Computational Geometry in Python: From Theory to Application

CHARLES MARSH

Charlie (BCS, Princeton) has been an engineering lead at Khan Academy, then Cedar, and nowadays does ML at Spring Discovery.

When people think computational geometry, in my experience, they typically think one of two things:

> Wow, that sounds complicated.
>
> Oh yeah, convex hull.

In this post, I'd like to shed some light on computational geometry, starting with a brief overview of the subject before moving into some practical advice based on my own experiences (skip ahead if you have a good handle on the subject).

What's all the fuss about?

While convex hull computational geometry algorithms are typically included in an introductory algorithms course, computational geometry is a far richer subject that rarely gets sufficient attention from the average developer/computer scientist (unless you're making games or something).

Theoretically intriguing…

From a theoretical standpoint, the questions in computational geometry are often exceedingly interesting; the answers, compelling; and the paths by which they're reached, varied. These qualities alone make it a field worth studying, in my opinion.

For example, consider the Art Gallery Problem: We own an art gallery and want to install security cameras to guard our artwork. But we're under a tight budget, so we want to use as few cameras as possible. How many cameras do we need?

When we translate this to computational geometric notation, the 'floor plan' of the gallery is just a simple polygon. And with some elbow grease, we can prove that $n/3$ cameras is always sufficient for a polygon on n vertices, no matter how messy it is. The proof itself uses dual graphs, some graph theory, triangulations, and more.

Here, we see a clever proof technique and a result that is curious enough to be appreciated on its own. But if theoretical relevance isn't enough for you…

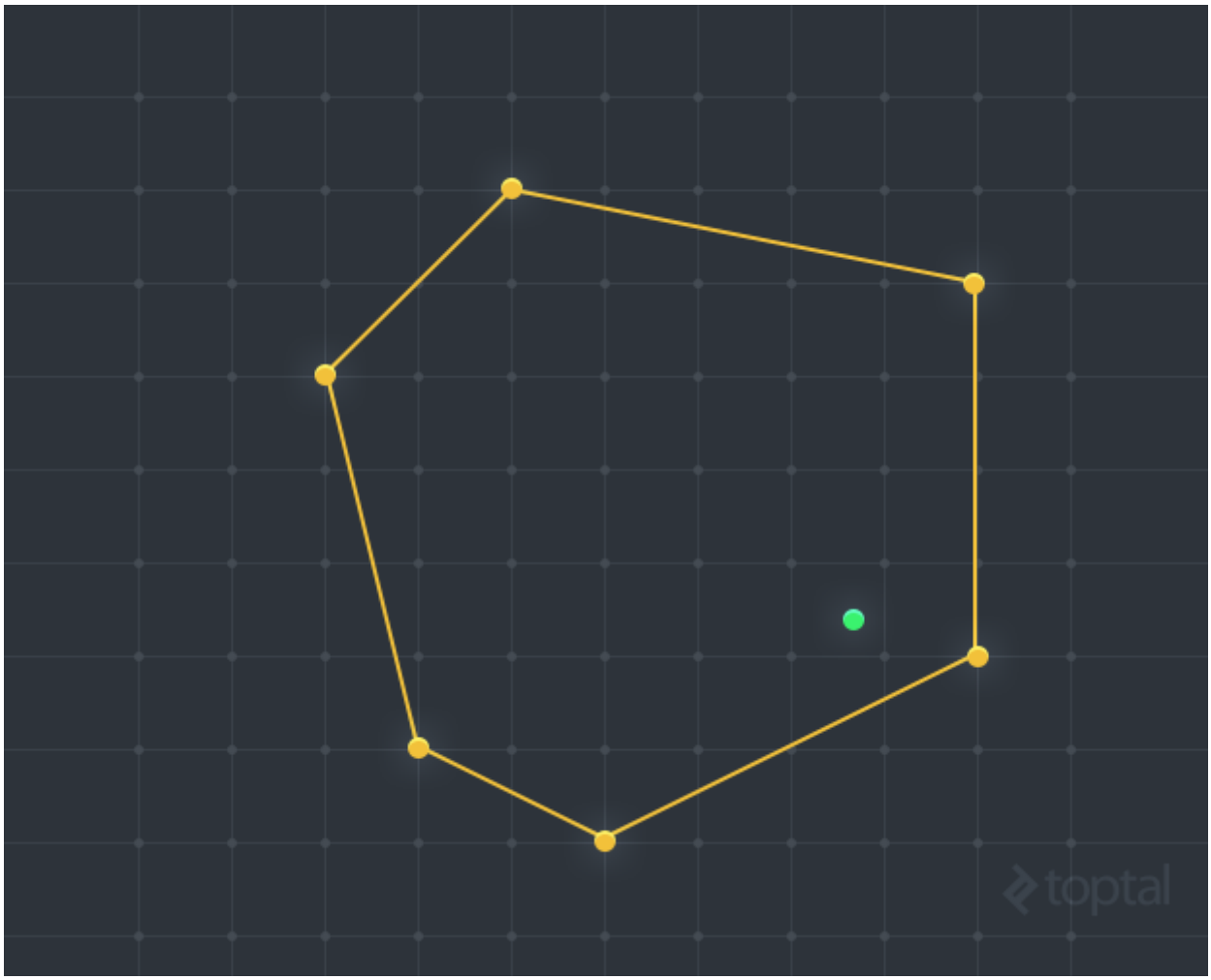
And important in-practice

As I mentioned earlier, game development relies heavily on the application of computational geometry (for example, collision detection often relies on computing the convex hull of a set of objects); as do geographic information systems (GIS), which are used for storing and performing computations on geographical data; and robotics, too (e.g., for visibility and planning problems).

Why's it so tough?

Let's take a fairly straightforward computational geometry problem: given a point and a polygon, does the point lie inside of the polygon? (This is called the point-in-polygon, or PIP problem.)

PIP does a great job of demonstrating why computational geometry can be (deceptively) tough. To the human eye, this isn't a hard question. We see the following diagram and it's *immediately* obvious to us that the point is in the polygon:



Even for relatively complicated polygons, the answer doesn't elude us for more than a second or two. But when we feed this problem to a computer, it might see the following:

```
poly = Polygon([Point(0, 5), Point(1, 1), Point(3, 0),
               Point(7, 2), Point(7, 6), Point(2, 7)])
point = Point(5.5, 2.5)
poly.contains(point)
```

What is intuitive to the human brain does not translate so easily to computer language.

More abstractly (and ignoring the need to represent these things in code), the problems we see in this discipline are very hard to rigorize ('make rigorous') in a computational geometry algorithm. How would we describe the point-in-polygon scenario without using such tautological language as 'A point is inside a polygon if it is inside the polygon'? Many of these properties are so fundamental and so basic that it is difficult to define them concretely.

> How would we describe the point-in-polygon scenario without using such tautological language as 'it's inside the polygon if it's inside the polygon'?

Difficult, but not impossible. For example, you could rigorize point-in-polygon with the following definitions:

- A point is inside a polygon if *any infinite ray beginning at the point intersects with an odd number of polygon edges* (known as the even-odd rule).
- A point is inside a polygon if *it has a non-zero winding number*(defined as the number of times that the curve defining the polygon travels around the point).

Unless you've had some experience with computational geometry, these definitions probably won't be a part of your existing vocabulary. And perhaps that's emblematic of how computational geometry can push you to *think differently*.

Introducing CCW

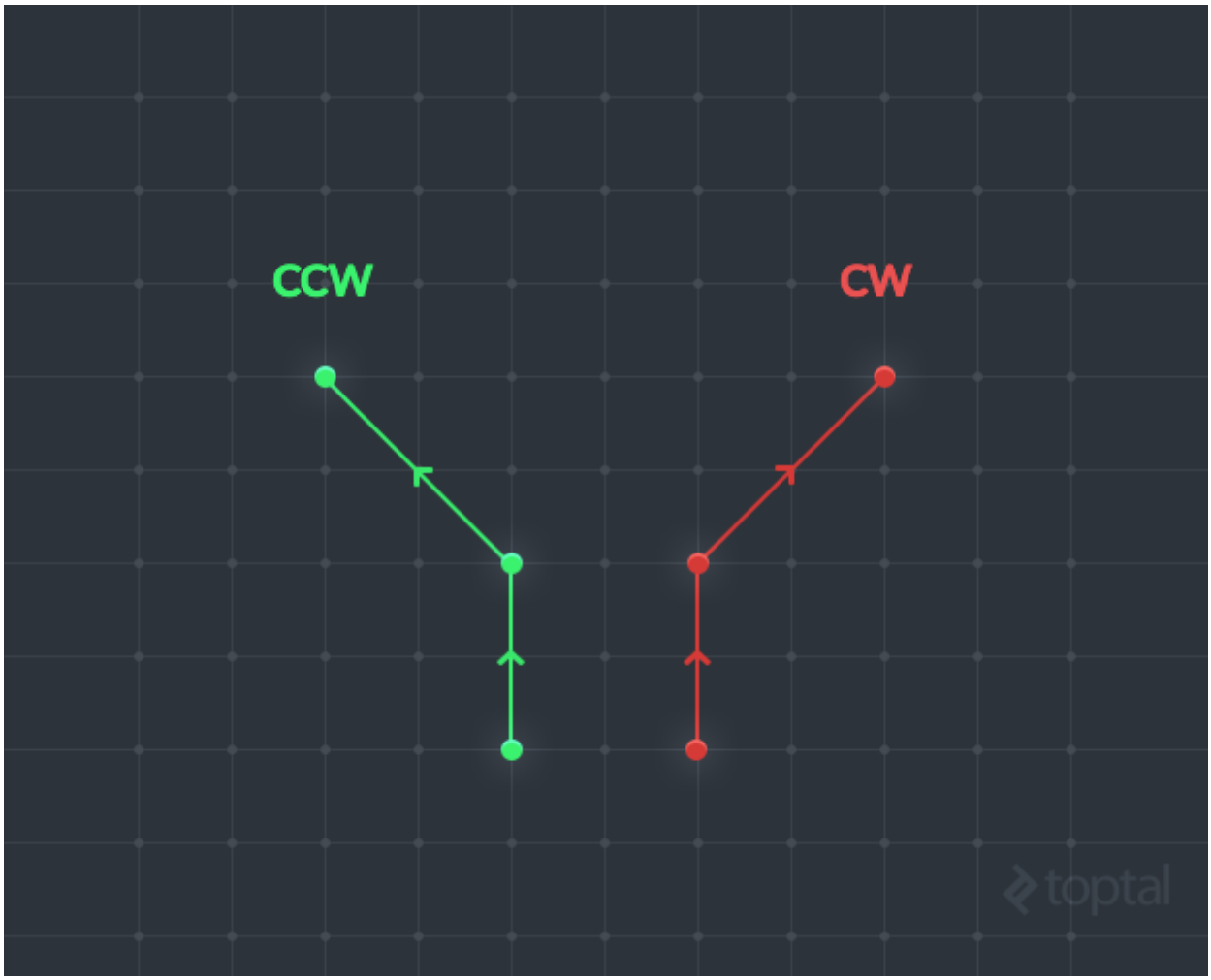
Now that we have a sense for the importance and difficulty of computational geometry problems, it's time to get our hands wet.

At the backbone of the subject is a deceptively powerful primitive operation: counterclockwise, or 'CCW' for short. (I'll warn you now: CCW will pop up again and again.)

CCW takes three points A, B, and C as arguments and asks: do these three points compose a counterclockwise turn (vs. a clockwise turn)? In other words, is A -> B -> C a counterclockwise

angle?

For example, the **green** points are CCW, while the **red** points are not:



Why CCW Matters

CCW gives us a *primitive* operation on which we can build. It gives us a place to start rigorizing and solving computational geometry problems.

To give you a sense for its power, let's consider two examples.

DETERMINING CONVEXITY

The first: *given a polygon, can you determine if it's convex?* Convexityis an invaluable property: knowing that your polygons are convex often lets you improve performance by orders of magnitude. As a concrete example: there's a fairly straightforward PIP algorithmthat runs in Log(n) time for convex polygons, but fails for many concave polygons.

Intuitively, this gap makes sense: convex shapes are 'nice', while concave shapes can have sharp edges jutting in and out—they just don't follow the same rules.

A simple (but non-obvious) computational geometry algorithm for determining convexity is to check that every triplet of consecutive vertices is CCW. This takes just a few lines of Python geometry code (assuming that the `points` are provided in counterclockwise order—if `points` is in clockwise order, you'll want all triplets to be clockwise):

```
class Polygon(object):
    ...
    def isConvex(self):
        for i in range(self.n):
            # Check every triplet of points
            A = self.points[i % self.n]
            B = self.points[(i + 1) % self.n]
            C = self.points[(i + 2) % self.n]
            if not ccw(A, B, C):
                return False
        return True
```

Try this on paper with a few examples. You can even use this result to *define* convexity. (To make things more intuitive, note that a CCW curve from A -> B -> C corresponds to an angle of less than 180°, which is a widely taught way to define convexity.)

LINE INTERSECTION

As a second example, consider line segment intersection, which can also be solved using CCW alone:

```
def intersect(a1, b1, a2, b2):
    """Returns True if line segments a1b1 and a2b2
intersect."""
    return ccw(a1, b1, a2) != ccw(a1, b1, b2) and ccw(a2, b2,
a1) != ccw(a2, b2, b1)
```

Why is this the case? Line segment intersection can also be phrased as: given a segment with endpoints A and B, do the endpoints C and D of another segment lie on the same side of AB? In other words, if the turns from A -> B -> C and A -> B -> D are in the same direction, the segments can't intersect. When we use this type of language, it becomes clear that such a problem is CCW's bread and butter.

A Rigorous Definition

Now that we have a taste for the importance of CCW, let's see how it's computed. Given points A, B, and C:

```
def ccw(A, B, C):
    """Tests whether the turn formed by A, B, and C is ccw"""
    return (B.x - A.x) * (C.y - A.y) > (B.y - A.y) * (C.x -
A.x)
```

To understand where this definition comes from, consider the vectors AB and BC. If we take their cross product, AB x BC, this will be a vector along the z-axis. But in which direction (i.e, +z or -z)? As it turns out, if the cross product is positive, the turn is counterclockwise; otherwise, it's clockwise.

This definition will seem unintuitive unless you have a really good understanding of linear algebra, the right-hand rule, etc. But that's why we have abstraction—when you think CCW, just think of its intuitive definition rather than its computation. The value will be immediately clear.

My Dive Into Computational Geometry and Programming Using Python

Over the past month, I've been working on implementing several computational geometry algorithms in Python. As I'll be drawing on them throughout the next few sections, I'll take a second to describe my computational geometry applications, which can be found on GitHub.

Note: My experience is admittedly limited. As I've been working on this stuff for months rather than years, take my advice with a grain of salt. That said, I learned much in those few months, so I hope these tips prove useful.

Kirkpatrick's Algorithm

At the core of my work was an implementation of Kirkpatrick's Algorithm for point location. The problem statement would be something like: *given a planar subdivision (a bunch of non-overlapping polygons in the plane) and a point P, which polygon contains P?* Think point-in-polygon on steroids—instead of a single polygon, you've got a plane-ful of them.

As a use-case, consider a web page. When a user clicks on the mouse, the web page needs to figure out *what* the user clicked on as quickly as possible. Was it Button A? Was it Link B? The web page is composed of non-overlapping polygons, so Kirkpatrick's Algorithm would be well-positioned to help out.

While I won't discuss the algorithm in-depth, you can learn more here.

Minimum Bounding Triangle

As a subtask, I also implemented O'Rourke's algorithm for computing a minimum enclosing/bounding triangle (that is, finding the smallest triangle that encloses convex a set of points) in linear-time.

Note: Computing the minimum bounding triangle does not help or hurt the asymptotic performance of Kirkpatrick's Algorithm as the computation itself is linear-time—but it's useful for aesthetic purposes.

Practical Advice, Applications, and Concerns

The previous sections focused on why computational geometry can be difficult to reason about rigorously.

In practice, we have to deal with a whole new host of concerns.

Remember CCW? As a nice segue, let's see yet another one of its great qualities: it protects us against the dangers of floating-point errors.

Floating-Point Errors: Why CCW is King

In my computational geometry course, Bernard Chazelle, an esteemed professor who's published more papers than I can count, made it a rule that we couldn't mention angles when attempting to describe an algorithm or a solution.

It became a rule that we couldn't even *mention*angles. Why? Angles are messy—angles are "dirty".

Why? Angles are messy. Angles are "dirty". When you have to compute an angle, you need to divide, or use some approximation (anything involving Pi, for example) or some trigonometric function.

When you have to compute an angle *in code*, you'll almost *always*be approximating. You'll be off by some tiny floating point degree of precision—which matters when you're testing for equality. You may solve for some point in the plane through two different methods and, of course, expect that `p1.x == p2.x and p1.y == p2.y`. But, in reality, this check will fail *often*. Further (and quite obviously), these points will then have different hashes.

To make matters worse, your degree of error will increase as your tiny differences propagate through your computations. (For some more scientific examples, this paper goes through what

can go wrong when computing the convex hull or Delaunay triangulation.)

So, what can we do about this?

ALMOSTEQUAL

Part of the Python computational geometry problem is that we're requiring *exactness* in a world where things are *rarely* exact. This will become a problem more often than when handling angles. Consider the following:

```
# Define two random points
p1 = RandomPoint()
p2 = RandomPoint()

# Take the line through them
l1 = Line(p1, p2)

# Shift both points up by sqrt(2)
p1.y += sqrt(2)
p2.y += sqrt(2)

l2 = Line(p1, p2)

# Slope 'should' be the same?
if abs(l1.slope - l2.slope) > 0:
    print "Error!"
# Error!
```

In fact, this code will print "Error!" roughly 70% of the time (empirically). We can address this concern by being slightly more lenient with our definition of equality; that is, by sacrificing a degree of accuracy.

One approach I've used (and seen in, e.g., some OpenCV modules) is to define two numbers as equal if they differ only by some small value epsilon. In Python, you might have:

```
def almostEqual(x, y, EPSILON=1e-5):
    return abs(x - y) < EPSILON

class Point(object):
    ...
    def __eq__(self, that):
        return (almostEqual(self.x, that.x) and
                almostEqual(self.y, that.y))
```

In practice, this is very helpful. Rarely, if ever, would you compute two points that differ by less than 1e-5 that are actually meant to be different points. I highly recommend implementing this type of override. Similar methods can be used for lines, for example:

```
class Line(object):
    ...
    def __eq__(self, that):
        return (almostEqual(self.slope, that.slope) and
                almostEqual(self.intercept, that.intercept))
```

More advanced solutions have been proposed, of course. For example, the 'exact geometric computation' school of thought (described in this paper) aims to have all decision paths in a program depend solely on the *sign* of some computation, rather than its exact numerical value, taking away many of the concerns related to floating-point computations. Our *near equality* approach just scratches the surface, but will often be sufficient in practice.

CCW IS KING

At a higher level, it's (arguably) problematic that we even *define*our solutions in terms of such exact computational quantities as angles or point coordinates. Rather than addressing the symptoms alone (i.e., washing over floating-point errors with `almostEqual`), why not address the cause? The solution: instead of thinking in terms of angles, **think in terms of CCW**, which will help to abstract away the concerns associated with floating-point computation.

Here's a concrete example: lets say you have some convex polygon P, a vertex v, and some point u outside of the polygon. How can you figure out if the line uv intersects P above or below v, or not at all, in constant time?

The brute force solution (besides being linear-time, rather than constant) would be problematic as you'd have to compute some exact line intersection points.

One constant-time approach I've seen involves:

- Calculating some angles using `arctan2`.
- Converting these angles to degrees by multiplying by 180/Pi.
- Examining the relationships between these various angles.

Luckily, the author used the `almostEqual` technique above to smooth over the floating-point errors.

In my opinion, it'd be better to avoid the issue of floating-point errors entirely. If you take a few minutes to look at the problem on paper, you can get a solution based entirely on CCW. The intuition: if the vertices adjacent to v are on the same side of uv, then the line does not intersect; else, see if u and v are on the same side of the line between the adjacent vertices and, depending on the result, compare their heights.

Here's the Python code for testing intersection above v(intersection below just reverses the direction of the comparisons):

```
def intersectsAbove(verts, v, u):
    """
        Returns True if uv intersects the polygon defined by
'verts' above v.
        Assumes v is the index of a vertex in 'verts', and u
is outside of the
        polygon.
    """
    n = len(verts)

    # Test if two adjacent vertices are on same side of line
(implies
    # tangency)
    if ccw(u, verts[v], verts[(v - 1) % n]) == ccw(u,
verts[v], verts[(v + 1) % n]):
        return False

    # Test if u and v are on same side of line from adjacent
    # vertices
    if ccw(verts[(v - 1) % n], verts[(v + 1) % n], u) ==
ccw(verts[(v - 1) % n], verts[(v + 1) % n], verts[v]):
        return u.y > verts[v].y
    else:
        return u.y < verts[v].y
```

The solution isn't immediately obvious to the naked eye, but it's in the *language* of a computational geometry algorithm: 'same side of the line' is a classic element of that trusty algorithm.

Done is Better than Perfect

In the computational geometric literature, there's often a fair amount of wizardry involved in seemingly simple operations. This gives you a choice: you can do things the hard way, following some paper that defines an incredibly advanced solution to a not-so-advanced problem—or you can do things the easy way with a bit of brute force.

Again, I'll use an example: sampling a random interior point from an arbitrary polygon. In other words, I give you some simple polygon, and you give me a random point inside it (uniformly distributed across the polygon).

Often, interior points are required for testing. In that case, you don't have any specific runtime requirements on the computational geometry algorithm that produces them (within reason). The quick and dirty solution, which takes ~2 minutes to implement, would be to pick a random point within a box containing the polygon and see if the point itself is within the polygon.

For example, we may miss twice and find a valid sample only on the third point:

Here's the code:

```
class Polygon(object):
    ...
    def interiorPoint(self):
        """Returns a random point interior point"""
        min_x = min([p.x for p in self.points])
        max_x = max([p.x for p in self.points])
        min_y = min([p.y for p in self.points])
        max_y = max([p.y for p in self.points])

        def x():
            return min_x + random() * (max_x - min_x)

        def y():
            return min_y + random() * (max_y - min_y)

        p = Point(x(), y())
        while not self.contains(p):
            p = Point(x(), y())

        return p

    def contains(self, p):
        for i in range(self.n):
            p1 = self.points[i]
            p2 = self.points[(i + 1) % self.n]
            p3 = self.points[(i + 2) % self.n]
            if not ccw(p1, p2, p3):
                return False
        return True
```

This is known as *rejection sampling*: take random points until one satisfies your criteria. While it may require several samples to find a point that meets your criteria, in practice, **the difference will be negligible** for your test suite. So why work any harder? In summary: don't be afraid to take the dirty route when the occasion calls for it.

By the way: if you want an *exact* algorithm for random sampling, there's a clever one here that I've implemented below. The gist of it:

Triangulate your polygon (i.e., break it into triangles).

Choose a triangle with probability proportional to its area.

Take a random point from within the chosen triangle (a constant-time operation).

Note that this algorithm requires you to triangulate your polygon, which immediately imposes a different runtime bound on the algorithm, as well as the necessity that you *have* a library for triangulating arbitrary polygons (I used poly2tri with Python bindings).

```
from p2t import CDT

class Triangle(object):
    ...
    def area(self):
        return abs((B.x * A.y - A.x * B.y) + (C.x * B.y - B.x
* C.y) + (A.x * C.y - C.x * A.y)) / 2

    def interiorPoint(self):
        r1 = random()
        r2 = random()
        # From http://www.cs.princeton.edu/~funk/tog02.pdf
        return (1 - sqrt(r1)) * A + sqrt(r1) * (1 - r2) * B +
r2 * sqrt(r1) * C


class Polygon(object):
    ...
    def triangulate(self):
        # Triangulate poly with hole
        cdt = CDT(poly.points)
        triangles = cdt.triangulate()

        def convert(t):
            A = Point(t.a.x, t.a.y)
            B = Point(t.b.x, t.b.y)
            C = Point(t.c.x, t.c.y)
            return Triangle(A, B, C)
        return map(convert, triangles)
```

```
    def interiorPoint(self):
        # Triangulate polygon
        triangles = self.triangulate()
        areas = [t.area() for t in triangles]
        total = sum(areas)
        # Calculate normalized areas
        probabilities = [area / total for area in areas]
        weighted_triangles = zip(triangles, probabilities)

        # Sample triangles according to area
        r = random()
        count = 0
        for (triangle, prob) in weighted_triangles:
            count += prob
            # Take random point from chosen triangle
            if count > r:
                return triangle.interiorPoint()
```

Hopefully, the extra effort is evident from the code. Remember: as they say at Facebook, "done is better than perfect". The same goes for computational geometry problems.

Visual and Automated Testing

As many of the problems you work on in computational geometry are defined in terms of easily visualizable qualities or quantities, visual testing is *particularly* important—although insufficient on its own. The ideal test suite will have a combination of visual and randomized automated testing.

> The ideal test suite will have a combination of visual and randomized automated testing.

Again, we proceed by example. Consider testing our implementation of Kirkpatrick's Algorithm. At one step, the algorithm needs to bound the given polygon by a triangle and triangulate the region between the polygon and the outer triangle. Here's a visual example, where the solid green line defines the initial polygon, and the dashed lines define the triangulated region:

Confirming that this triangulation has been executed correctly is very difficult to verify through code, but is immediately evident to the human eye. *Note: I highly suggest using Matplotlib to aid in your visual testing—there's a nice guide here.*

Later, we will want to verify that the algorithm correctly locates points. A randomized, automated approach would be to generate a bunch of interior points for every polygon and make sure that we return the desired polygon. In code:

```
class TestLocator(unittest.TestCase):
    ...
    def runLocator(self, polygons):
        # Pre-process regions
        l = Locator(polygons)

        # Ensure correctness
        for polygon in polygons:
            # Test 100 random interior points per region
            for k in range(100):
                target = polygon.interiorPoint()
                target_polygon = l.locate(target)
                self.assertEqual(polygon, target_polygon)

self.assertTrue(target_polygon.contains(target))
```

We could then use the `runLocator` method on different sets of polygons, giving us a well-diversified test suite.

Open-Source Solutions

Computational geometry has a nice suite of open-source libraries and solutions available regardless of your programming language of choice (although C++ libraries seem to crop up a disproportionate amount).

The benefits of using existing open-source solutions (as with scientific computing in Python) are well-known and have been discussed extensively, so I won't go on about it here. But I thought I'd mention a few Python-centric resources that I found useful:

- poly2tri: a great library for fast triangulations of polygons. Also supports (and this is often crucial) polygons with *holes* in them. Written in C++, poly2tri also has Python bindings and was quite easy to get up and running. See my `triangulate` method above for a taste for the function calls.

- scipy.spatial: includes functions for computing convex hulls, Delaunay Triangulations, and more. Fast (as always), reliable, etc. *Note: I found it useful to use my own* `Point` *datatype with a* `toNumpy` *method:* `def np(self): return [self.x, self.y]`. *Then, I could easily call scipy.spatial methods, e.g.:* `scipy.spatial.ConvexHull(np.array(map(lambda p: p.np()), points))`.
- OpenCV: the open-source computer vision library has some nice standalone computational geometry modules. In particular, I used its minimum enclosing triangle function for a while before implementing it myself.

Conclusion

I hope this post has given you a taste for the beauty of computational geometry as a Python developer, a subject rich with fascinating problems and equally fascinating applications.

In practice, computational geometric implementations present unique challenges that will push you to exercise new and exciting problem-solving skills.

If you're interested in learning more or have any questions for me, I can be reached at charlie@toptal.com.

TAGS Python Matplotlib Scipy Numpy OpenCV