Computational Geometry

Austin Bingham
COFOUNDER - SIXTY NORTH

@austin_bingham

Robert Smallshire
COFOUNDER - SIXTY NORTH

@robsmallshire

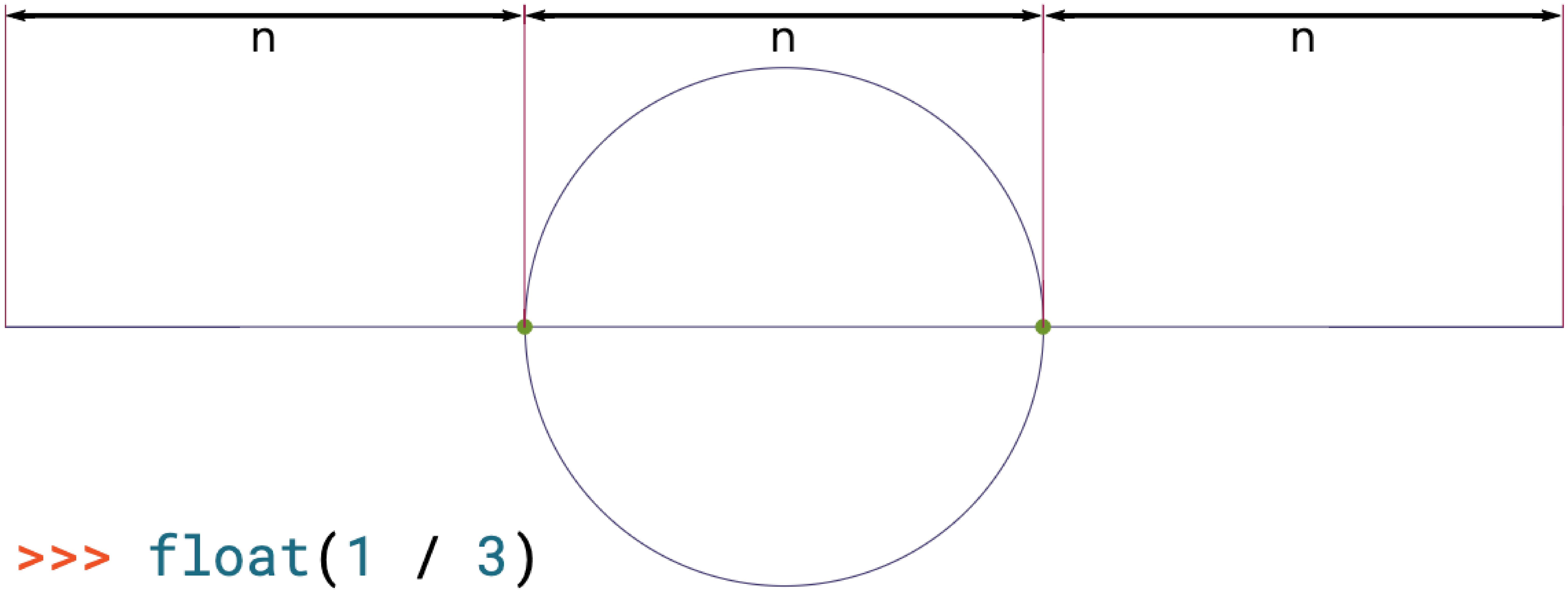
Overview

A problem from the domain of computational geometry

Initial implementation using `float`

Improved implementation with a different numeric type

Computational Geometry




```
>>> float(1 / 3)
0.3333333333333333
```

Rational Numbers

1/3

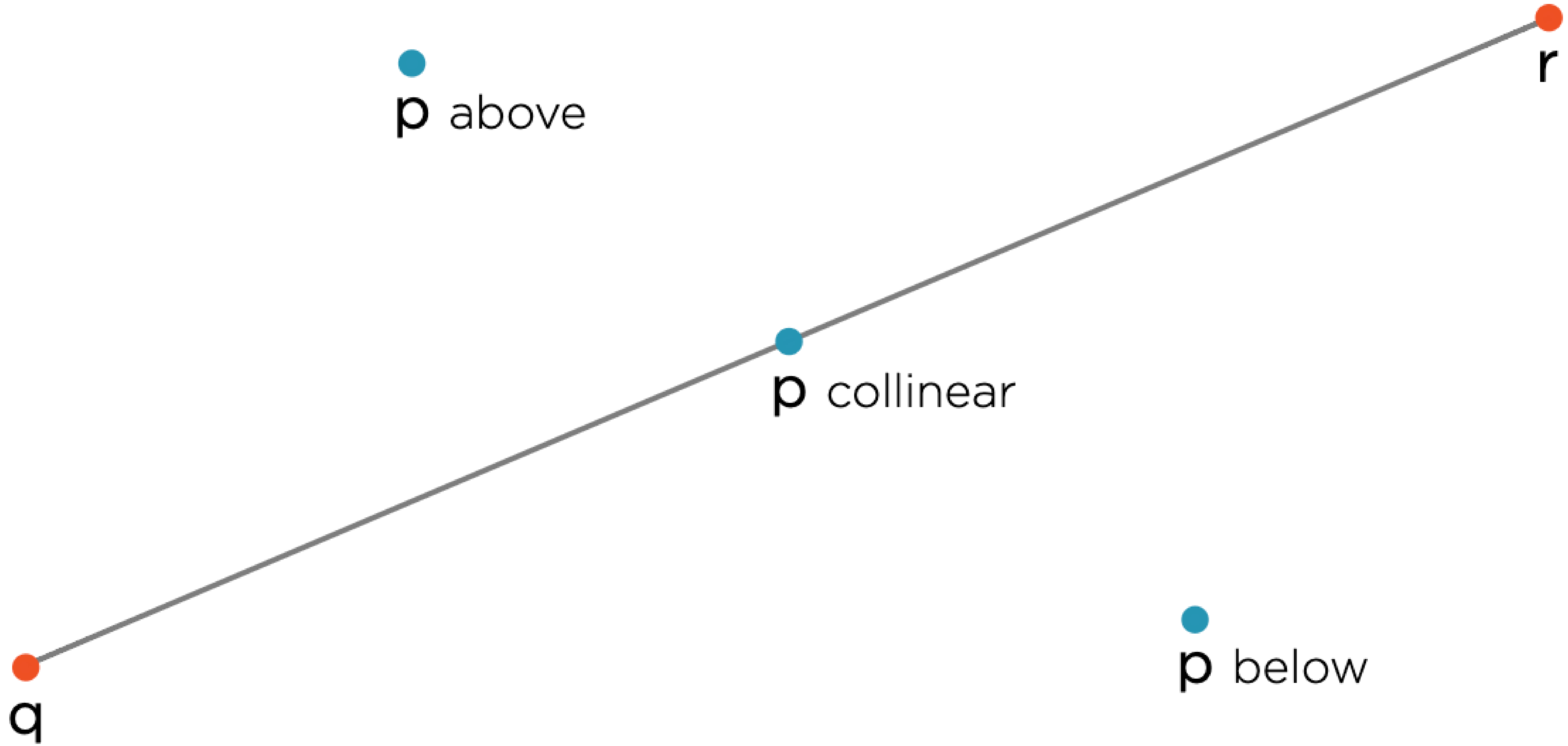
`Fraction` can be used for implementing robust geometric algorithms with rational numbers

These can be surprising and must avoid calculations with irrational numbers

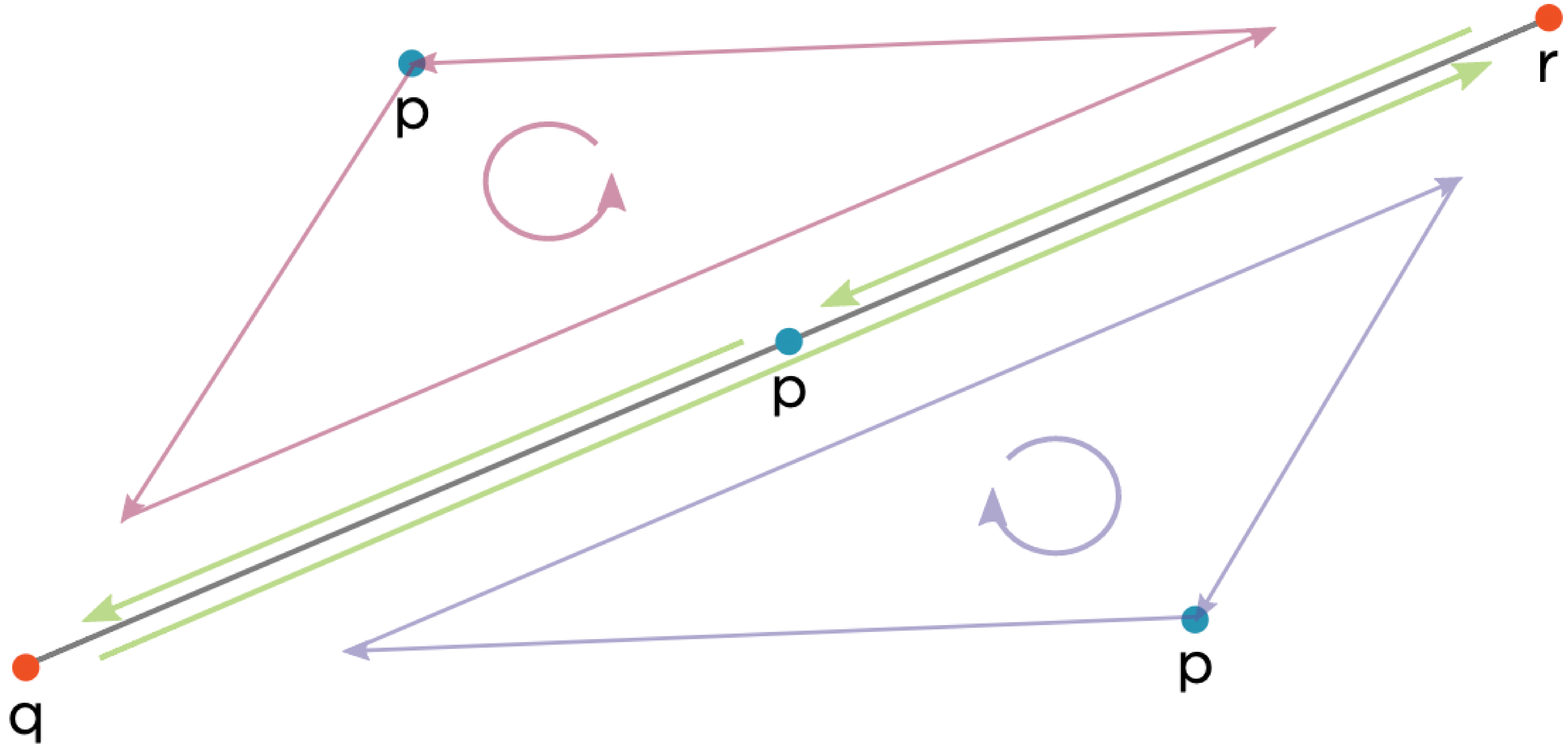
Operations like square-root are not allowed!

Collinearity

Collinearity

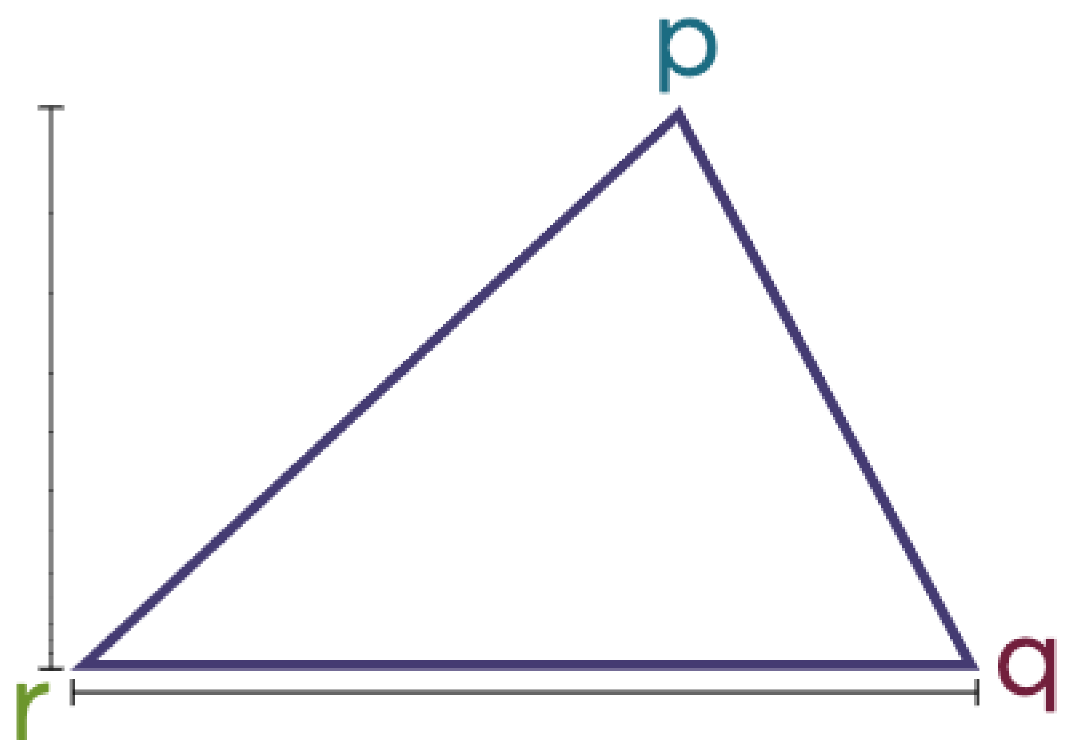


Orientation Test



Mathematics

$$\text{sign}\left(\det\begin{bmatrix} 1 & px & py \\ 1 & qx & qy \\ 1 & rx & ry \end{bmatrix}\right)$$



Orientation Test

$$\operatorname{sign}\left(\det \begin{bmatrix} 1 & px & py \\ 1 & qx & qy \\ 1 & rx & ry \end{bmatrix}\right)$$

+1
counterclockwise
above

0
straight
on

-1
clockwise
below

Supporting Functions

We need functions for computing signs and determinants

Both are straightforward, if not entirely obvious

Sign Function

Sign Function

```
>>> sign(-5)
-1
>>> sign(5)
1
>>> sign(0)
0
>>> def orientation(p, q, r, num_type=float):
...     p = tuple(map(num_type, p))
...     q = tuple(map(num_type, q))
...     r = tuple(map(num_type, r))
...     d = (q[0] - p[0]) * (r[1] - p[1]) - (q[1] - p[1]) * (r[0] - p[0])
...     return sign(d)
...
>>> a = (0, 0)
>>> b = (4, 0)
>>> c = (4, 3)
>>> orientation(a, b, c)
1
>>> orientation(a, c, b)
-1
>>> d = (8, 6)
>>> orientation(a, c, d)
0
>>>
```

Determinant Formula

$$det = (qx - px) * (ry - py) - (qy - py) * (rx - px)$$

So far, so good!

All of our work has avoided `float`, so we haven't had to deal with loss of precision

What if we use `float` for our input data?

Using Float

```
>>> e = (0.5, 0.5)
>>> f = (12.0, 12.0)
>>> g = (24.0, 24.0)
>>> orientation(e, f, g)
0
>>> e = (0.5, 0.5000000000000018)
>>> orientation(e, f, g)
1
>>> e = (0.5, 0.5000000000000019)
>>> orientation(e, f, g)
0
>>> e = (0.5, 0.5000000000000044)
>>> orientation(e, f, g)
0
>>> e = (0.5, 0.5000000000000046)
>>> orientation(e, f, g)
1
>>>
```

float Transect

```
>>> px = 0.5
>>> q = (12.0, 12.0)
>>> r = (24.0, 24.0)
>>> os = [orientation((px, py), q, r) for py in pys]
>>> print(os)
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
 -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
 -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
 -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
 -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, -1, -1, -1,
 -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1,
 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
>>>
```

You should now be wary of using `float` for geometric computation.
Trying to use tolerances or similar techniques will only move the fringing effect around.

A More Appropriate Numeric Type

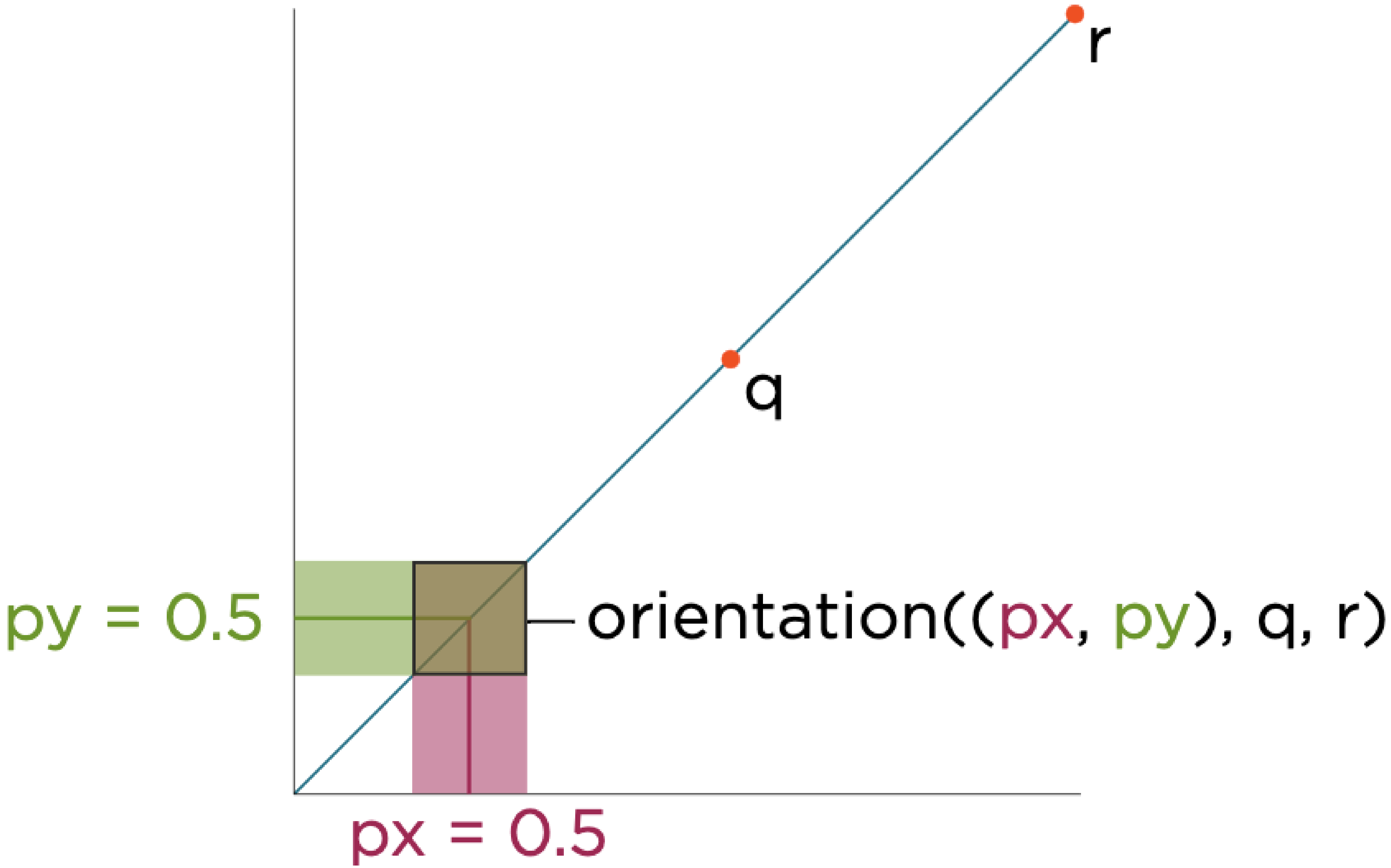
Fraction Transect

```
>>> from fractions import Fraction
>>> px = 0.5
>>> q = (12.0, 12.0)
>>> r = (24.0, 24.0)
>>> os = [orientation((px, py), q, r, num_type=Fraction) for py in pys]
>>> print(os)
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
 -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
 -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
 -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
 -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
 -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
 -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
 -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
 -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
>>>
```

Using `Fraction` we get exact results because we have effectively infinite precision.

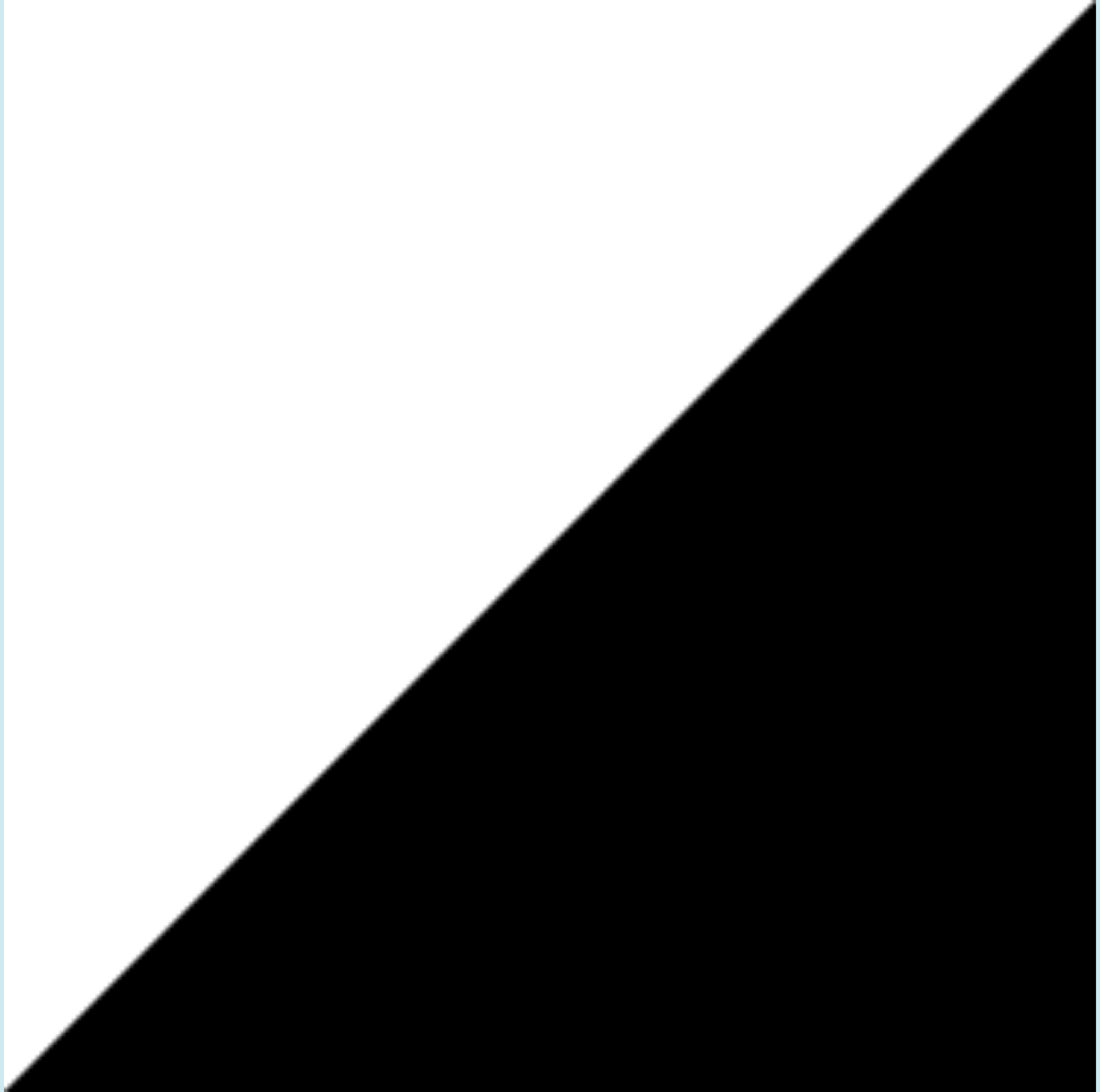
Rendering the Results

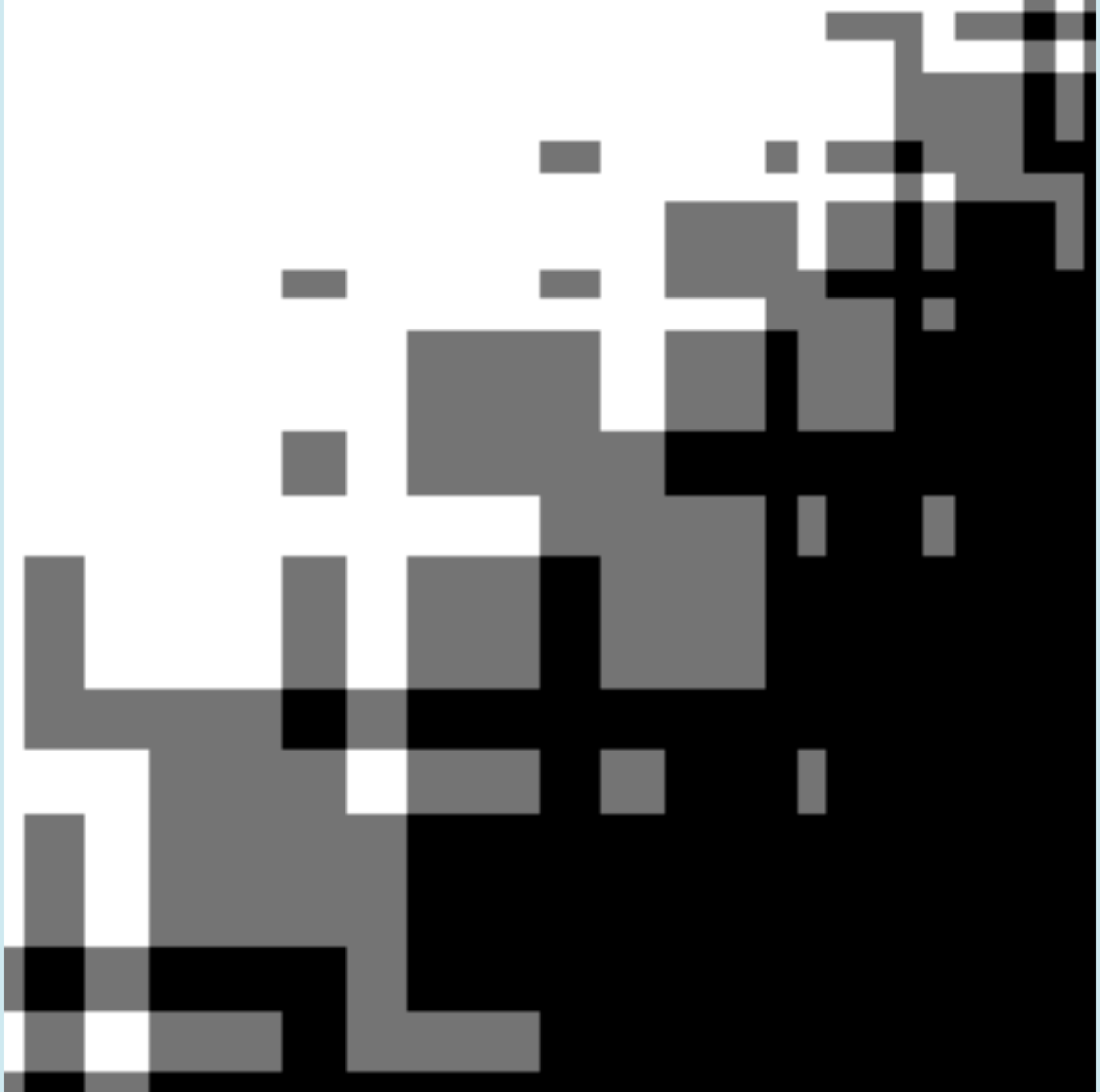
Transect



Render Transect

```
>>> import bmp
>>> color = {-1: 0, 0: 127, +1: 255}
>>> pixels = [[color[orientation((px, py), q, r, num_type=Fraction)]
...            for px in pys]
...           for py in reversed(pys)]
>>> bmp.write_grayscale('above_below.bmp', pixels)
>>> pixels = [[color[orientation((px, py), q, r)]
...            for px in pys]
...           for py in reversed(pys)]
>>> bmp.write_grayscale('above_below.bmp', pixels)
>>>
```





Summary

Computational geometry can be fascinating and counter-intuitive

Naïve use of `float` can lead to misleading or incorrect results

`Fraction` can avoid some shortcomings of `float`, though it's often much slower

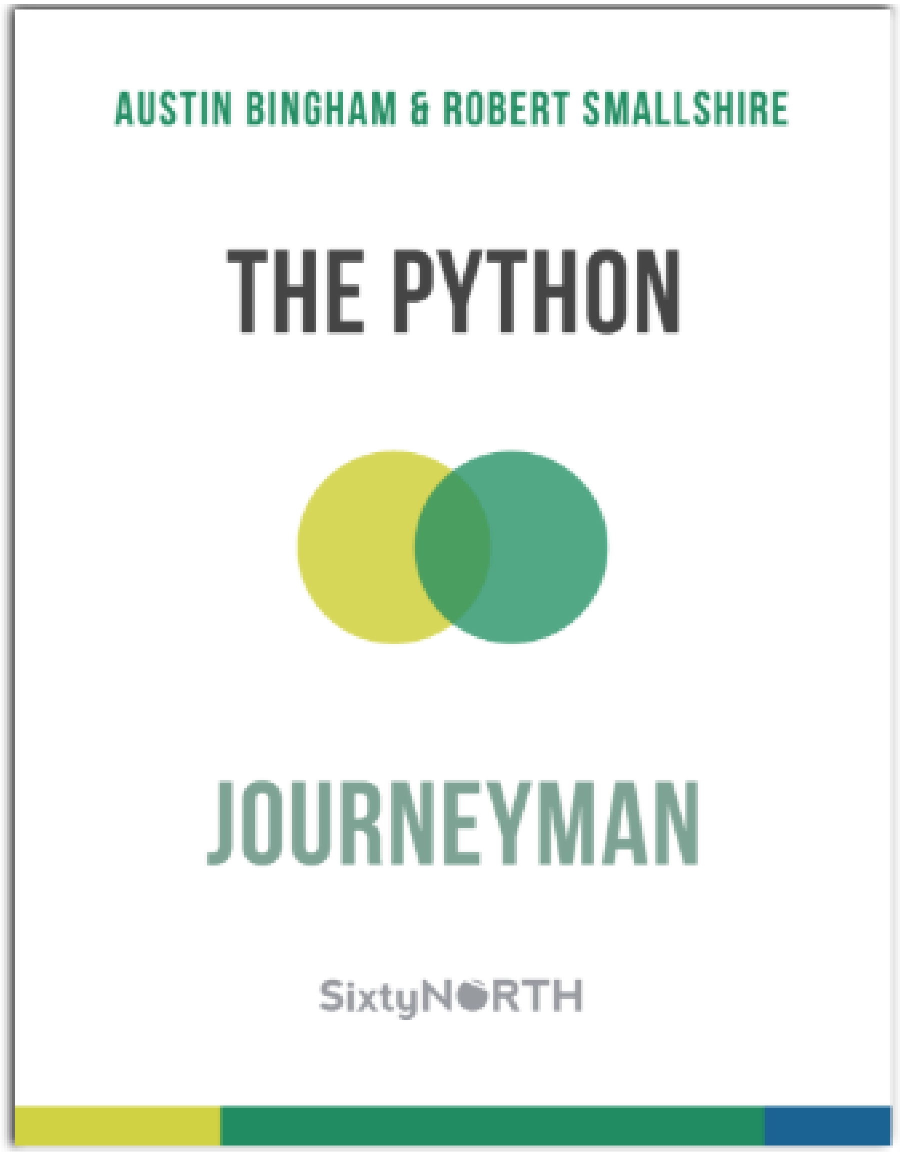
Well done!

Core Python

on

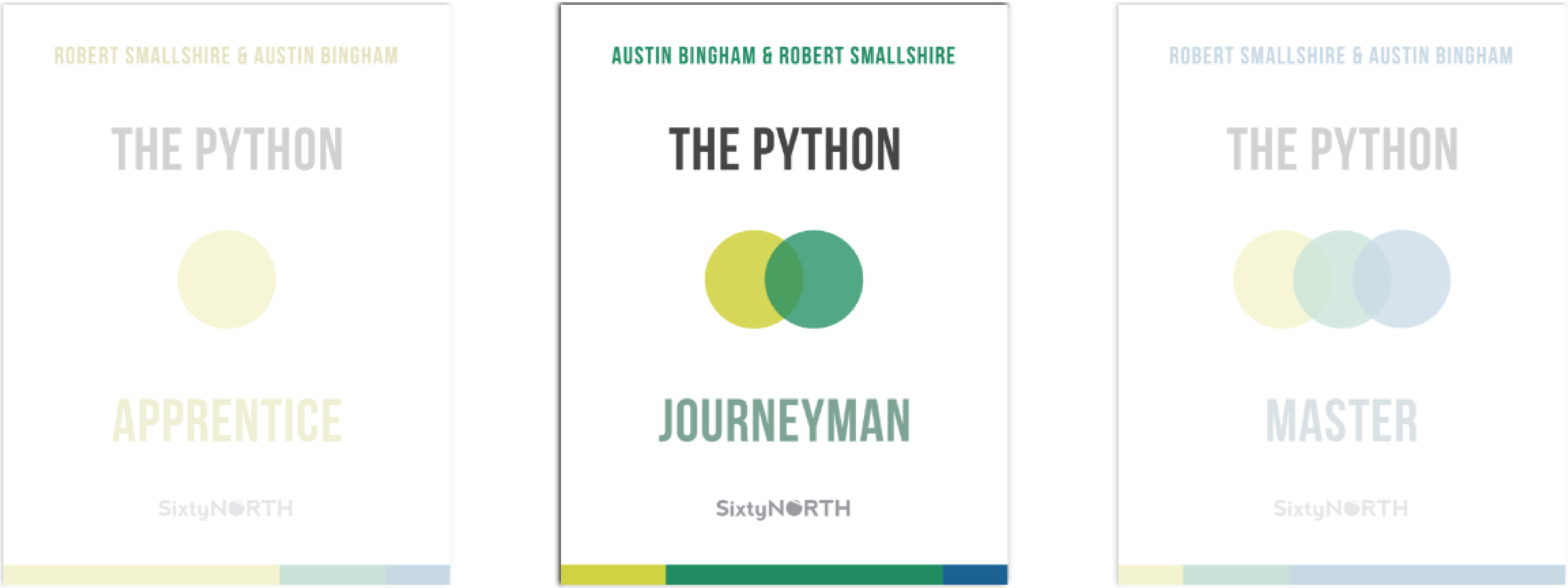
PLURALSIGHT

The Python Craftsman



leanpub.com/b/python-craftsman

The Python Journeyman



leanpub.com/python-journeyman

Happy Programming!