The *Predicate-form* 3D Vector Line Equation

p • ZAlignRotator == XYLocation

A Modern Perspective

The *age of computing* brings forth new criteria for 3D Geometry -- namely, numerical representations easy-to-use for devising software number-crunching algorithms. These representations must also be eminantly human-friendly, meaning, easy to teach and learn, visualize, sketch, and display. No longer important are features that support pencil & paper mental arithmetic. That relaxation invites representations specifically designed for doing geometry as a *human-computer partnership*.

The ideas presented here extend the thinking in The 2D Vector Line Equation. We begin with the mathematics for representing 2D line **L**. Its uniqueness is captured in two numerical features: orientation **o** and location ℓ. The effect of coordinate rotation **L** → **L'** is quite illuminating (Fig. 1).

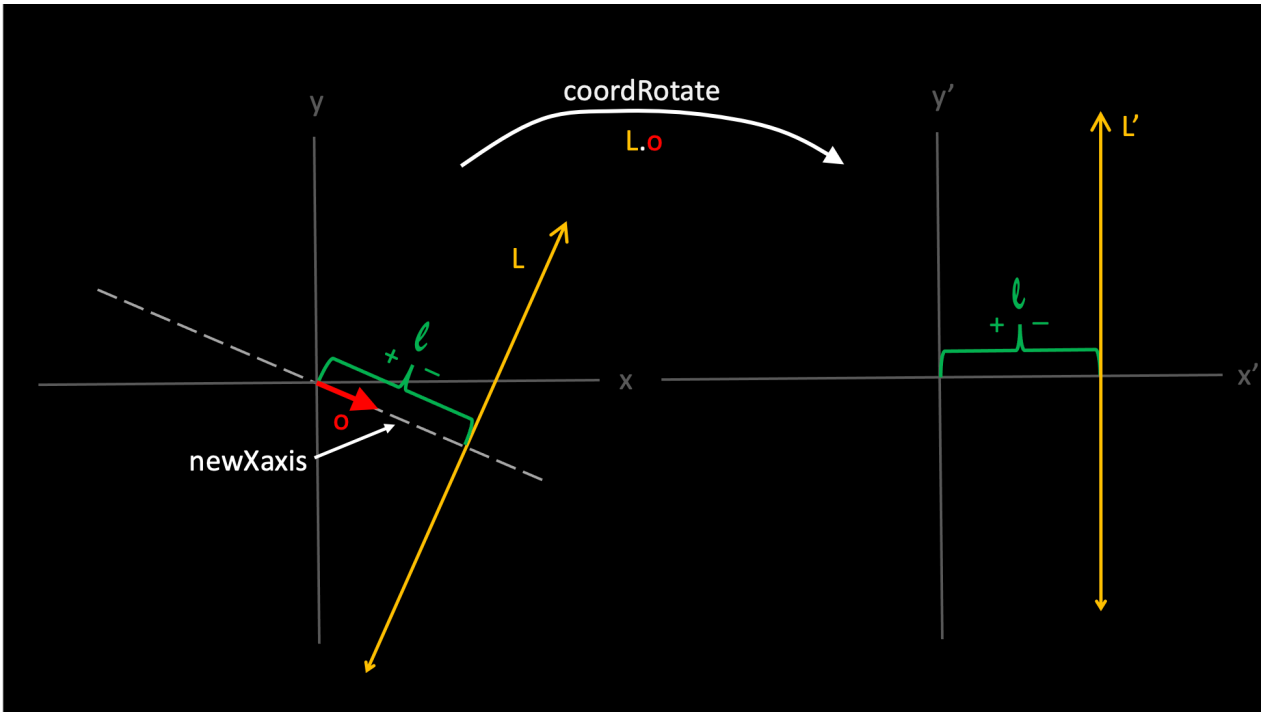

Fig. 1. All 2D Lines stand vertical in rotated coordinates controlled by line L's orientation **L.o**

2D coordinate rotation may be applied mathematically (and computationally) to points and lines. The vector-math way to say how the X-Y axes want to be rotated is by giving **newXaxis**, a 2D direction vector specifying where to place this alternative x^{+}*-axis*. When we rotate the X-Y axes so that **newXaxis** takes on **L**'s orientation **L.o**, what happens to **L'** ? As shown in Fig. 1, the transformed line **L'** stands perfectly vertical, with run direction pointing upward. All points on **L'** are now neatly stacked to share an *invariant x' coordinate*, the value of line **L**'s location feature ℓ. Would it be possible to do something analogous to represent an extended line in 3D?....yes! (Fig. 2).

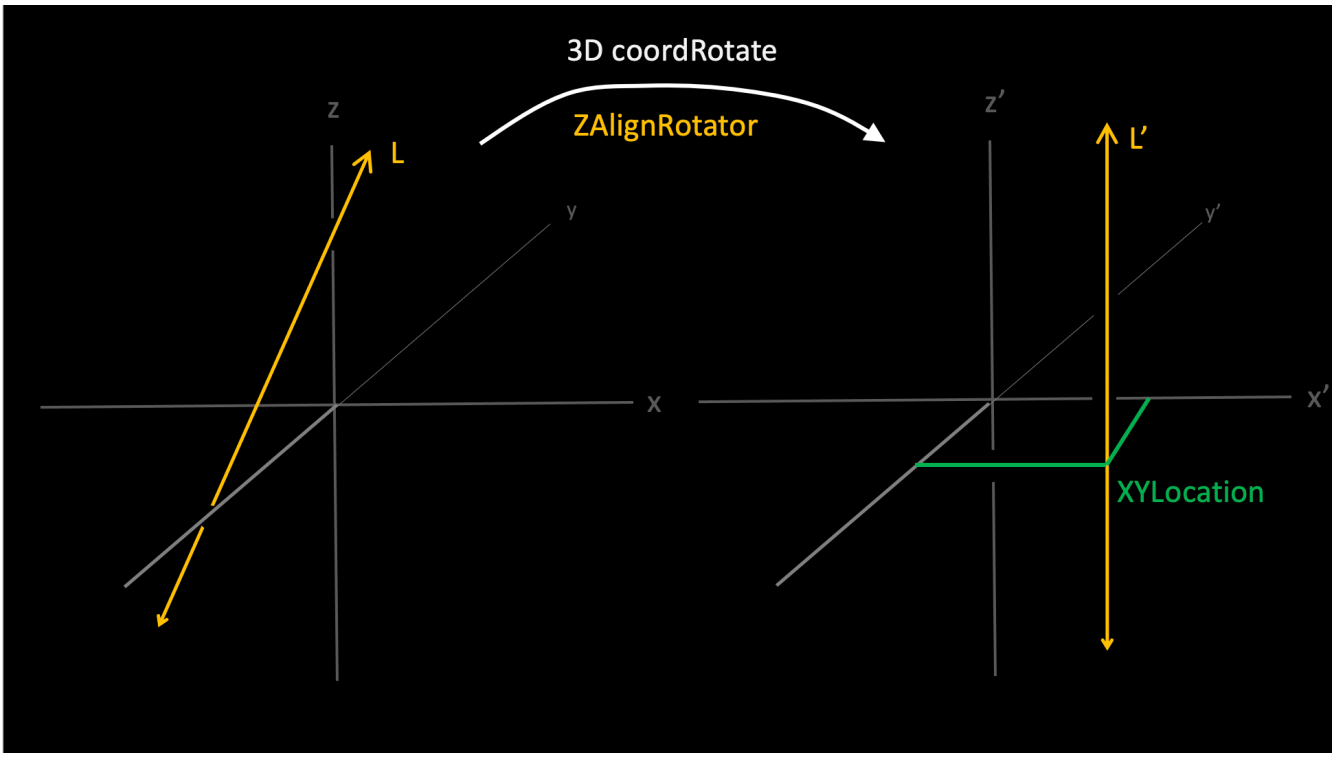

Fig 2. All 3D Lines can similarly be coordinate-rotated by forming and applying ZAlignRotator

ZAlignRotator captures the run direction of line L, while XYLocation pins down its spatial location (albeit in rotated coordinates).

To learn this advanced 3D numerical geometry, students must first acquire the prerequisite concepts.

Direction Vectors, Alternate Axes and Coordinate Rotations

These 3 thinking tools are first mastered in 2D. The jump into 3D is smoothed -- representing 3D spatial directions with 3D direction vectors (**d** = point on the unit sphere [x y z]). Sometime after this experience consolidates, students are ready to be introduced to alternate, rotated axes sets called "Rotators" (Fig. 3a)

$$R_{newAxes} = [\ \mathbf{newXaxis}\quad \mathbf{newYaxis}\quad \mathbf{newZaxis}\ ]$$

a bundle of three 3D direction vectors numerically specifying alternative axes (Fig 3a). They must be mutually orthogonal (form a cube corner) and obey the right-hand rule:

$$\mathbf{newXaxis} \times \mathbf{newYaxis} = \mathbf{newZaxis}$$

Students undertake practical applications in an interactive 3D workspace to cement the concept and utility of Rotators. In the DataflowGeometry3D course, a sequence of 3 *Applications Projects* works to acquaint students at an introductory level:

Mutual Perpendicular Finder → Flight Path Solver → Hypertube Drilling

Hypertube Drilling Project -- a gravity tunnel from San Francisco to Seoul

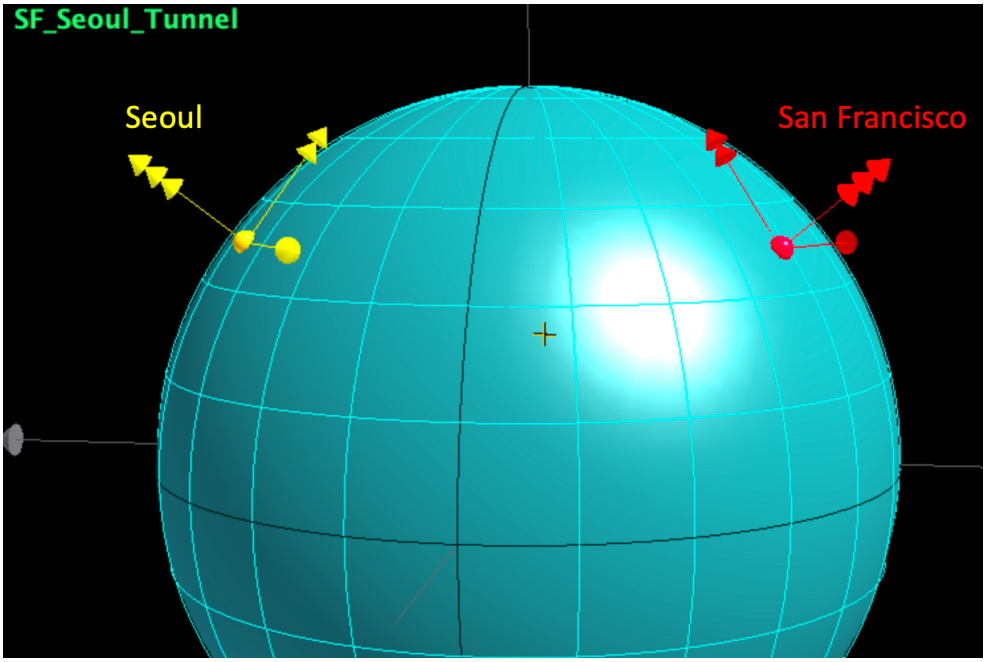

Fig. 3a. 3D graphics of Earth superimposing "Rotators" (local 3D axes) onto S.F. and Seoul

This project introduces the student to a practical application of 3D coordinate rotation. The drill teams in SF and Seoul need their bore-holes specified as *local* compass-azimuth + elevation angles. This can be achieved by rotating the global tunnel direction into the local axes systems shown in yellow and red.

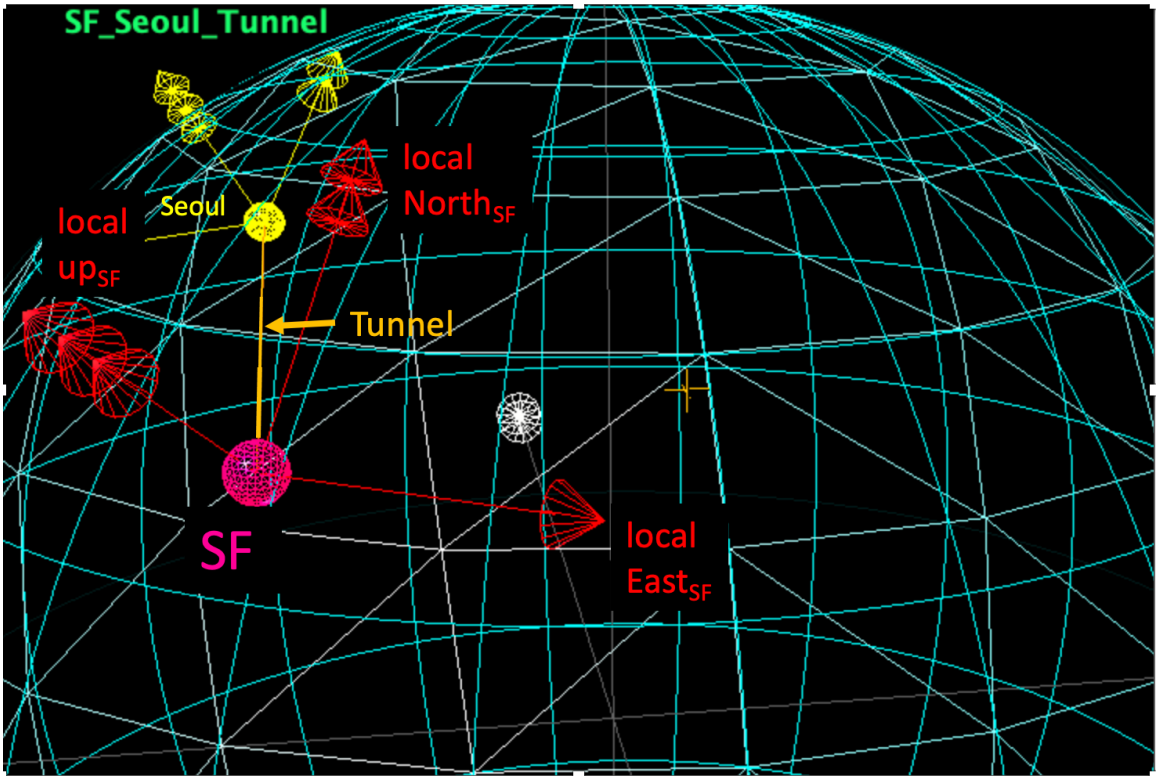

Fig. 3b. The SF tunneling direction (orange) must be converted into local SF coordinates (red axes)

Students emerge from this project able to *think and compute* with these alternate-axes Rotators.

Predicate-form 3D Vector Line Equation

What do all the points on a 3D extended line have in common mathematically?

This question may be answered in different ways. Our choice -- one that is optimized for a partnership of human + computer is the following (illustrated in Fig. 2);

$$\mathbf{p} \bullet \mathbf{ZAlignRotator} == [\ XYLocation_x \quad XYLocation_y \quad \text{------}\ ]$$

Here we're using vector times matrix notation ("•") to indicate transforming point **p** by coordinate rotation **ZAlignRotator**. This operator has the effect of stacking all the line's points vertically so that they share invariant x' and y' coordinates, i.e., all coordinate-variation along line L' has been *rotated* into the z' axis.

We call this a *predicate form* of the line equation. Why? Notice the "double equals" (==) sign bridging the two sides of the equation. In computer science, this notatation literally means

"evaluate the left side, then evaluate the right side, then compare for equality"

The computational result is going to be TRUE or FALSE. Given any 3D point **p** = [x y z], and any 3D line **L** represented numerically as feature pair **L** = [**ZAlignRotator** **XYLocation**], this formula gives us the means to *directly test if the point is on that line*. In other words, the representation we are describing serves directly as a *point-membership-test* for arbitrary points **p** and Line **L**.

Conventional generative-form of the 3D Line Equation

Taking a more traditional approach, two points $\mathbf{p_0}$ and $\mathbf{p_1}$ define a unique line. The conventional parameterized 3D vector line equation looks like this in computer science notation:

$$\mathbf{p} \leftarrow \mathbf{p_0} + t\mathbf{v} \quad \text{where} \quad \mathbf{v} \leftarrow \mathbf{p_1} - \mathbf{p_0} \qquad\qquad L = [\ \mathbf{v} \quad \mathbf{p_0}\ ]$$

The "←" signifies assignment (information copying). The right side $\mathbf{p_0} + t\mathbf{v}$ is evaluated, then assigned to **p**. **v** is a difference vector (some multiple of the run direction of the line). Variable scalar parameter t multiplies **v** in order to generate variable offsets from $\mathbf{p_0}$. Can this point-generator function double as a point-membership-test? No. You would need parameter value t associated with point **p** before the right side $\mathbf{p_0} + t\mathbf{v}$ can be computed -- the only givens are **p** and [**v** $\mathbf{p_0}$]. [Granted, the 3D vector *cross product* can be employed to bend [**v** $\mathbf{p_0}$] representation into predicate form: (**p** - $\mathbf{p_0}$) **x** **v** == < 0 0 0 >]

From this analysis, there is reason to qualify the parametric 3D line equation $\mathbf{p} = \mathbf{p_0} + t\mathbf{v}$ as a *generative* form. Is there something more versatile about the *predicate form steeped in coordinate rotation* being offered here? What range of problems will it make easier to solve?...in simplest terms, those where *rotating coordinates will simplify the solution*. And, there are many such problems.

Problems with 3D Lines Students Ought to Be Able to Solve (with ordinary mental effort)

A modern 3D line representation should support automated, bug-free, fully-generalized (handles all input cases), degeneracy-flagging, and easy-to-develop algorithmic solutions. Here are 13 problems involving 3D lines successfully conquered by high school college-prep seniors using their software implementations of the [**ZAlignRotator** **XYLocation**] representation:

AXIOMATIC:

- Given points **p1** **p2**, generate line L's numerical representation
- Given run direction $\mathbf{d}_{run}$ and anchor point **p**, generate line L's numerical representation

RUDIMENTARY:

- Generate points along line **L** at desired spacing.
- Decide (T/F) if point **p** is on line **L** (within epsilon *e*).

INTERMEDIATE:

- Decide if Line **L** is parallel to plane **PL** (within epsilon *e*).
- How distant is point **p** from line **L**?
- Given point **p** and line **L**, compute $\mathbf{p}_L$, the point on the line closest to **p**.
- Compute the *intersection of two planes* **PL1**, **PL2** (whose result is line **L**).
- Compute the *intersection of a plane* **PL** *and line* **L** (whose result is a point **p**).
- extrude Line **L** along direction **d** to generate plane **PL**.

ADVANCED:

- For two skew lines **L1**, **L2**, compute their *shortest bridging line segment* **LS**.
- Detect lines **L1**, **L2** being coplanar, and compute their *point of intersection* **p**.
- compute the intersection points of Line **L** piercing Sphere **SPH** (points **p1**, **p2**).

These are well-defined, meaty, and useful problems, all of which have been solved by above-average, high-school, pre-STEM students employing the novel 3D vector line predicate presented here.

Deriving a 3D Line's **ZAlignRotator** and **XYLocation** features

A student experienced in applying 2D coordinate rotations to simplify problems knows how they work computationally. Given the choice of **newXaxis**, input point **p** is transformed into **p'** as two dot products:

p' ← [**p** • **newXaxis** **p** • **newYaxis**] where **newYaxis** ← rotate90ºCCW (**newXaxis**)

This math works because the vector dot product **p** • $\mathbf{d}_{axis}$ projects any point **p** onto any rotated axis $\mathbf{d}_{axis}$ as *a coordinate along that axis*. Students accept that *point projection* works nearly the same way in 3D:

p' ← [**p** • **newXaxis** **p** • **newYaxis** **p** • **newZaxis**]

The new ground to be plowed is how you arrive at the 3 axes. In preparation, students tackle the **Mutual Perpendicular Finder** Project, which introduces the 3D vector cross-product and the normalized cross-product. The latter helps the student find (calculate) the 3D direction mutually perpendicular to two given input vectors **v1, v2** (or **d1 , d2**). The student then moves forward into the **Flight Path Solver Project** (shortest intercontinental flight path from city A to B), which showcases the utility of the normalized cross-product for calculating Great Circle routes.

ZAlignRotator Next, students learn that specifying a Rotator only requires specifying *2 out of 3 axes* -- the missing axis can be auto-filled by the software computing their mutual perpendicular. In the **Hypertube Drilling Project**, this knowhow is put to good use (Fig.3). As a further option, students learn that a Rotator may be specified giving *only one axis* -- the software will *arbitrarily* fill in the two missing axes. This option is well suited for calculating a 3D line's ZAlignRotator:

ZAlignRotator ← [--------- --------- $\mathbf{d}_{run}$] (see details in Appendix A)

We're OK with the software arbitrarily choosing the missing **newXaxis** and **newYaxis**, so long as these are *immutable* during the lifetime of the Line, i.e., giving it stable numerics.

XYLocation Once **ZAlignRotator** is computed, it can be used to compute **XYLocation** given coordinates for any point **p** on the Line. The student simply coordinate-rotates point **p** → **p'**:

p' ← **p** • **ZAlignRotator**

XYLocation ← [p'_x p'_y]

The geometric meaning of XYLocation is illustrated in Fig. 2. After coordinate-rotating line **L**, line **L'** is perfectly aligned with the z-axis, and all points on it share an invariant [x' y'] coordinate-pair. Putting numbers to a 3D line's *spatial location* becomes straighforward in rotated coordinates. That said,

representing an object's feature in a transformed space *does take some getting used to*. The payoff is that a coordinate rotation is "built into" **L**'s representation, making that transform at-the-ready as a problem simplifier. When students later confront intersecting a 3D Line with a Plane or a Sphere (Fig. 4), and can morph the problem into a special case where the Line stands perfectly vertical at known x and y, the benefits of this novel representation will become apparent.

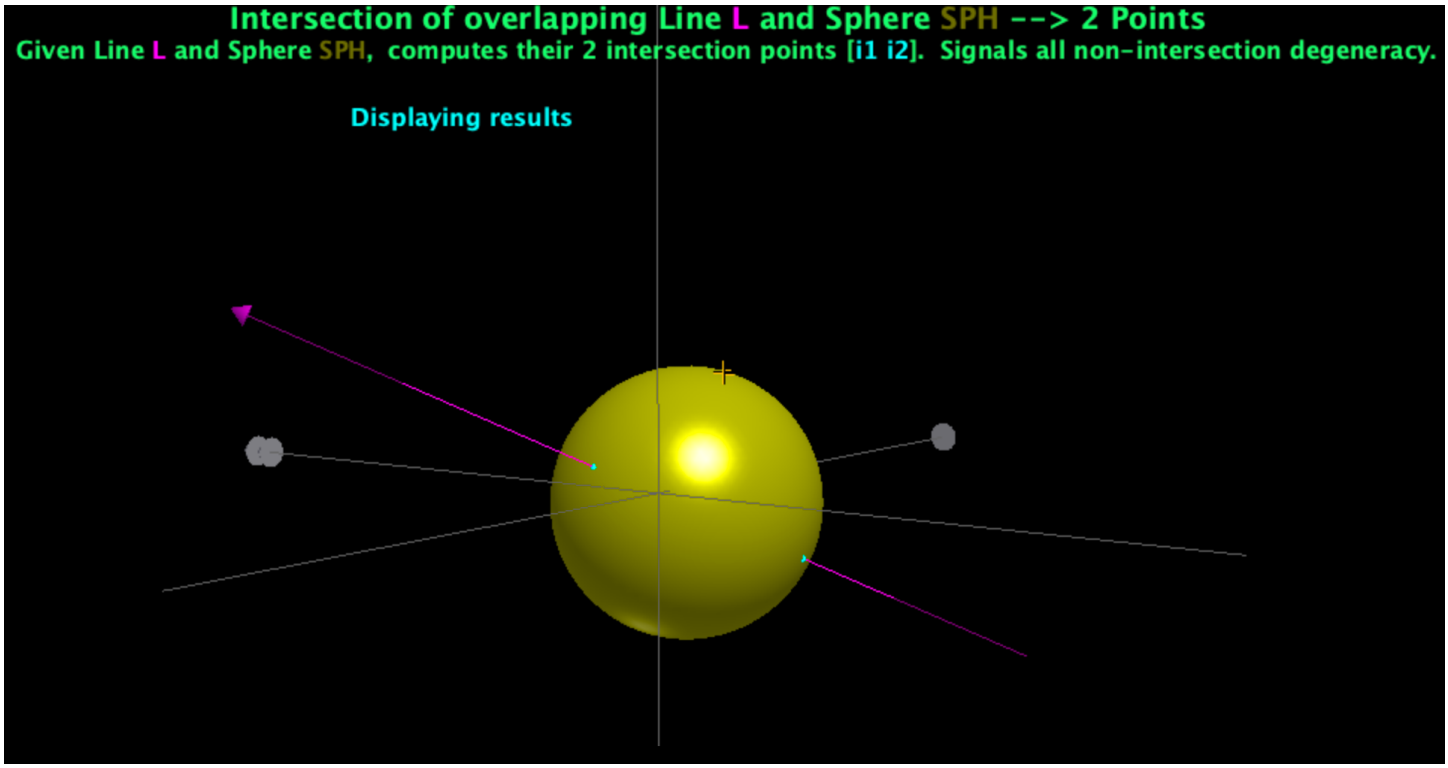

Fig. 4 Example of an advanced 3D problem solved computationally by high school math students.

Discussion

As mentioned at the outset, geometric representations useful for blended mathematics+computation, needn't cater to mental arithmetic. A great example is representing 3D directions in space. For centuries it made sense to use spherical angle-pair [ϕ θ] for 3D pointing directions, in degree units (e.g., geolocation of the Eiffel Tower [2.29450ºE, 48.85822ºN]. Compare that to the equivalent geocentric 3D direction vector:

[0.6573970726109154, 0.026340586810345526, 0.7530838349142049]

Nobody would have thought such a numerical representation the least bit useful before software computing came along to manage all the numbers. But this format is quite preferable for algorithms. Why? Direction vectors eliminate all the *exception-handling baggage* attached to computing with spherical angles -- the asymmetries, ambiguities and singularities. The ancients can hardly be faulted for those oversights.

In algorithmic mathematics, we're not chasing after *minimal representation* (using the fewest numbers), because the more *overcompressed* the numerical information, the more complicated the algorithms. Therefore, what objection can there be to representing 3D lines with 11 numbers (9 for ZAlignRotator, 2 for XYLocation) if doing so eliminates algorithmic complexity and minimizes the possibility for bugs? What high school math teacher wouldn't want bug-avoidance *apriori* baked into the mathematics?

There is a way to connect [ZAlignRotator XYLocation] representation back to a long-established mathematical insight -- that a 3D line can be represented as the intersection of two planes. If we expand the 3D coordinate rotation of point **p** into separate matrix column•row operations, we get:

p • **ZAlignRotator.newXaxis** == XYLocation.x

p • **ZAlignRotator.newYaxis** == XYLocation.y

These are two *plane equations* in predicate form **p** • **o** == ℓ. So, in essence, ZAlignRotator and XYLocation implicitly chunk the information about two intersecting planes determining the line. That observation may give comfort to those trained to represent lines as simultaneous plane equations.

Speaking of mathematical esthetics, here's a question: Should a 3D line's representation refer numerically to any *specific points* on a line while representating *all its points*? The 3D parametric line equation **p** = $\mathbf{p}_0$ + t**v** does so by necessity, citing $\mathbf{p}_0$ to pin down location. But the various 2D line equation forms, slope-intercept (y = mx + b) and standard (ax + by = c) bring together all the included points without ever referring to any particular one. The 3D Vector Line Predicate respects this esthetic. That said, the arguments in favor of it are *pragmatic* in terms of ease-of-use and computational dexterity.

One limitation to be noted is that the theory does not extend neatly to higher dimensions (the way the parametric line equation does). The next higher dimension above 3 which supports N x N rotational matrices is 7.

Summary

A novel form for the 3D Line Equation is put forth -- in the form of a *predicate* able to decide which points are on the line, and which are not. It seems to offer a good balance of human understandability and computational robustness/versatility.

The utility of this representation might not be immediately apparent -- we don't expect the words and formulas on these pages to substitute for the experience of using it in problem-solving. Nor can these inert pages impact you the way dynamic 3D graphics contribute to a deeper intuitive understanding.

If you're willing to try it out (a time commitment of a 1-year math course = 145 hours), it's not hyperbole to predict you'll come away possessing a more powerful command of 3D geometry.

Appendix A. Constructing a Rotator from a Single-axis

The numerical representation for 3D Lines documented here depends on software that can be given *just one* of the 3 axes, and then auto-define the other two (meeting all the criteria for a valid 3D axis-set). How can this be accomplished using algorithmic vector math?

We know that once 2 out of the 3 axes are defined numerically, the 3rd axis is 100% determined geometrically -- and computable using the cross product of the other two. Here are the 3 cases:

Given Axes (assured perpendicular)	Completed Rotator
[**newXaxis** **newYaxis** -------]	[**newXaxis** **newYaxis** **newXaxis x newYaxis**]
[-------- **newYaxis** **newZaxis**]	[**newYaxis x newZaxis** **newYaxis** **newZaxis**]
[**newXaxis** -------- **newZaxis**]	[**newXaxis** **newZaxis x newXaxis** **newZaxis**]

Therefore, we only have to show, given one axis, how a 2nd perpendicular axis can be computed from it.

There are always 3 *reference axis directions* available as inputs to cross-products:

dirX = [1 0 0] **dirY** = [0 1 0] **dirZ** = [0 0 1]

Let's say we're given numerics for **newZaxis**. The task is to arbitrarily compute a **newXaxis**. This approach will always work:

arbitraryAxis = [1 0 0] **arbitraryAlternateAxis** = [0 1 0]

if (**newZaxis x arbitraryAxis** NOT EQUAL TO **[0 0 0]**)

newXaxis ← **newZaxis x arbitraryAxis**

else

newXaxis ← **newZaxis x arbitraryAlternateAxis**

One of these two cross-products *must be non-zero*, and that guarantees a computed **newXaxis** orthogonal to the given **newZaxis**.

The exact same algorithmic approach works for any *given axis* working to compute any *missing 2nd axis*.

This algorithmic math is what makes possible forming a **ZAlignRotator** for 3D extended lines, using the line's run direction $\mathbf{d}_{run}$ (or points **p1 p2**):

ZAlignRotator ← RotatorForNewZAxis (directionFromTo (**p1**, **p2**))

ZAlignRotator ← RotatorForNewZAxis ($\mathbf{d}_{run}$)