a similar fashion, using both coordinate translation and rotation to massage the problem toward a friendly special case. The finished algorithm is a *gateway* result, opening the door to solving a whole host of advanced problems in robotics, graphics and navigation. There is a palpable sense of empowerment felt by the student as they conquer this problem, aware that they are no longer hamstrung by the idealizing assumptions that limit paper and pencil geometry. The intersection-of-two-circles problem crosses a threshold into a new realm of cybermath.

The methodology relies heavily on visual thinking, and understanding how visual manipulations correlate to vector processing operations. Ability to draw quality sketches is a key success factor. This is because the “aha” moment in algorithmic geometry is usually harvested in paper and pencil mode. Coding the algorithm is all downhill.

Each student is required to keep a notebook of sketches. These comprise graphic specifications of the algorithms, and are the key to understanding how the lines of code work to solve the problem. They trace the thought process leading up to coding.

At the midpoint of the course, the student will have written a 2D vector geometry library comprising about 8 classes, 120 algorithms and 750 lines of code, covering points, distance, directions, lines, lineSegs, triangles, circles, arcs, translations and rotations.

A Problem Challenge in Robot Motor Coordination

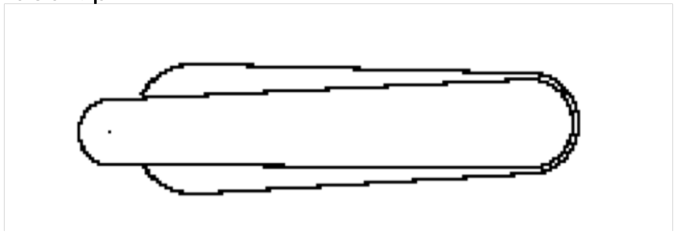


Here is a robotics challenge students may elect to solve, applying their new understanding of 2D:

The painting hand at **h** is maneuvered by the shoulder motor at **sh** and elbow motor at **el**. The upper-arm length is r_1, and the forearm length is r_2. You are given the coordinates of **sh**.

For any specified point **p** the paint hand **h** can reach, what shoulder and elbow angles $[\ \theta_{sh}\ \ \theta_{el}\ ]$ should be assumed?

The elbow motor angle is with respect to the upper arm on which it is mounted. The home positions ($\theta == 0$) of the motors are shown in the folded-up arm:



Solving this problem involves 4 mental steps and about 10 lines of new code. The "aha" moment comes when the student sees in their sketch that the elbow is constrained by fixed distances from 2 points ([**sh** r1] , [**h** r2]). Therefore, **el** falls out from the intersection of two circles. That algorithm is already a done deal....off-the-shelf!

Once **el** is obtained, direction $\mathbf{d}_{\mathbf{sh} \to \mathbf{el}}$ can be computed, then converted to shoulder motor angle θ_{sh}. The student takes for granted his lower-level functions already written.

The elbow motor angle requires one additional step: After $\mathbf{d}_{\mathbf{el} \to \mathbf{h}}$ is computed, it has to be rotated into the upper arm's moving frame-of-reference (defining 0.0 radians for the elbow motor). In the folded up arm, we can see that this newXaxis always points from the elbow back toward the shoulder ($\mathbf{d}_{\mathbf{el} \to \mathbf{sh}}$). Directions are rotated just like points:

$$\theta_{el} = \text{angleOf(rotate (}\mathbf{d}_{\mathbf{el} \to \mathbf{sh}} , \mathbf{d}_{\mathbf{el} \to \mathbf{h}}\text{))}$$

This problem is typical of the degree-of-difficulty a student can manage after a half-semester of algorithmic geometry. To the uninitiated, this may seem audacious and unrealistically advanced for high school juniors and seniors. But if you interview the initial students who've gone through the pilot, they explain the success of the approach in these terms:

• the geometry concepts are small in number, elegant, powerful, and designed to fit together like a jigsaw puzzle.

• the Java is a little strange initially, but is picked up by osmosis in the first 2 weeks

• theory concepts are immediately put into practice hands-on writing interactive graphics

• the algorithms written pile on synergistically...in every algo, we leverage past success, always moving to a higher level – the automation of math problem solving is way cool!

Being Swept Up into 3D

The point of spending 8 weeks learning 2D vector software geometry is twofold. The student is being prepared intellectually for an easy glide up into 3D. In addition, the student's 2D library will serve as a workhorse for solving 3D problems.

In the pilot, the transition was surprisingly spontaneous. As soon as I let down the gauntlet to start into 3D, my students made a copies of their Vec2 class library, named it Vec3, and with no prompting, rewrote every vector function to work with [x y z]. They surmised how a 3D direction vector is defined and computed. The only help needed on DirVec3 involved the converter from polar angle [ϕ θ] → **d** = [x y z]. With some coaching, their 2D converter [θ] → **d** = [x y] did most of the heavy lifting.

The first structured object taken on in 3D is the plane.

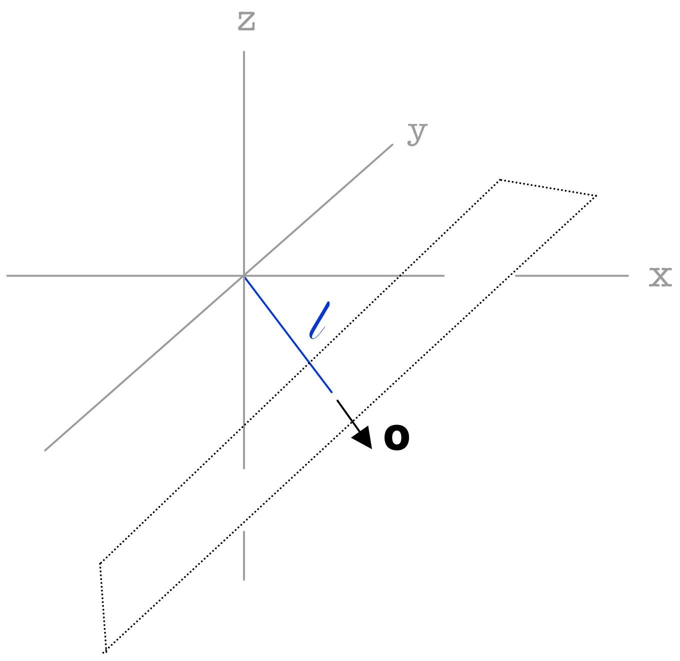
What is invariant about the points on a plane?

The analogy to the 2D line is unmistakable. The plane has a *directional* feature (its tilt), and a *positional* feature (its signed distance from the origin).

We can represent it using the same predicate as the 2D line:

$$\mathbf{p} \bullet \mathbf{o} == \ell$$

It will take introduction of cross product theory before being able to compute a plane from 3 points.



$$\mathbf{p} \bullet \mathbf{o} == \mathbf{l}$$

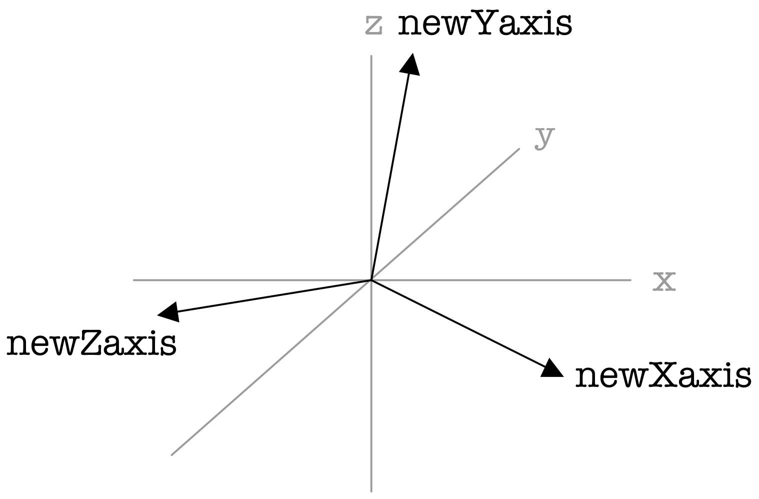
Invariance of points p on a plane

As motivation for the cross product, the student is challenged to extend the concept of coordinate rotation to 3D.

Just as in 2D, we can specify a general-purpose coordinate rotation by supplying a valid new set of axes.

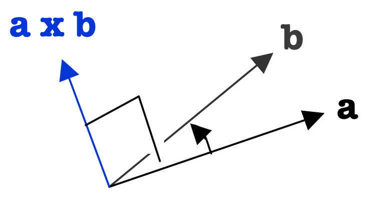
Students can see their way to rotating **p** → **p'**. There is one more dot product to be computed for the z-coord.

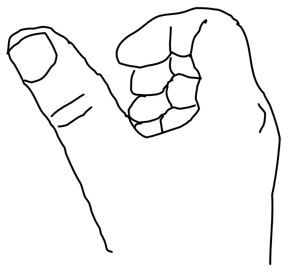
What is not obvious is how to compute such an axis set (what we in software call a *rotator*, the operational equivalent of an orthogonal matrix).



Rotator R = [**newXaxis newYaxis newZaxis**]

For this, the student is shown the cross product of two direction vectors **a b**, and how **(a x b)**$_{\text{norm}}$ computes their mutual orthogonal direction:





The cross product is the key vector function needed to write algos that produce rotators on demand, for example:

```
Rotator R1 = RotatorForNewXandYAt(newXaxis, newYaxis);

Rotator R2 = RotatorForNewZAt(newZaxis);
```

In the latter case, **`newZaxis`** underspecifies the rotator, but this function is quite useful nonetheless in cases where we are willing to let the *software* pick **newXaxis** and **newYaxis** *arbitrarily.*

Point Invariance Predicate of a 3D line

What is invariant about all the points **p** on an infinite 3D line? The conventional approach (web and textbooks) springs from point pair **p1 p2**, and appeals to a linear combination of them parameterized by t:

$$\mathbf{p} == \mathbf{p1} + t\,(\mathbf{p2} - \mathbf{p1})$$

A *point invariance predicate* is a formula you can plug arbitrary point **p** into, deciding if **p** is included in the object. The above formula fails as a predicate! Why? The value of t is unknown. To compute whether **p** is on given line [**p1**, **p2**], we'd have to also be given t. This representation is what a computer scientist would call a weaker *generative* form....it suffices to generate points on the object, given values for t:

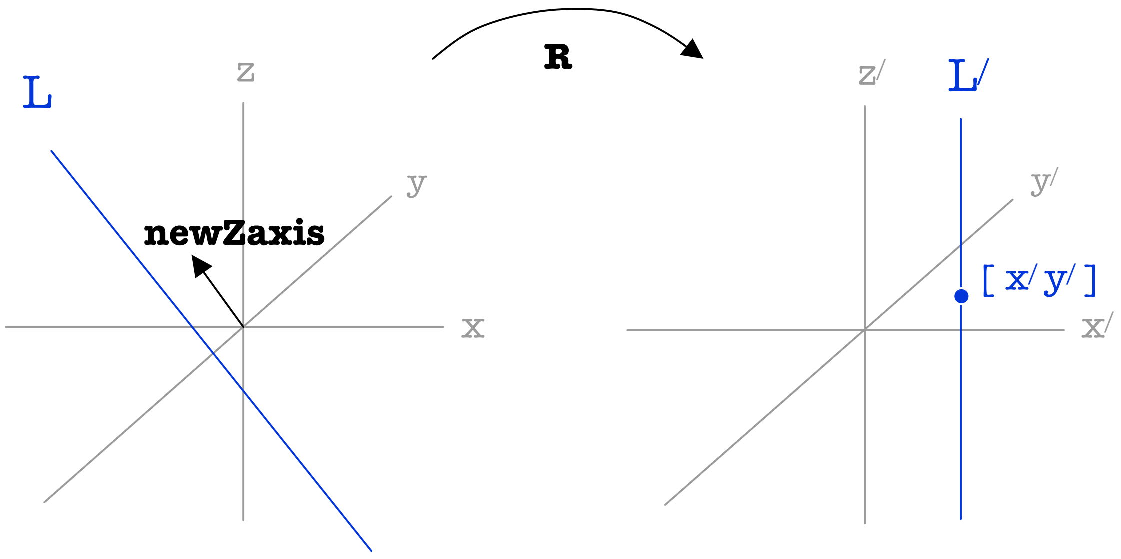
$$\mathbf{p} = \mathbf{p1} + t\,(\mathbf{p2} - \mathbf{p1})$$

but lacks the strength of an invariance predicate. The compare (==) and assignment (=) operators above mean totally different things in computer science. Precomputational geometry makes no such distinction. Predicates calculate true-false *decisions*, such as whether an arbitrary point **p** is included in an object.

In algorithmic geometry, when deciding how best to represent an object in software, students are coached to seek out the object's *point invariance predicate*, a formula that can compute whether arbitrary point **p** is included in the object.

How can we state the invariance of points on a 3D line in such predicate form?

The invariance of all points on line L is as follows: If coordinates are rotated in such a way that **newZaxis** is chosen *parallel* to the line, then in the new coordinate system, the line will stand perfectly vertical. All points along it will share an invariant [x' y'] value.



We represent an infinite 3D line as a rotator **R** and an invariant [x' y'].

The invariance of points on L can now be captured. For any point **p** on L,

rotate (**R** , **p**) == [x' y' --]	3D line predicate

How can **R** and invariant [x' y'] be computed? Two points are sufficient to determine a line. Assume we are given **p1** and **p2** (**p1** != **p2**). (!= means "not equal to")

First, we calculate the direction $\mathbf{d}_L$ going from **p1** → **p2**. This is just the normalized vector difference **p2** – **p1**:

$$\mathbf{d}_L = (\mathbf{p2} - \mathbf{p1})_{\text{norm}}$$

R wants to specify a rotation where this direction $\mathbf{d}_L$ defines the **newZaxis**. This is the coordinate rotation that will make the line stand perfectly vertical (aligned to z' axis):

```
Rotator ZAlignRotator = RotatorForNewZAt(d_L);
```

Then, rotate either given point (**p1**or **p2**) by this amount to calculate the invariant [x' y']:

```
[ x' y' -- ] = rotate (ZAlignRotator, p1);
```

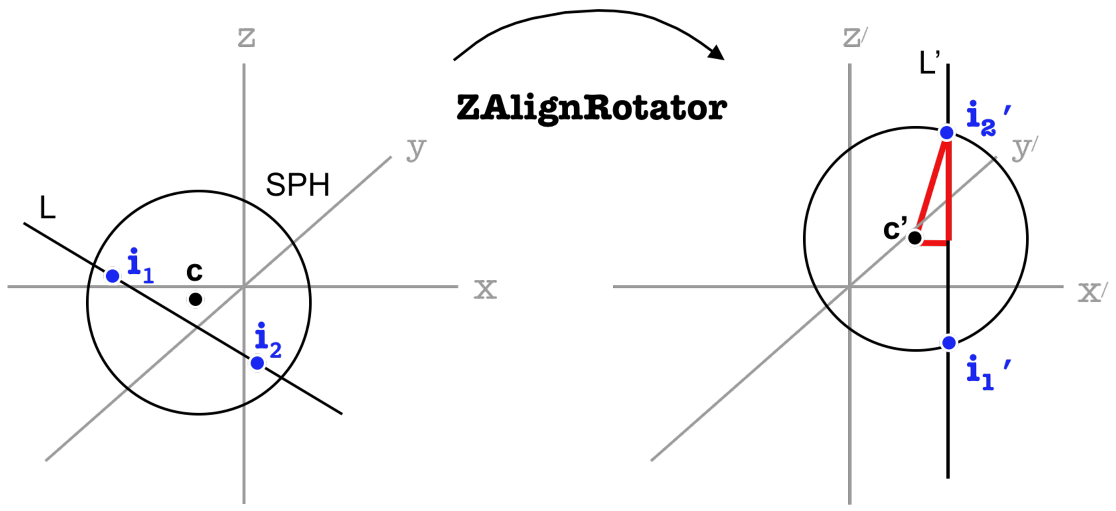
The "constants" in the object's point invariance predicate pin down all the information needed to represent the object:

L = [ZAlignRotator   invariant [x' y']]   (representation of 3D infinite line)

In Java:

```
public class Line3D  {
    Rotator ZAlignRotator;
    Vec3    invariantXY;
}
```

The advantage of this approach becomes apparent once you begin tackling problems involving 3D lines. For instance, on our current final exam, students are asked to design an algorithm solving for the intersection of arbitrary sphere SPH and line L.



A straightforward solution awaits if you rotate coordinates making the line stand vertical. The rotator that does so comes "free" in the line's representation. Rotating the sphere merely entails rotating its center **c** → **c'**. You should be able to solve the special case on the right. The [x' y'] coordinates of the intersection points are the invariantXY feature of line L. Their z' coordinates fall out using **c'** and Pythagoras applied to the red triangle. Unrotate the solution points by ZAlignRotator to finish up.

Generating points on a 3D line using a parameterization variable is no problem. In the rotated space where the line stands vertical, the z' coordinate parameterizes all the points along the line...you just pick a z', tack it onto invariant [x' y'] and unrotate:

```
public generatePoint (z_prime) {
   return unrotate (ZAlignRotator,
                    new Vec3 (invariantXY.x,invariantXY.y, z_prime));
```

}

The 3D object intersections students solve algorithmically include the following:

Line-Plane	**Plane-Plane**	**3-Plane**	**Sphere-Sphere**
Sphere-Circle3D	**3-Sphere**	**Sphere-Plane**	

3-Sphere intersection forms the basis for GPS positioning. When students master this algorithm toward the end of the course, there is a "heady" feeling analogous to physics students of the '50s having teased apart how a transistor works, or a century earlier, Victorian kids having figured out how basic electricity works.

The rarified air these kids breathe has much to do with the fact that they know they are already surpassing at a young age what very few older adults can understand. Such highly cerebral, transcendent experiences are exactly what is needed to propel young people to become scientists and inventors.

A menu of 3D problem challenges (mini-projects) awaits the student the last 2 weeks of the course. Students may elect 2 topics from the following menu (which is expanding every year):

3D wireframe graphics	Robotic gas pump attendant
CAD pipe rendering graphics	Optical reflection and refraction
Computer Vision	Molecular Brownian Rotation
Interstellar Navigation (by camera star tracking)	

In each challenge, the student solves an advanced 3D geometry problem that opens the door to competence in an area of science-engineering. For example, for molecular biochemistry modeling, you are asked to devise an algorithm that confers a perfectly *random* 3D rotational orientation to a virtual molecule. Or, in computer vision, infer the location of a 3D point by acquiring it in images taken from two different vantage points. Or, in interstellar navigation, deduce a spacecraft's 3D attitude by sighting two very distant reference objects (requiring application of rotational inference[1]).

These challenges can typically be solved in less than 20 lines of new code, but may require the better part of an hour of trial and error in paper and pencil mode to solve the key geometry puzzle. When the solution finally runs on the student's computer, the program is exercising a large portion of the student's geometry library going back to the beginning lab. The edifice of one's mental effort over dozens of hours is somehow compacted into a split second of focused brilliance, vanquishing a college-level (or advanced-degree) problem. Students expectably come away with a surge of confidence about their ability as problem-solvers.

For example, the interstellar navigation problem is a PhD-level problem (judging by the few space scientists who have published papers addressing it). Students who can problem-solve at this level before entering college are going to see themselves as math whizzes. This is the type of learning experience we educators should be shooting for.

Results of Initial Proof-of-Concept Course

The first teenagers to study algorithmic geometry completed a proof-of-concept pilot the summer of 2005. The initial testbed involved two students freshly minted from high school, and in recent possession of their graduation laptop gift. The purpose of the 9-week, 72-hour course was to obtain an initial reality check on the feasibility of imparting all the novel aspects of pedagogy, primarily the revised theory foundation and the uphill of some computer programming.

The syllabus (see Appendix A) was based on a draft version of *Flexing the Power of Algorithmic Geometry*[2]. The software toolset was based on CodeWarrior/Java (we've since switched to Eclipse/Java). Hand-drawn posters were prepared for theory lecture graphics. We rotated the class meeting amongst our three homes, with three classes a week totaling 8 hours. There was no homework, a norm we have adopted in order that students benefit from working on problems in an environment with full technical support.

The class rhythm consists of a short theory lecture-discussion followed immediately by a programming lab exercise where students individually write code implementing the theory, and then test their solutions for correctness by means of interactive graphics.

Appendix B presents a walkthrough of a complete problem encountered 3/4 way through the course – the intersection of a sphere and 3D circle.

One student had never programmed before, and the other had one year of Java. The two students are profiled briefly here:

"David W." Finished grade 12, SAT-M = 670, received C in geometry, no programming experience, team leader 2 years in FIRST robotics competition. Currently: Engineering junior at Purdue.

"Cliff B." Finished grade 12, SAT-M = 720, Logo Turtle Graphics (k-5), 1-year AP CS Java. Currently: junior in Mechanical Engineering at Cal Poly SLO.

Toward the end of the course, both solved the problem of GPS positioning (intersection points of 3 overlapping spheres), and interstellar navigation by star tracking (rotational inference, directional triangulation). The source code libraries written by these two students demonstrate a proficiency that would be judged *extraordinary* at the pre-college level. Inasmuch as the students wrote their Java code without prompting, it is inescapable that the student possesses a rigorous, hands-on understanding of introductory vector math.

How is it Different?

Let's now try to summarize, and in doing so answer the seminal questions about algorithmic geometry posed at the outset. What is it? Where does it come from? Why is it emerging now? How is it different? And, where does it fit in?

It is a modern style of geometry undertaken as a close-knit partnership between human and computer. Using software programming, the person translates paper and pencil sketched solutions into working algorithms that crank out numerical answers, handling all possible cases of input. Previously solved problems are fully automated, and sit at the ready to be called into action to assist in solving the next problem. This pattern of solution reuse leads to highly layered, highly leveraged activity, in which the *aggregate* of one's past mental effort is brought to bear in an instant solving a daunting problem, but the steps leading up to it involve ordinary mental effort focused on bite-sized chunks. The synergy going on in this partnership is a blend of mathematics, computer science/AI and cognitive science/psychology. You could call it applied cybernetics.

The *thoughtware* vested in the partnership straddles grey matter, sketches on paper, source code, and the computer behavior while exercising it. The geometry becomes operational in silicon, and springs to life in vivid color on the screen.

The foundational concepts are steeped in the classical Greek distance formulas, Cartesian points, and direction vectors (popularized by mathematical physicists[3]). While points and distances are represented in software the same way as has been done for 50 years, slope and angle are more recently supplanted by direction vectors, which stand up better to computer-based representation. Accordingly, all *directional* properties, object features, and operations are restandardized using direction vectors. Coordinate rotation is done using rotators, which specify a set of new axes (basis vectors). Rotators rotate points and directions by dot product projection onto new axes. Angle is demoted in importance, and with it, trigonometry. 3D rotators streamline and simplify coordinate rotations that previously had to be represented using combination roll-pitch-yaw angles. The singularity, ambiguity and asymmetry headaches inherent in representing freeform solid-body rotation with a set of angles are burdens no 21st century student will miss.

Object representations are arrived at by asking what is invariant about all the points on an object, and striving to capture the answer in a computable *point invariance predicate*. This approach points the way to streamlined representation of a 3D line, which embeds an Z-axis-alignment rotator. Problems involving lines in 3D (such as the intersection of two planes) are vastly simplified, as they now can be solved in the rotated space where the line stands vertical. This representation offers a major simplification over the traditional linear algebra representation, based on $\mathbf{p} = \mathbf{p_0} + t\ (\mathbf{p_1} - \mathbf{p_0})$.

Most importantly, the *degree-of-difficulty* of problems that may be solved is elevated beyond what most would think possible for high school students. An example is solving for the 2 intersection points of 3 overlapping spheres, the basis for GPS positioning. A more sophisticated example is interstellar navigation, in which the spacecraft can assume any arbitrary 3D attitude. Problems of this type are notoriously daunting using angles. Using a rotator instead to represent craft attitude simplifies things, and prepares the student to learn how to infer the craft's attitude by observing two remote galaxies, a technique called rotational inference. Once attitude is determined, the craft's location may be solved by sighting 3 nearby stars, and applying directional triangulation (the intersection of 3 planes). Judging by the aerospace literature, star-tracking navigation is a PhD-level problem. However, near the end of their first algorithmic geometry course, a high-school student will have amassed the know-how and software library to solve interstellar navigation geometry in 3 mental steps, and 11 additional lines of code.

While the problems solved are very advanced for high school, in time we'd like to prove that algorithmic geometry is no more difficult to teach and learn than traditional geometry. The concepts and methodology are different. Working intimately with one's personal computer by committing each problem solved to an automated algorithm seems to change the *mental ergonomics* of geometry in a fundamental way. Ordinary mental effort is applied in bite-sized chunks, and the human brain is tasked with short bursts of incremental creativity, its key strength. By layering solutions, the human is able to rapidly ascend a learning curve, leading to geometric power and sophistication in under 100 hours of formal theory and lab exercises.

The algorithmic approach overcomes a key difficulty inherent in traditional analysis. An example can be seen in the *intersection points of two circles* problem. A pure analysis using 10 scalar variables [x1 y1 r1] [x2 y2 r2] → [ix1 iy1] [ix2 iy2] bogs down quickly as the circle constraint entails squaring the difference of variables:

$$(ix1 - x1)^2 + (iy1 - y1)^2 = r1^2$$

There is an explosion of terms. When a square root must be applied to a sum or difference, analysis has little to offer...further simplification grinds to a halt.

Software geometry avoids this headache by virtue of the fact that the variables being processed are *not* unknowns. Though they are treated abstractly as unknowns by the programmer, when the computer takes over and runs the algo, the inputs processed are always hard numbers. Arithmetic operations *crunch* the numbers, yielding other hard numbers. When the software comes up to a square root operation, the operand expression has already been crunched down. Processing consolidates information. That's on the computer side – on the human side, we still have the freedom to think through a problem using abstract unknowns...points, directions, distances, and higher-level objects. In the partnership, human and computer play to their respective strengths, and sidestep their limitations. Appendix B illustrates a classically daunting problem, the intersection of a sphere and 3D circle, and how the algorithmic approach reduces it to 4 inductive mental steps, and 17 new lines of Java code.

A few words should be said about the difference between geometry and algebra. While there is some overlap superficially to the content presented in a linear algebra class, the study of geometry is quite distinct in that the subject matter is about *space*. It benefits from a person's lifelong spatial experience, and the visual imagination processes that come with it. For instance, when confronting the problem of how two planes intersect to form a 3D line, there is no need for formal axiomization or proof of this phenomenon...we rely on the student's intuition. Visual intelligence makes geometry less abstract than algebra. This is one of the arguments why Algorithmic Geometry is likely better matched to high school juniors and seniors, compared with AP Linear Algebra[4].

Unlike linear algebra, *all* the quantities we use in algorithmic geometry have spatial meanings that one can point to in a sketch. Let's compare algebraic and geometric representations of a plane:

$$ax + by + cz - d = 0 \qquad \text{(algebraic plane equation)}$$

$$\mathbf{p} \bullet \mathbf{o} == \ell \qquad \text{(algorithmic geometry plane predicate)}$$

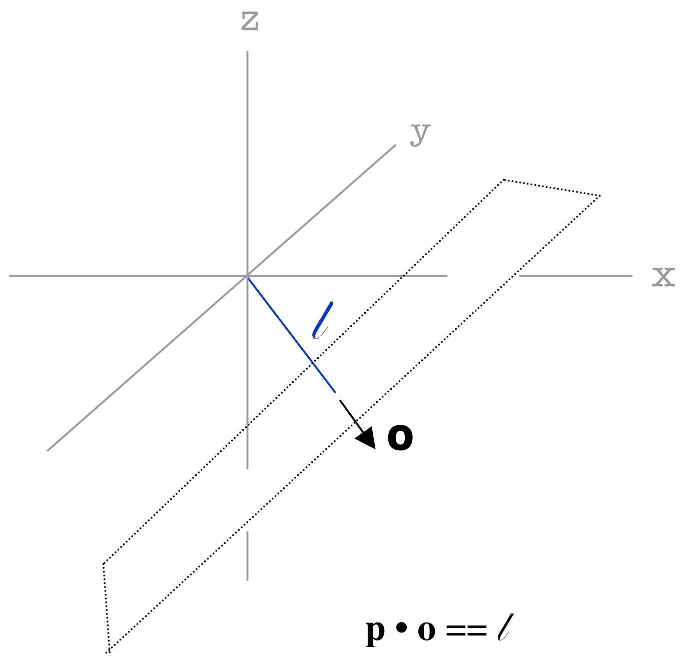
Syntactically, after expanding the dot product, these two formulas look almost identical. However, what does "a" stand for in the top equation? It doesn't have any *meaning* other than as the coefficient of x.

In the geometric formula, all the individual quantities have spatial meanings:

o is the plane's orientation, the direction pointing perpendicular to the plane (its "normal" direction).

ℓ is its ± signed distance from the origin, + defined by arrow **o**.

p is any point in 3D space. If **p** is in the plane, the comparison for equality will evaluate `true`. For all other points, it will be `false`.



Through the medium of sketching, these geometric concepts benefit from the student's native visual intelligence in problem solving. For example, imagine what will be the effect of a 3D coordinate rotation on a plane? With the right visualization graphics (or handheld manipulatives), it is easy to become convinced that rotation has no effect on the value of ℓ. The plane's distance from the origin is *preserved* during coordinate rotation.

It is also obvious that the plane's orientation **o** must change for a new choice of axes. Since 3D directions like **o** are conceived as points on the unit sphere, and we know how to rotate any point **p** for a new set of axes R, then it is a simple matter to transform the plane's orientation **o**: we'll rotate it just as if it were a point:

$$\mathbf{o'} = \text{rotate}(R, \mathbf{o}) \qquad R = [\ \mathbf{newXaxis}\ \ \mathbf{newYaxis}\ \ \mathbf{newZaxis}\ ]$$

Combining these mental steps, the student is now poised to write a simple algorithm that automatically transforms any plane object PL → PL' for desired coordinate rotation R:

```
public static Plane rotate (Rotator R, Plane PL) {
    return new Plane (Vec3.rotate(R, PL.o),  PL.l);
}
```

Compared to algebra, advanced geometry is much more solidly grounded in real world experience and intuition, and might therefore be expected to "click" with a wider group of high school students than the more abstract and rule-based linear algebra. The feedback from our initial students, both now in engineering college, is that algorithmic geometry gave them a decided leg up understanding the linear algebra and engineering vector math they encountered. We think of math-geometry education, like Papert[5] and Piaget before him, as a process of hanging formal education onto a person's native geometry intelligence, something that will serve a lifetime in situations requiring spatial problem-solving.

The algorithmization of one's math know-how is a truly epochal, revolutionary change. Under this new methodology, you solve problems in bite-sized chunks, doing the creative, symbolic work with help from paper and pencil, and committing each incremental solution to an automated algorithm. Assuming the creative solution and its edifice in software can handle all possible cases of input (i.e., you design general solutions), then algorithms may be reused in any new problem context without ever revisiting the details. You are afforded the luxury of forgetting *how* you solved the problem while applying it to solve harder problems. This is a remarkably effective way to allocate all-too-scarce human attentional resources. It allows us to solve wicked difficult problems with ordinary mental effort, the complexity conveniently "hidden" in the mental work previously expended and now compartmentalized in software. You have to experience this mental accelerant firsthand to get an appreciation for it. Once you taste the power, you will find the sheer economy irresistible.

Problems are only focused on intently for a short time. The typical problem takes about 20-30 minutes total, including problem-definition, solution sketch, pseudo code, Java coding, and algorithm testing. Over a 145-hour course, ~200 algorithms are mastered. As each problem is solved, it can be walked away from confident that it has been solved for good. Appendix B illustrates a complete walkthrough of an advanced 3D problem.

The approach taken in algorithmic geometry differs significantly from first-generation computer-assisted math, e.g. Geometers Sketchpad[6], Cabri, MATLAB and Maple. In first generation tools, the goal was to take math theory content predating the advent of computers, and interactively bring it to life. This pattern of infusing old content into a new medium is exactly as McLuhan[7] described in the 1960s. In 2nd generation cybermath, the *theory content* itself begins to morph under evolutionary pressure wrought by 50 years doing math in the new medium (computing). The theory foundation that emerges is *post-computational*, meaning basic concepts are chosen to suit a *human-computer partnership* solving geometry problems – the basic constructs desired are ones good for algorithm-writing. 2nd generation cybermath invites a significant rethinking and refactoring of math education content, asking the question, how would the basic ideas of a branch of mathematics change, had the subject waited to be invented until the age of computers?

Such an intellectually imaginative approach is the one needed to get our students moving upward in the international sweepstakes that will determine standard-of-living a decade or two from now. The Hart-Rudman Commission Reports[8] laid out the argument that math-science education is a predictor of a nation's future economic success. Their visionary document got me started on my quest to modernize 9-12 geometry education.

Why is it appearing now?

Algorithmic geometry is emerging at this time because a growing cadre of professionals cross-trained in math and computer science have discovered the immense power and synergy at their fingertips as they codify their math knowledge into an expanding library of solutions to problems. Vector math is a natural fit to computation, as it allows the problem thinker to delegate all the scalar arithmetic drudgery. This frees up human attention to be focused on higher-level objects and relationships. Since previously solved problems never have to be revisited, and provide fodder for solving yet harder problems, the human side of the partnership can ascend a rapid learning curve.

Direction vectors originated in mathematical physics[3], and have gradually worked their way into scientific software over the past 30 years.

The shifts in geometric theory are motivated by the need to simplify and restandardize the topic for ease in thinking and writing algorithms. The translation of problem-solving thought → sketches → pseudocode → runnable code has been designed for minimal friction and maximum compatibility. The pieces are made to fit together like a Lego set. Clearly, traditional geometry had no such requirement in mind. The first 50 years of computer geometry concentrated on translating pre-computational geometry ideas literally into the new medium. Trig library functions like arccos that programmers find annoyingly ill-designed for computation are an example of this dutiful allegiance (arccos has two possible angle solutions).

The next 50 years will see a new generation of geometry ideas flourish whose foundational concepts fully anticipate and integrate the medium of computation, as if geometry had waited to be invented in our post-computational world. The new geometry comes to grips with the key representational issues in *delegating geometry work to computers*. The observation that angle *overcompresses* directional information at a cost of algorithmic complexity exemplifies 21st century math-informatics thinking.

High quality software programming tools suitable for introducing math students to software geometry, available for free over the internet (Eclipse/Java), are a recent phenomenon in the years since the dot-com explosion. A high school course for math students involving programming would not have been practical or affordable *en masse* before the post-2000 generation of software tools came on the scene.

Fitting Algorithmic Geometry into the Curriculum

Where does a topic like this fit into the curriculum? This is a complicated question, but a good starting point is recognizing that there is a sub-group of high school students who are aggressive math learners. Having taken accelerated math since middle school, by junior or senior year, these students have exhausted all the AP offerings: calculus and statistics. This suggests a niche for a capstone course in advanced geometry.

Unlike calculus and statistics, there is no standard freshman-level geometry course to be repackaged as an AP course. At the collegiate level, geometry content (other than remedial) is highly specialized by math-science-engineering discipline. For instance, a Physics major will learn the vector calculus of gradients, divergence and curl as a warm up to electro-magnetism. A math major immerses in matrix math via Linear Algebra, using MATLAB. A computer science student majoring in computer graphics will be introduced to 4D homogeneous coordinates implemented in C++. An engineering student specializing in robotics will learn yet another approach based on Jacobian matrix inversion. Computational Geometry is offered in some computer science departments as preparation for Geographic Information Systems. Computational biochemistry is still developing its own unique geometry based on tilings. All of these specialties have, as a common thread, vector geometry using software as the *medium* of expression.

Algorithmic Geometry is an attempt to *simplify and standardize* a vector geometry foundation -- fully integrated with the software medium -- and applicable to a wide swath of STEM pursuits. Developing hands-on competence tackling wicked-difficult problems is intended to seduce young people with their power as problem-solvers.

Why should it be introduced in high school? The rationale is simple – the recruitment of students into the STEM pipeline has to match up with the student's decision timeframe – grade 11-12 being the critical time when college and major are being decided. Algorithmic Geometry is equal parts *STEM marketing* and state-of-the-art geometry education. Attentive to social bonding as a parameter, we are adopting a "team teaching" approach, where a credentialed 9-12 math teacher will go through the training workshop alongside his/her handpicked student TA (bringing Java experience). The objective is a more supportive teaching environment, and for the learners, a peer role model already comfortable with algorithmic geometry.

NCTM's *Principles and Standards for School Mathematics: Standards for Grades 9-12*[9] calls out *representation* as the deepest level of math understanding, and champions the opportunity at every grade level to apply math using the computer, and to connect to real world applications. The question of how best to represent geometric properties and objects, bridging the media of brain, pencil and paper, software programming, computer graphics and science applications is the paramount theme of algorithmic geometry.

What level of math understanding will enrolling students need to be successful? This is one of the research questions we still have ahead of us. It seems reasonable that the readiness criteria will be similar to electing AP calc or stat. Students should be well-versed in geometry, algebra, trigonometry, functions, and the basics of vectors and matrices. Scientific notation and familiarity with very small numbers will be helpful in understanding how a digital computer represents real numbers. A faculty for spatial visualization is fairly indispensable, for instance, the ability to rotate a 3D object mentally.

Ability to draw passable-quality sketches is a necessity. Ability to think through proofs, and/or compose complex creative output (music, poetry, writing, art, drama, sewing) are other readiness variables we'll be looking at. In terms of social orientation, ability to work solo deeply-engaged in a problem for up to 30 minutes is a success factor. For our earliest students, we're looking for some leadership and early-adopter comfort, given the additional role these students will be offered as paid TAs.

What about computer science / programming experience? This is an important research question, both in terms of student readiness, and teacher preparedness. Based on the long, storied history of computers in learning, we can almost predict that schools without an established computer science curriculum and staff will not be ready to participate as pilot sites. Some local Java talent must be cultivated from which to recruit TAs, not to mention managing the laptop-desktop-software infrastructure on a day-to-day basis. We are realistic about piloting in schools where CS is established.

Given that, the course has been designed so that no previous software programming is required. Only a small subset of Java is used, and the naturalness of Java mathematical expressions makes the language intuitive. The labs are constructed so that Java is learned piecemeal by example and mimicry. In the pilot, one of the two students had no previous programming, and by the end of week #2, had picked up the essential language skills. This is *not* a programming course – the focus is on geometry representations and problem-solving using sketching. Software programming is viewed as a *medium* in which applied math workers nowadays store and exercise their accumulated know-how. The power of automating one's brainpower becomes self-reinforcing, starting from the first lab. We would not attempt to teach a pilot course without a TA having Java experience working in support of the primary instructor, where both have been through the course and received training in managing the course logistics. A tech support person also wants to be on-call nearby to resolve non-courseware glitches.

The Computer Science Teachers Association (CSTA) in conjunction with ACM has developed a visionary, comprehensive roadmap, *A Model Curriculum for K-12 Computer Science*[10]. The US is lagging behind Israel, Scotland, France, Germany, Finland, South Africa and Ontario (Canada), places where computer science (or *informatics*) has become *required curriculum*[11]. Not to be confused with computer literacy, computer science education prepares students to integrate all that computing has to offer in whatever pursuit they embark on, with awareness of what is technically possible and the ethical issues that permeate such powerful technology. And, it offers a pre-professional track for those wanting to pursue careers in software development and IT. The goal is a well-educated citizenry able to effectively govern technological society for the common good, and economically, able to maintain its position in the global high-tech arena.

In terms of cognitive skills, the objective is to steep every child in *algorithmic problem-solving*. This means the ability to confront a complex problem in its entirety, systematically break it down into its constituent parts (including side effects), and then generate *rigorous* solutions in terms of a set of tangible steps that move from the starting point to the solution. The visionary genius behind this plan is that *systems analysis and design* will be taught as universal skills, the same way reading and writing expanded in the 20th century. Presumably, it will confer the same egalitarian benefit.

The Model Curriculum defines four levels of pedagogy spanning k-12:

Level I:	Foundations of Computer Science (k-8)
Level II:	Computer Science in the Modern World (grade 9 or 10)
Level III:	Computer Science as Analysis and Design (grade 10 or 11)
Level IV:	Topics in Computer Science (grade 11 or 12)

Algorithmic Geometry fits neatly into Level IV, where courseware and projects are intended to delve further into CS, explore an interdisciplinary topic, and offer project-based learning opportunities. Looking to a future time when all US schools offer CS education beginning in k-8, Algorithmic Geometry could become a mainstream Math-CS capstone elective able to be offered in most high schools.

Speaking of electives, Algorithmic Geometry students will expect to receive college credits just as they would for taking an AP course. A crucial enrollment hurdle innovative courses like this face is obtaining accreditation. University of California has a review process that takes a year to complete, and requires a school district as the applicant. Getting accredited in all 50 states seems daunting.

But a journey of a thousand miles begins with one step. That first step is building a small coalition of math-CS educators eager to cultivate next-generation 9-12 geometry education on their home turf. A model worth emulating is the post-Sputnik BSCS Biology curriculum modernization[12], in which a small group of dedicated, tenacious, NSF-funded educators redefined what was worth teaching and learning in their field, over the course of a decade of summer workshops. The impact of those heady days is still reverberating decades later in the phenomenal progress spawned in medicine, marine science, agriculture, and environmental science.

Seizing the Competitive Opportunity

We owe it to our young people to prepare them for the future, and equip them with the thinking tools needed to thrive in it. Geometry teaches spatial problem-solving, a key underpinning of our way of life. The revolution brought about by ubiquitous access to computers for solving spatial problems has given us vivid 3D animation, robots that can climb stairways, the ability to pinpoint our location anywhere on the globe, and be transported (virtually on-screen) anywhere else. The lessons learned in the trenches of geometric software development are about to yield an unexpected dividend – refashioning the core constructs of formal geometry education, optimized for semi-automated problem-solving. The spinoff from the software industry back into education: *cybergeometry*.

Though few people understand the significance of this new paradigm yet, what lies ahead potentially is a *quantum leap* in the problem-solving sophistication of students leaving high school. At the heart of this new approach is streamlined theory for working in 3D, coupled with a personal computer ready to automate everything downstream from the front-end creative process. The "uphill" cost incurred is learning how to delegate work to the machine – expressing your clever problem solutions in the medium of software.

For young people, gaining competence in this medium shapes their world view in the direction of mastery over technology, counterbalancing the passive, consumerist relationship they are thrust into by "smart" products -- iPods, DVDs, Wii. A course like Algorithmic Geometry peels back the skin, and demystifies the innards of 3D graphics, robotic motor coordination, GPS positioning, and machine vision. Students acquire a feel for occupying a position of *dominion* over their silicon servants.

Knowledge is power - the future belongs to those who can learn it, shape it, wield it, and transmit it. The opportunity to reinvent geometry education for the 21st century is here – we just need to seize it.

(The author may be contacted at pierre@AlgoGeom.org).

About the author

Pierre Bierre earned a BS in Theoretical Physics from Stony Brook University, and a MS in Computer Science from Univ. of Colorado Boulder. He served 4 years at Stanford Univ. Neuropsychology Lab as research software engineer, studying feline vision and learning theory. In the field of AI, he developed a theoretical framework for experiential sensorimotor learning, transmission of knowledge, and the totality of knowledge.

He worked as algorithm scientist for Becton Dickinson Biosciences, inventing their *Attractors* system of automated immune cell analysis. He founded a robotics group at BDB, and developed their first homegrown blood cell sample prepper, the LyseWash Assistant. As an independent entrepreneur, he invented and prototyped the BuildExact system for CAD-guided-laser contruction layout. As a spinoff of that effort, he invented algorithmic geometry, authored a coursebook for it, taught a pilot course, and is seeking grant support. He is a member of National Council of Teachers of Mathematics and Computer Science Teachers Association. He lives in the San Francisco Bay Area.

Further reading:

(1) Position and Attitude Sensing by Directional Triangulation, P. Bierre, Institute of Navigation, 2006 National Technical Meeting, Emerging Technologies Panel, Monterey CA.

(2) *Flexing the Power of Algorithmic Geometry*, Pierre Bierre, Spatial Thoughtware, 2008, 182 pages, 114 illustrations

(3) Jeffreys, H. and Jeffreys, B. S. "Direction Vectors." §2.034 in *Methods of Mathematical Physics, 3rd ed.* Cambridge, England: Cambridge University Press, p. 64, 1988.

(4) NSF 0733015, Cuoco, Albert, Linear Algebra and Geometry: Advanced Mathematics for More Students

(5) *Mindstorms – Children, Computers and Powerful Ideas*, by Seymour Papert, Basic Books, 1980

(6) Lifting the Curtain: The Evolution of The Geometer's Sketchpad, Daniel Scher, The Mathematics Educator, V10 No 1 Winter 2000

(7) *Understanding Media: The Extensions of Man*, Marshall McLuhan, Gingko Press 1964

(8) U.S. Commission on National Security in the 21st Century, Seeking a National Strategy (Hart-Rudman Commission Phase II report)

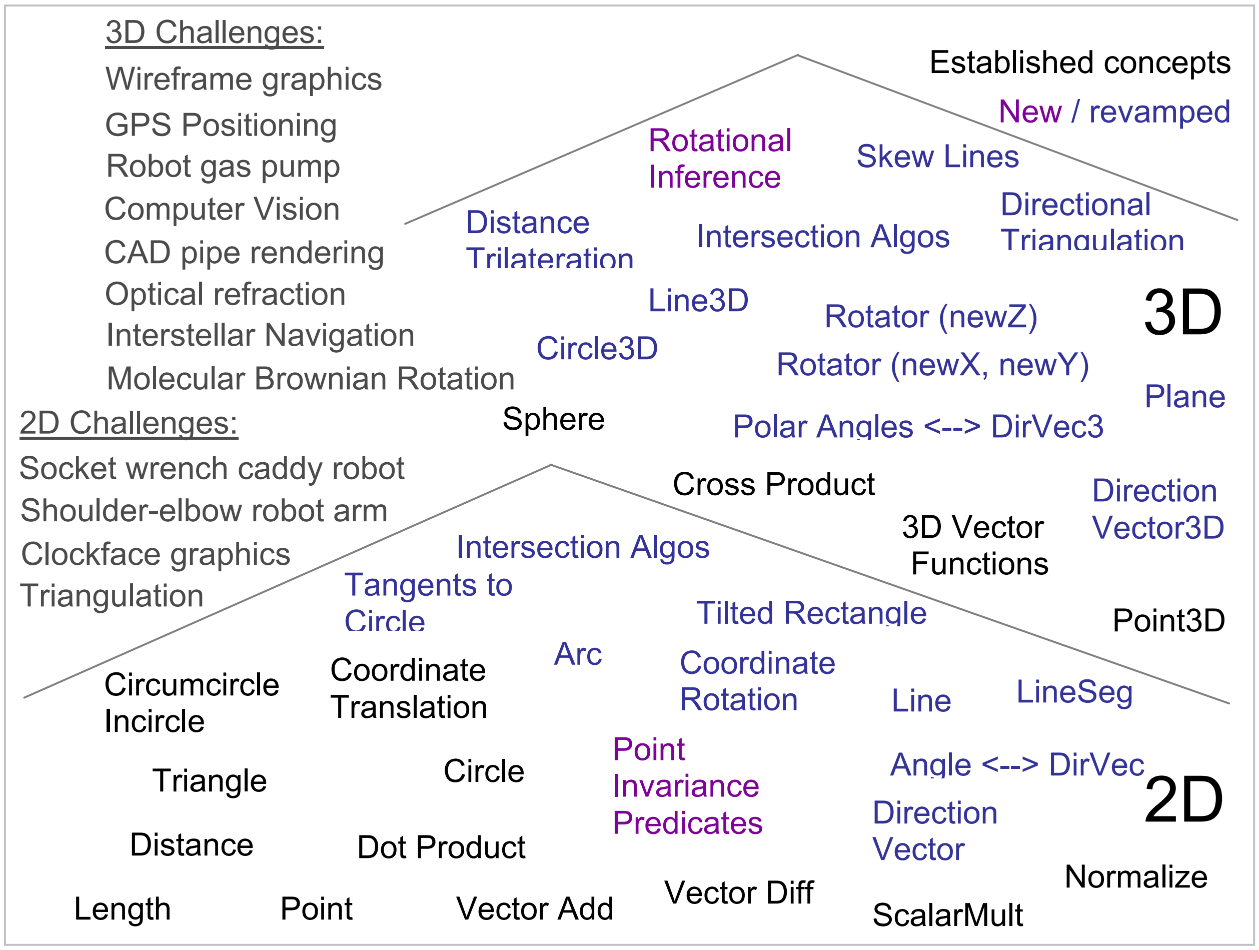
(9) Principles and Standards for School Mathematics: Standards for Grades 9-12, National Council of Teachers of Mathematics (order online: www.nctm.org)

(10) A Model Curriculum for K-12 Computer Science, Computer Science Teachers Association (download from: www.csta.acm.org)

(11) The New Educational Imperative: Improving High School Computer Science Education (Using worldwide research and professional experience to improve U.S. schools), CSTA Curriculum Improvement Task Force, Feb. 2005. ACM ISBN: # 1-59593-335-2 (download from www.csta.acm.com)

(12) *The BSCS Story: A History of the Biological Sciences Curriculum Study,* Laura Engleman, Editor, 2001. (Order from www.bscs.org)

Appendix A Algorithmic Geometry Course Syllabus



The syllabus progresses from bottom to top, starting with automation of 2D vector functions. Midway, the focus transitions from 2D to 3D. Topics in **black** involve automation of established theory concepts. Topics in **blue** represent areas where theory has been substantially revamped, largely the consequence of sidestepping angle, and restandardizing on direction vector as the natural way to represent spatial direction. Students explore 5 Project Challenges in **grey**.

Mathematicians will recognize 90% of the content as familiar. New innovations in purple merit more explanation than there is room for here:

- When deciding how to represent a geometric object, students are coached to seek out the object's *point invariance predicate* – a formula you can plug arbitrary point **p** into, computing a decision whether or not **p** is included in the object.

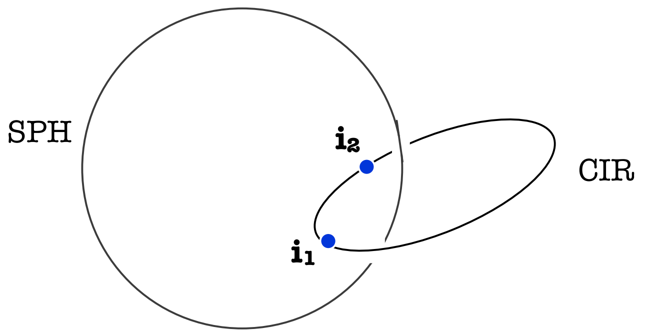
- Rotational inference is a new algorithm allowing 3D rotational attitude to be inferred from a pair of directional observations, a new mathematical 3D gyro.

Appendix B: Walkthrough of Problem-Solving Methodology (5 steps)

This problem is tackled about 3/4 of the way through the course.

1

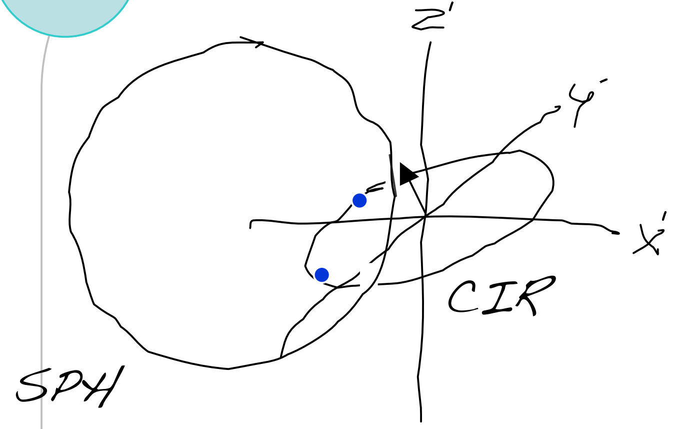
Problem: solve intersection of sphere and circle3D



Given	Compute Results
SPH = [**c** r]	numIntersectionPoints (0, **1**, **2**, ∞)
CIR = [**c** **orient** r]	point locations $\mathbf{i}_1$ $\mathbf{i}_2$

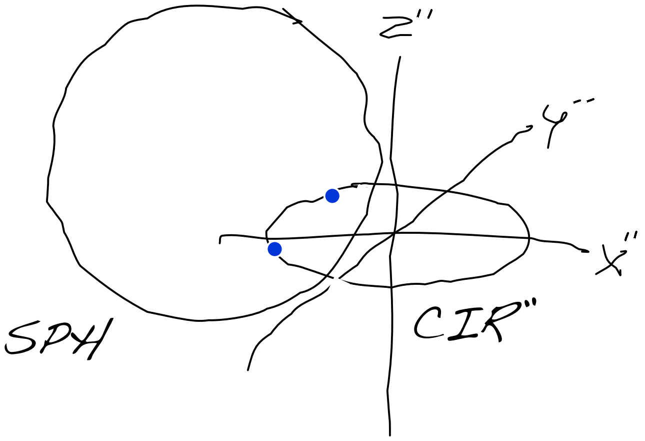
orient is the orientation of CIR (orientation of the plane it lies in)

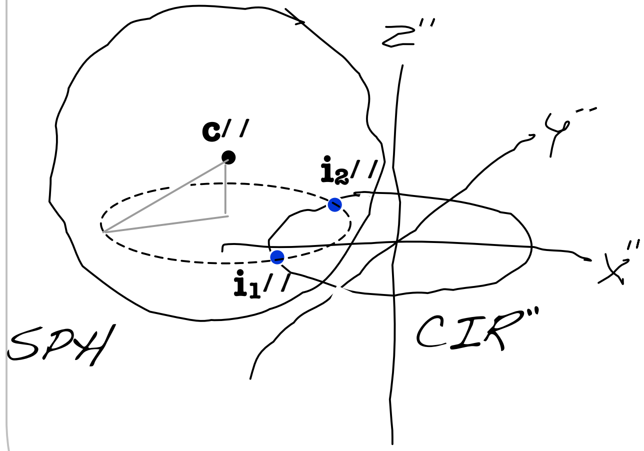
2 Sketch: mental steps that solve problem



shift into
CIR's local
coords

rotate coords
leveling CIR''





visualize 2nd
circle where
x''-y'' plane
cuts thru
SPH''

<u>Solve as intersection of 2 circles</u>!!!
(already have algo in my 2D library)

The student's sketch outlines a high-level, graphical specification for an algorithm.

3

Write *pseudocode*: outline of computational steps

```
int intersectionOf(SPH, CIR, /*returns*/ i1, i2 ) {
  // step 1: translate SPH into CIR’s local coords
  SPH’ = Translate (CIR.c, SPH)

  // step 2: rotate coords adopting CIR.orient as newZaxis
  Rotator R = RotatorForNewZAt(CIR.orient)
  SPH” = Rotate (R, SPH’)

  // step 3: specs for 2 circles (use Pythag to get missing radius)
  cir1 :  c = [ 0  0 ]                      r = CIR.r
  cir2 :  c = [ SPH”.c.x  SPH”.c.y]     r = sqrt ($SPH.r^2$ - $SPH”.c.z^2$)

  // step 4:compute intersection of cir1, cir2 (call from my 2D library)
  numPts = IntersectionOf(cir1, cir2,  /*returns*/  i1”, i2”)

  // step 5: down-transform i1”  i2”  back into original coords
  i1 = Untranslate(CIR.c, Unrotate(R, i1”))
  i2 = Untranslate(CIR.c, Unrotate(R, i2”))
  return numPts;
```

Translating the sketch into pseudocode requires understanding where and how the sketch implies *computation* of objects.   Note: Steps 1,2,4,5 invoke past work.

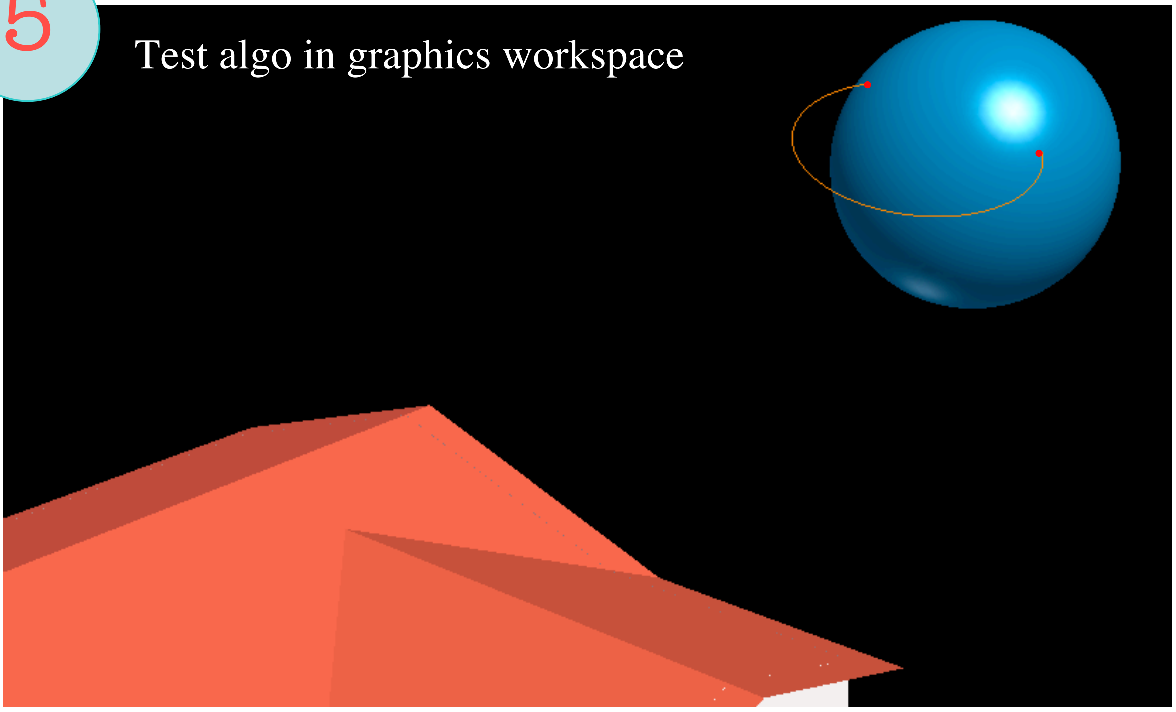
4

Put into Java source code using Eclipse editor

```java
public static int IntersectionOf (Circle3D Cir, Sphere Sph,
                                       /*returns*/ Vec3 IntPt1, Vec3 IntPt2)  {

//Step1,2. Translate to Cir-local coords; rotate coords so circle lies in X”-Y” plane
Sphere Sph_p  = Sphere.Translate(Cir.c, Sph);
Rotator R = Rotator.RotatorForNewZAt(Cir.o);
Sphere Sph_pp = Sphere.Rotate(R, Sph_p);

//Step3. Find the intersection of Sph_pp with the X”-Y” plane (circle2_pp)
if (Math.abs(Sph_pp.c.z) > Sph_pp.r) return 0; // sphere too distant from x-y plane
Circle circle2_pp = new Circle(new Vec2(Sph_pp.c.x,Sph_pp.c.y),0);
circle2_pp.r = Math.sqrt(Sph.r*Sph.r - Sph_pp.c.z*Sph_pp.c.z); // pythagoras rules!

//Step4. Find the intersection of 2 circles in the X-Y plane (2 pts.)
Circle circle1_pp = new Circle(new Vec2(0,0), Cir.r);
Vec2 IntPt1_pp = new Vec2(); Vec2 IntPt2_pp = new Vec2();
int numPtsFound = Circle.IntersectionPtsOf (circle1_pp , circle2_pp ,
                          /*returns*/ IntPt1_pp, IntPt2_pp);
if ((numPtsFound == 0) || (numPtsFound == 3)) return numPtsFound;

//Step5. Go back to 3D, unrotate and untranslate back into the problem space
Vec3 I1_pp = new Vec3(IntPt1_pp.x, IntPt1_pp.y, 0);
Vec3 I2_pp = new Vec3(IntPt2_pp.x, IntPt2_pp.y, 0);
Vec3 I1  = Vec3.Untranslate(Cir.c, Vec3.Unrotate(R, I1_pp));
Vec3 I2  = Vec3.Untranslate(Cir.c, Vec3.Unrotate(R, I2_pp));

//copy out
IntPt1.copyFrom(I1);     IntPt2.copyFrom(I2);
return numPtsFound;
}
```

5

Test algo in graphics workspace



Solution step	**Typical time expended (min.)**
1 Problem definition	5
2 Sketch out a solution	15
3 Pseudocode algo	10
4 Java implementation	20
5 Graphical algo testing	15
Total problem-solving time	65 min.

The intersection of sphere + circle3D is now solved for good, and may be called upon to solve higher-level challenges. Examples where this algo is useful:

- intersection points of 3 overlapping spheres (GPS positioning)
- 3D CAD pipe rendering with moving viewer (drawing cylindrical outlines)