Project 2: Line Simplification

A class that implements the Ramer-Douglas-Peucker line approximation algorithm

Topics

Classes and objects:

- Accessing fields and methods in an object

Algorithm design with recursion:

- Including careful thought about the order in which sub-problems are solved

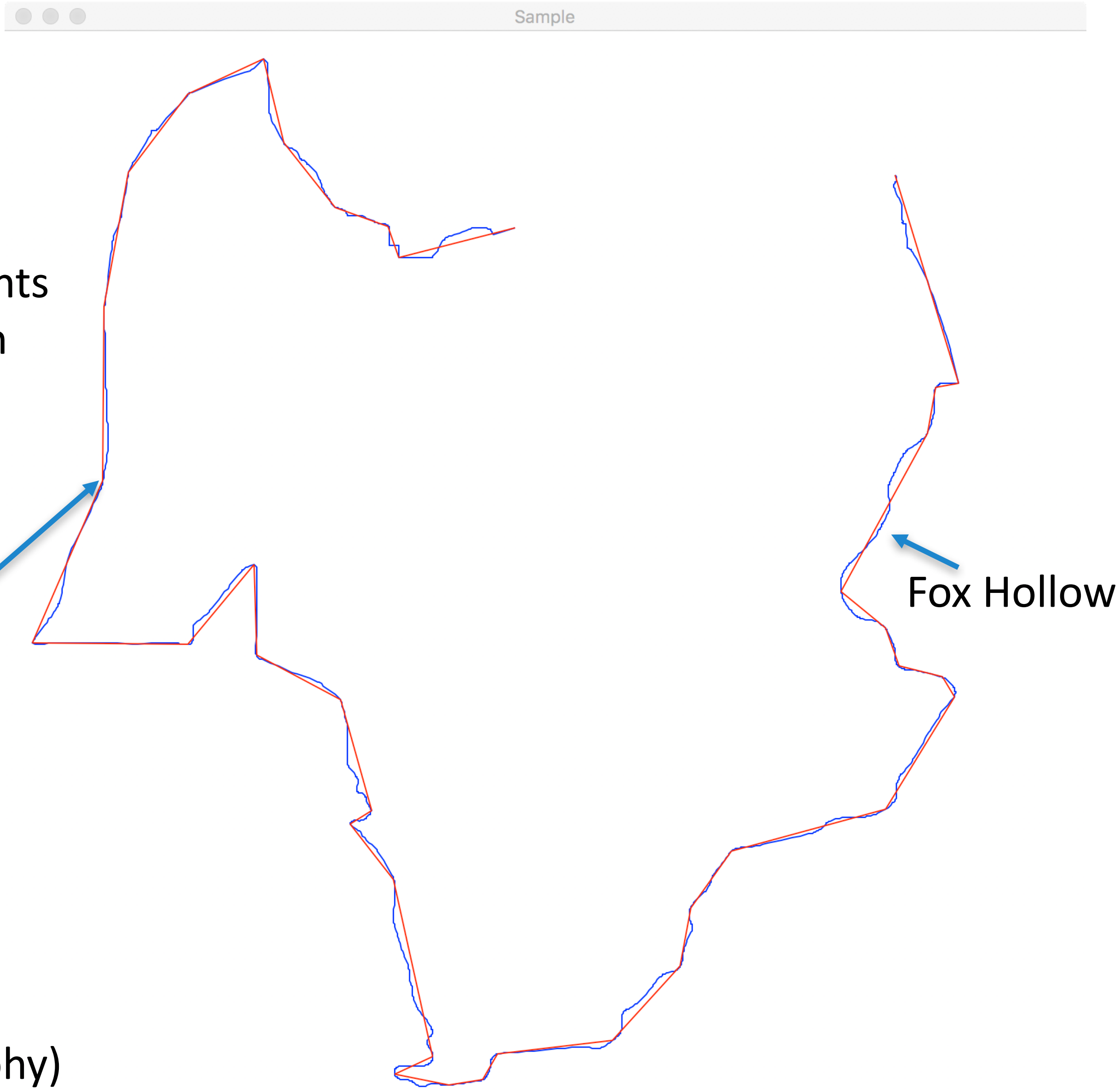
Model-View-Controller pattern:

- Modular design to separate graphics from application logic

Ramer-Douglas-Peucker

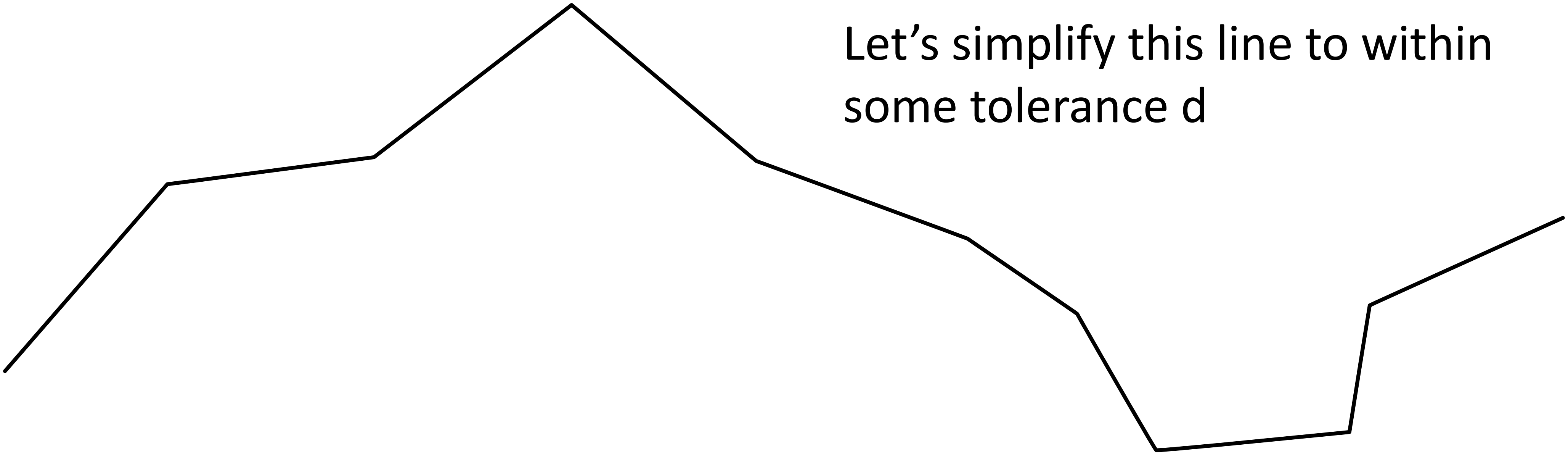
Approximating a polyline with a minimum of points

Blue: Original path, 1502 points
Red: Approximation to within 100 meters, 37 points

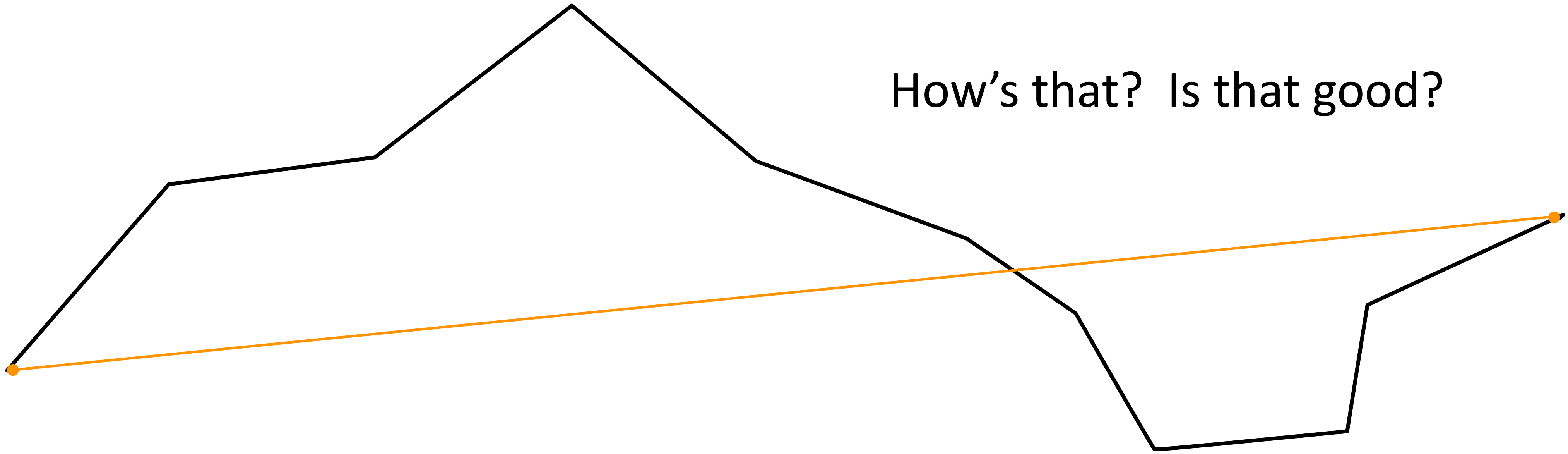


Also known as:
Douglas-Peucker (in cartography)
Duda-Hart split-and-merge (in robotics)

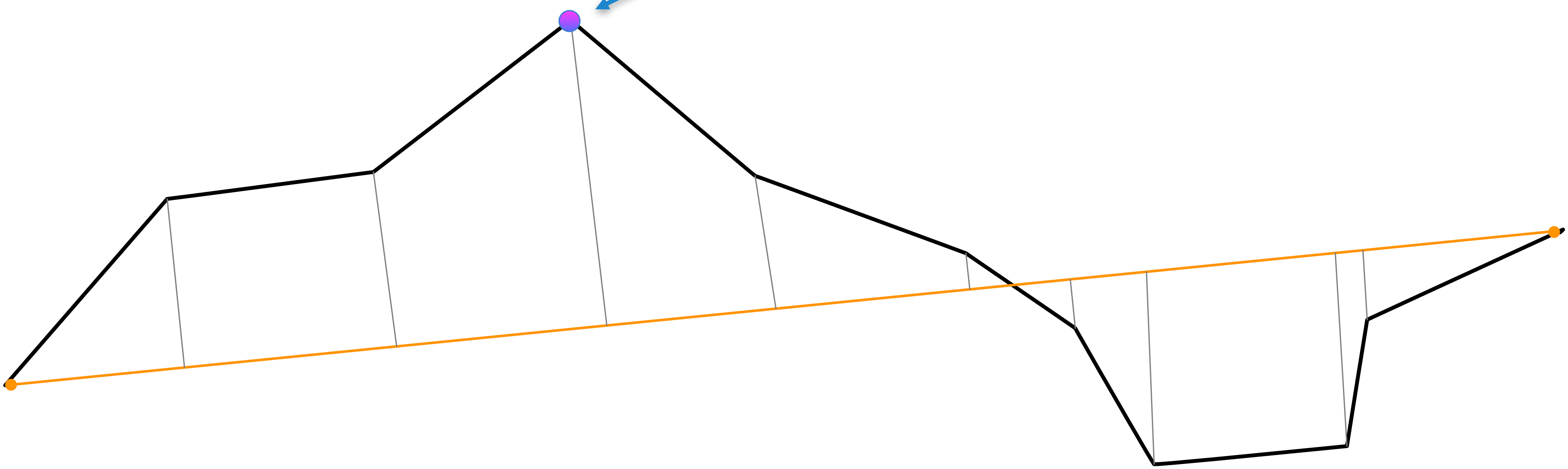
How it works …



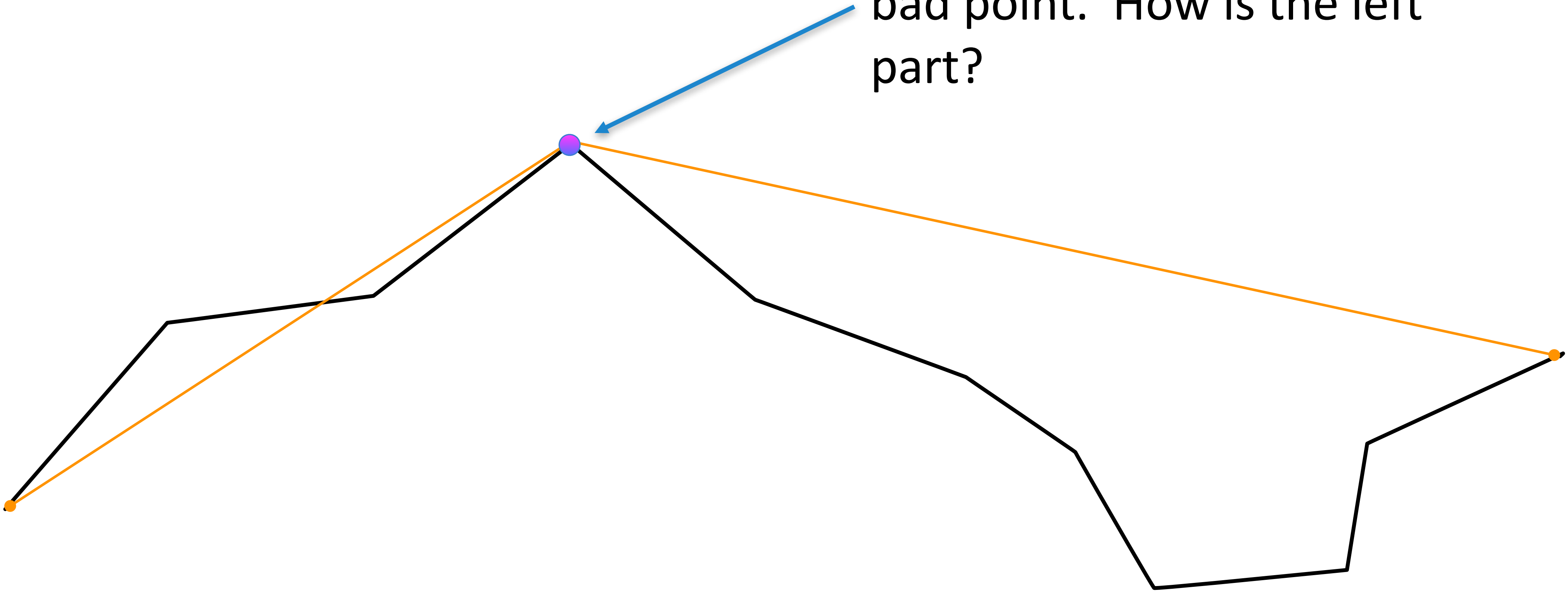
Let’s simplify this line to within some tolerance d



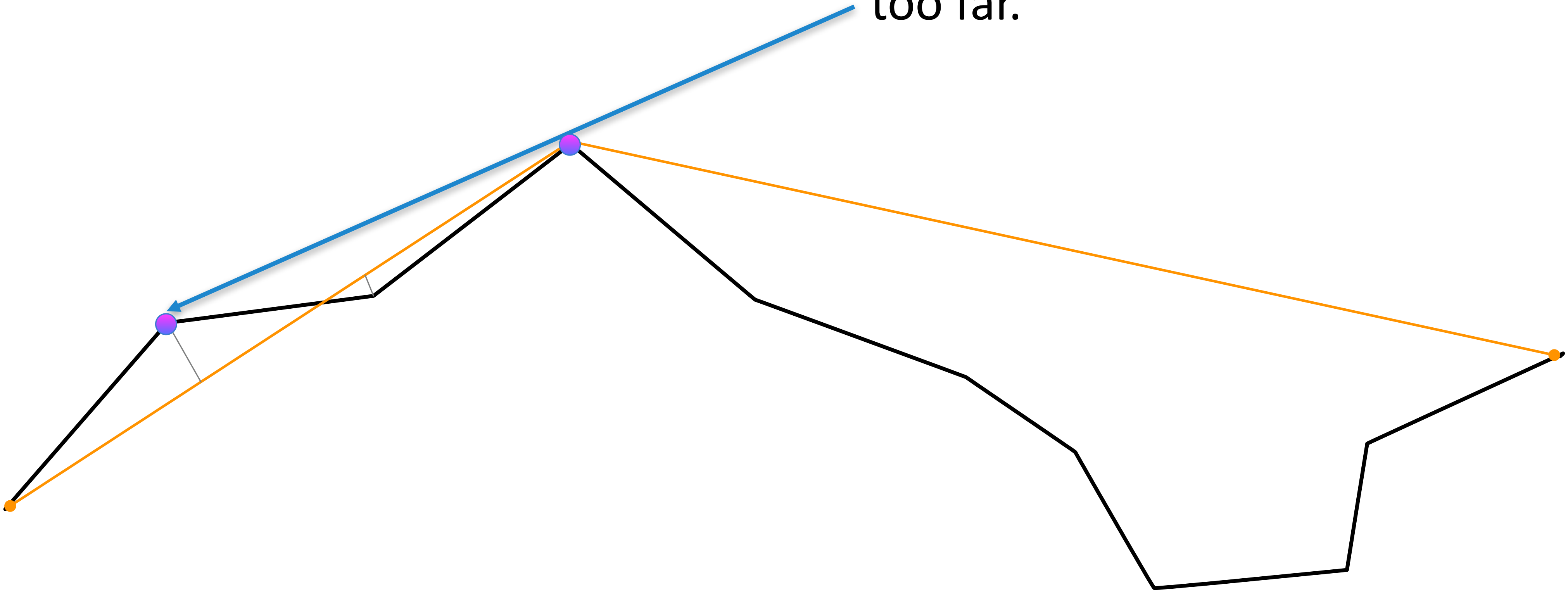
How’s that? Is that good?



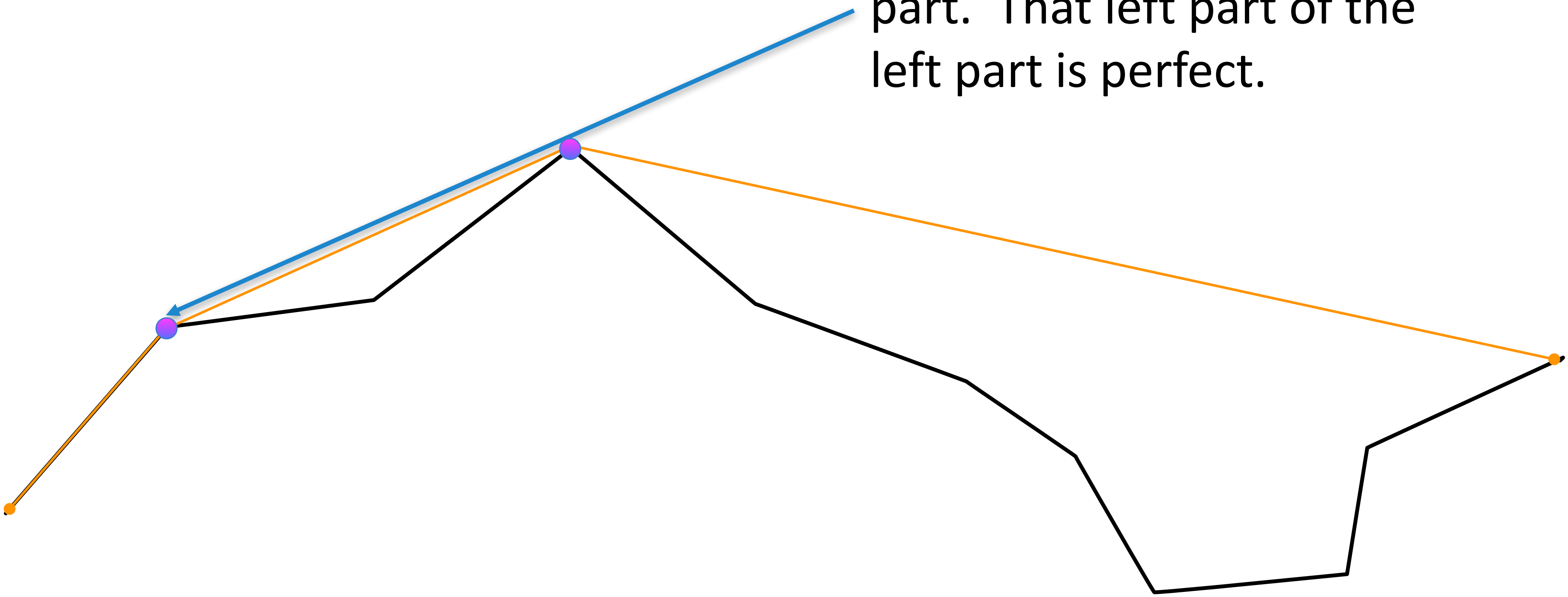
Nah, that’s pretty bad,
and this point is the worst.



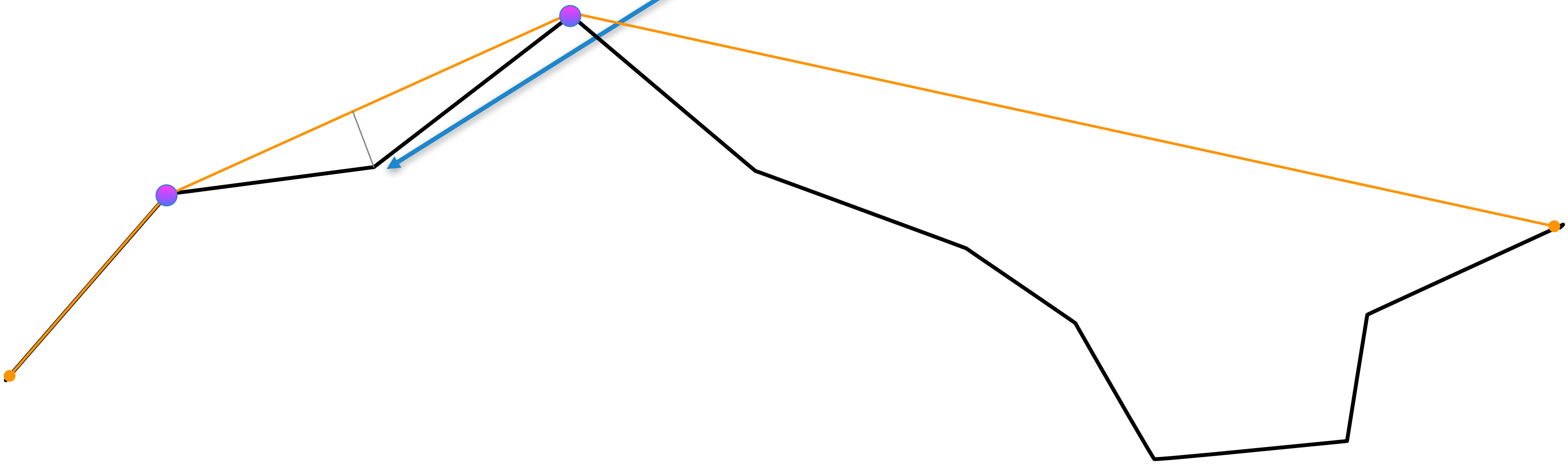
OK then, I'll break it at the bad point. How is the left part?



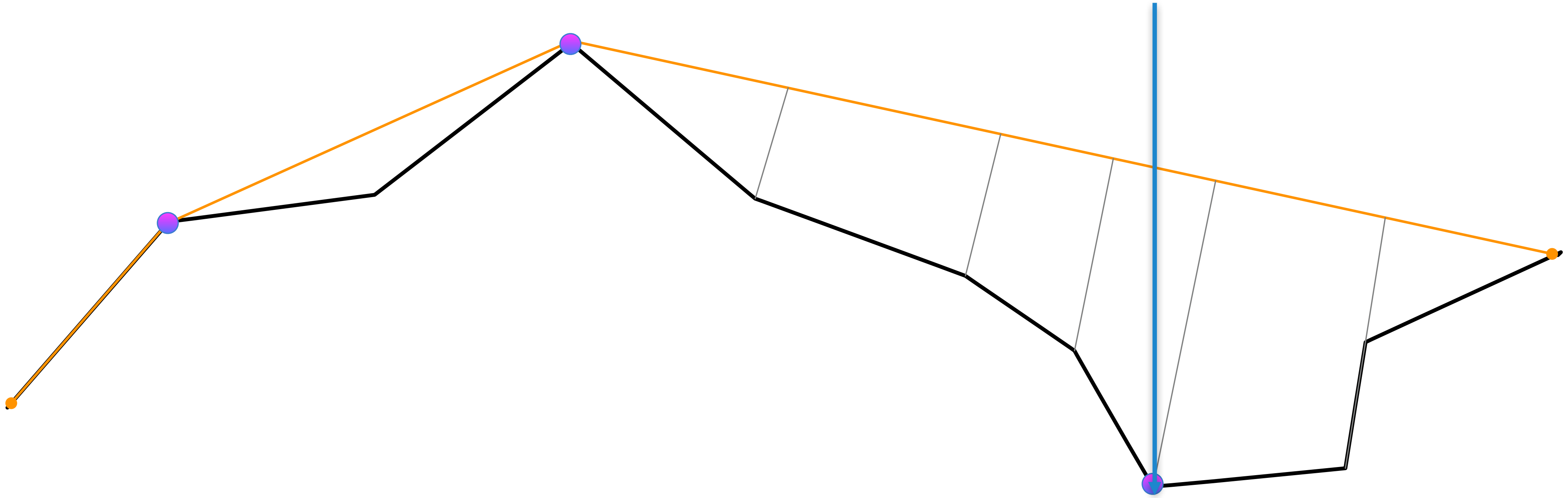
Still not great. That one is too far.



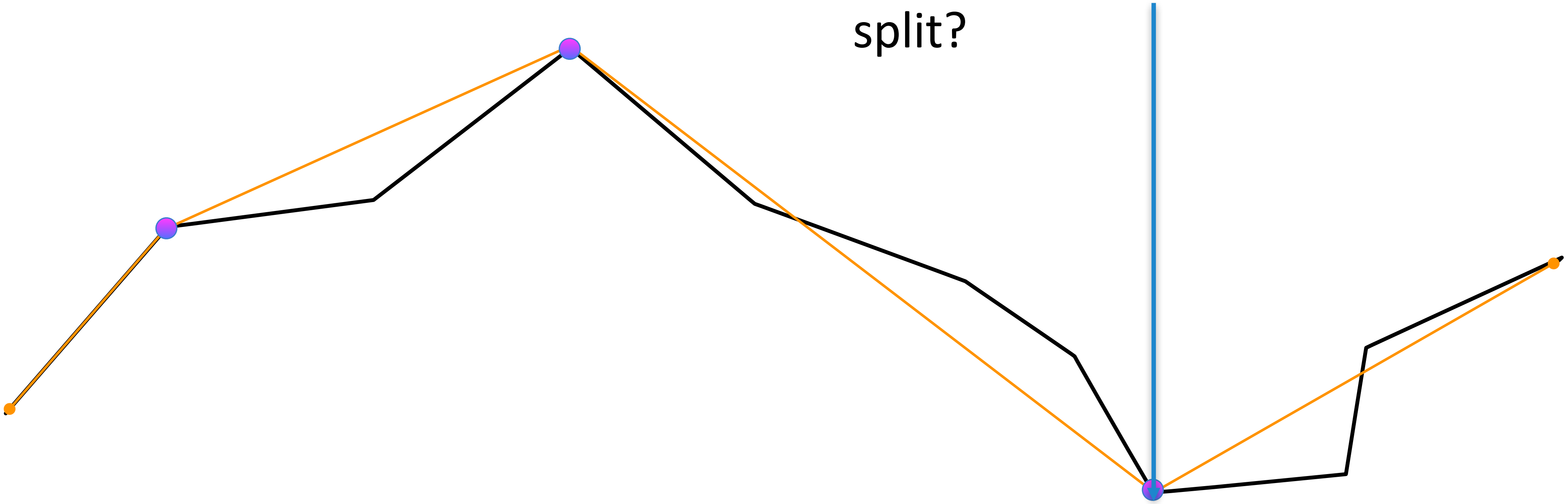
OK, I broke it at the worst part. That left part of the left part is perfect.



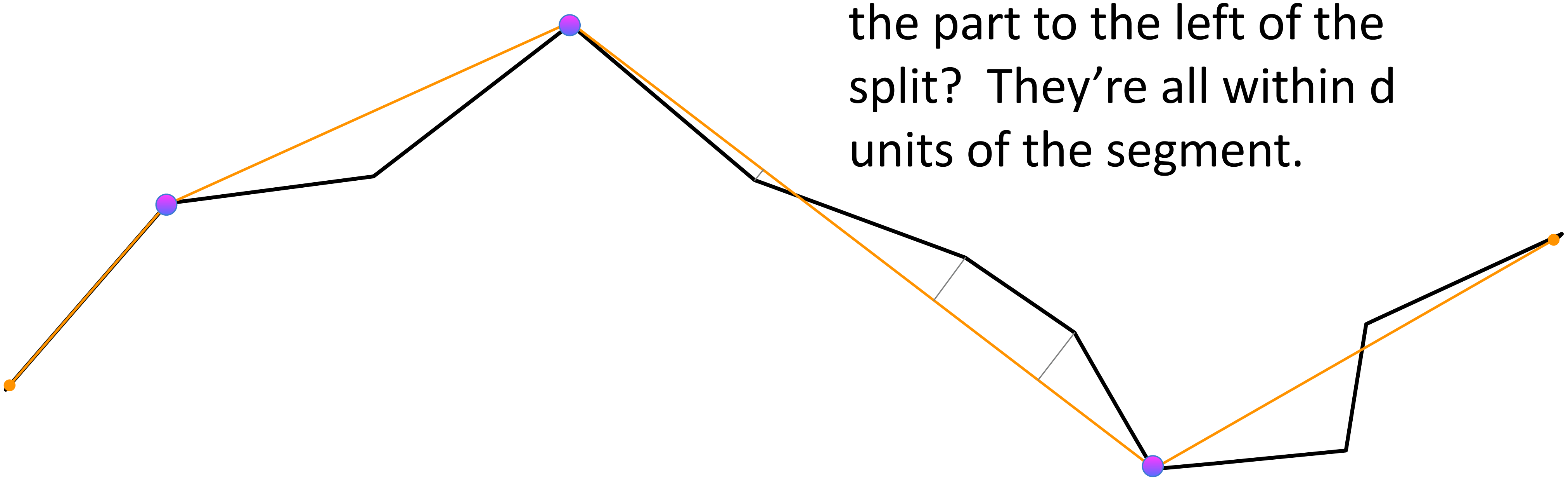
The right part of the left part is not perfect, but it’s close enough.



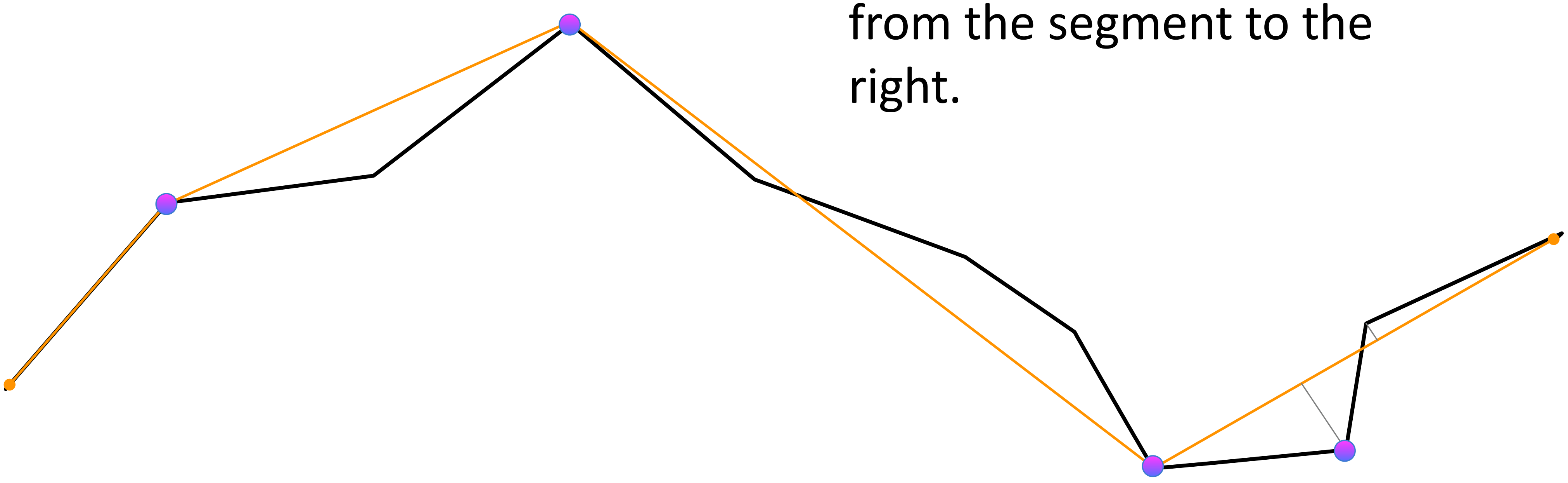
The right part is pretty bad, especially this point.



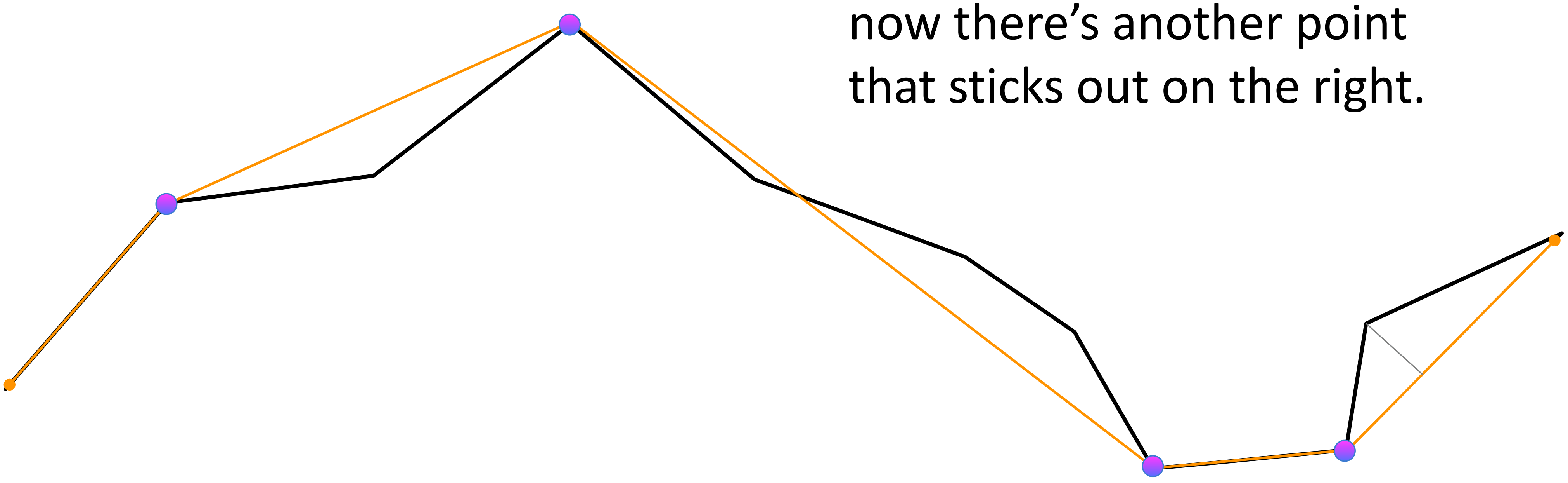
OK, I'll split it there. How's the part to the left of the split?



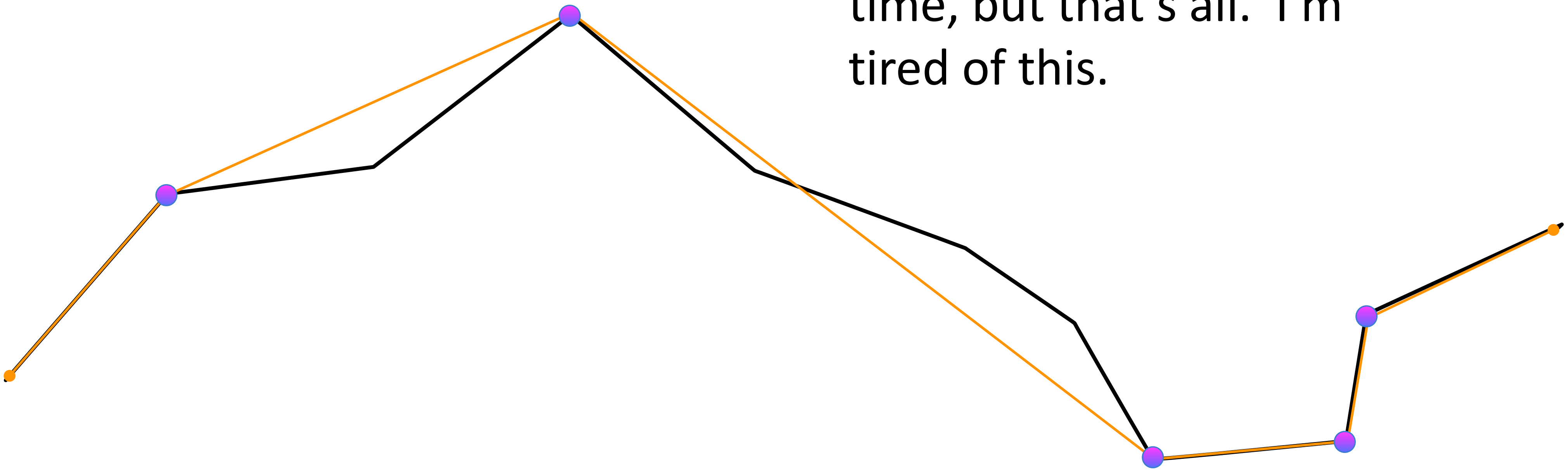
OK, I’ll split it there. How’s the part to the left of the split? They’re all within d units of the segment.



One point is farther than d from the segment to the right.



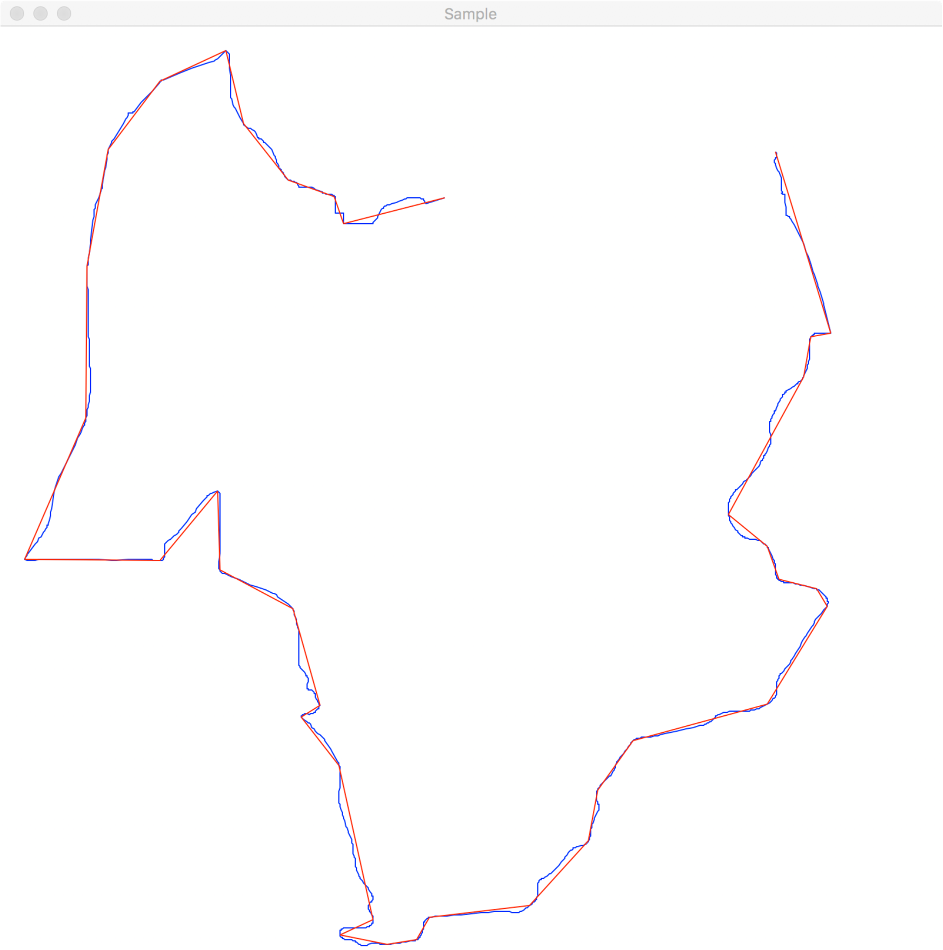
So I'll split it there. But now there's another point that sticks out on the right.



So I’ll split it one more time, but that’s all. I’m tired of this.

Let’s do it for real …

```
python3 plot_path.py data/FoxHollow.csv 800 800 100
```



geometry.py…

```
class PolyLine:
    """A polyline is a sequence of points, represented  as (x,y) tuples. …
    PolyLine.points and PolyLine.approx are public, read-only
    attributes.  Other attributes are private.
    """

    def __init__(self, points):
        self.points = points
        self.approx = points
        self.tolerance = 0

    def simplify(self, tolerance):
        """Approximate the polyline …
        ""”
        …
```

```
class PolyLine:
    ...
    def simplify(self, tolerance):
        """Approximate the polyline, within a maximum deviation """
        self.tolerance = tolerance
        self.approx = [ ]
        self._dp_simplify(0, len(self.points)-1)
        self.approx.append(self.points[-1])
        return

    def _dp_simplify(self, from_index, to_index):
        """Recursively build up simplified path, working left to
        right to add the resulting points to the simplified list.
         """
```

```
class PolyLine:
  ...

  def _dp_simplify(self, from_index, to_index):
    """Recursively build up simplified path, working left to
    right to add the resulting points to the simplified list.
     """
```

What are the base cases for _dp_simplify(from,to)?
What is the inductive case?

Base case: Just two points, nothing to subdivide.

```
def _dp_simplify(self, from_index, to_index):
    ...
    if to_index - from_index < 2:
```

More than two points … are they close enough?

```
def _dp_simplify(self, from_index, to_index):
    …
    for i in range(from_index+1, to_index):
        dev = deviation(seg_start, seg_end, self.points[i])
```

Determine the deviation (perpendicular distance) of each point.
Record the maximum deviation and which point is at maximum deviation.

('deviation' function is given.)

Base case: Already close enough. Just keep the starting point.

```
if max_deviation > self.tolerance:
     …
else:
   # Already good enough
   self.approx.append(self.points[from_index])
```

Inductive case: Split at point with maximum deviation, recursively call on left and then right part.

```
if max_deviation > self.tolerance:
    # Too much deviation.  Subdivide
    self._dp_simplify(from_index, max_index)
    self._dp_simplify(max_index, to_index)

else:
     ...
```

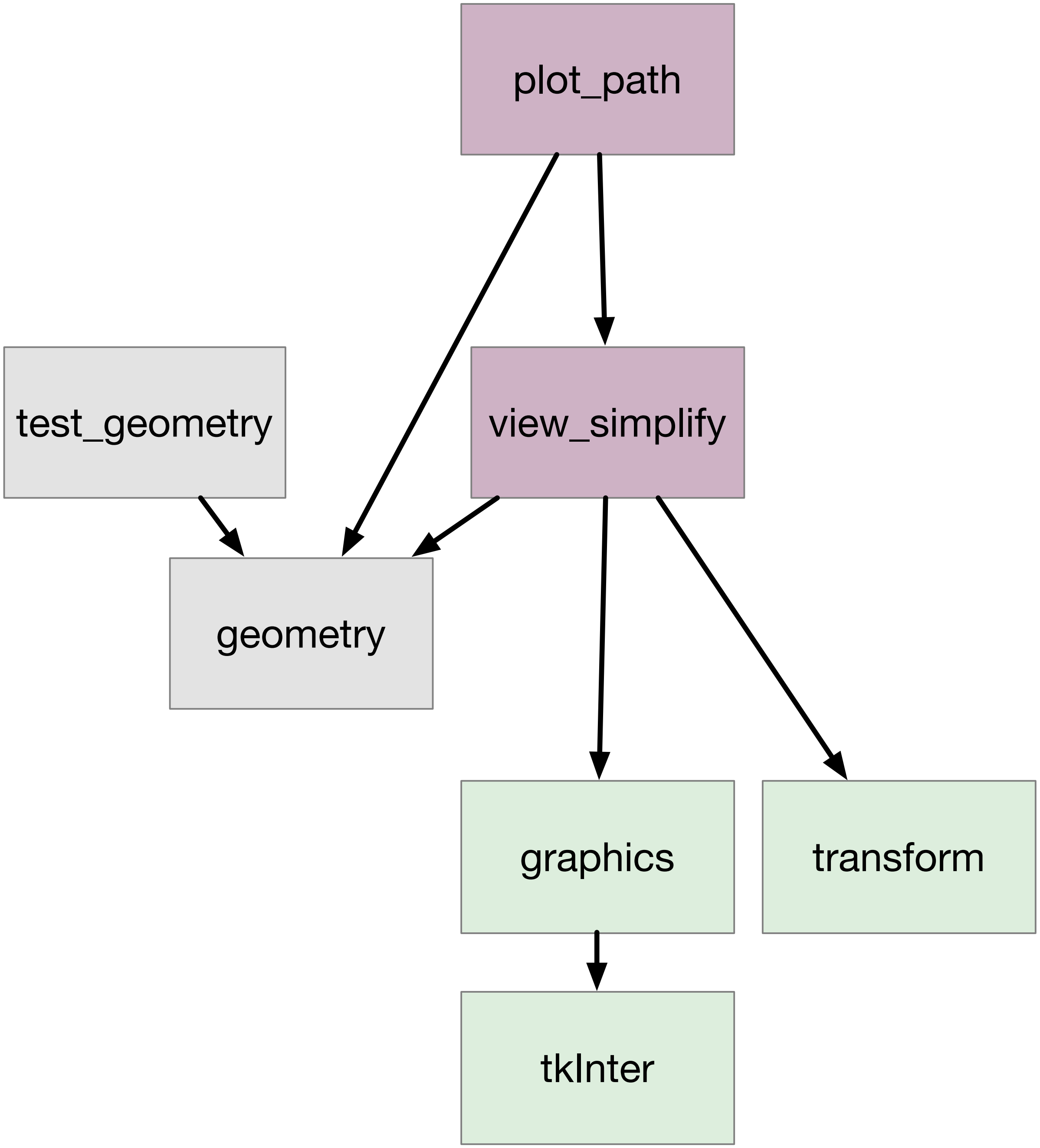
Hey, but what about the display?

We want graphics, but we don't want to mix the graphics code into geometry.py.

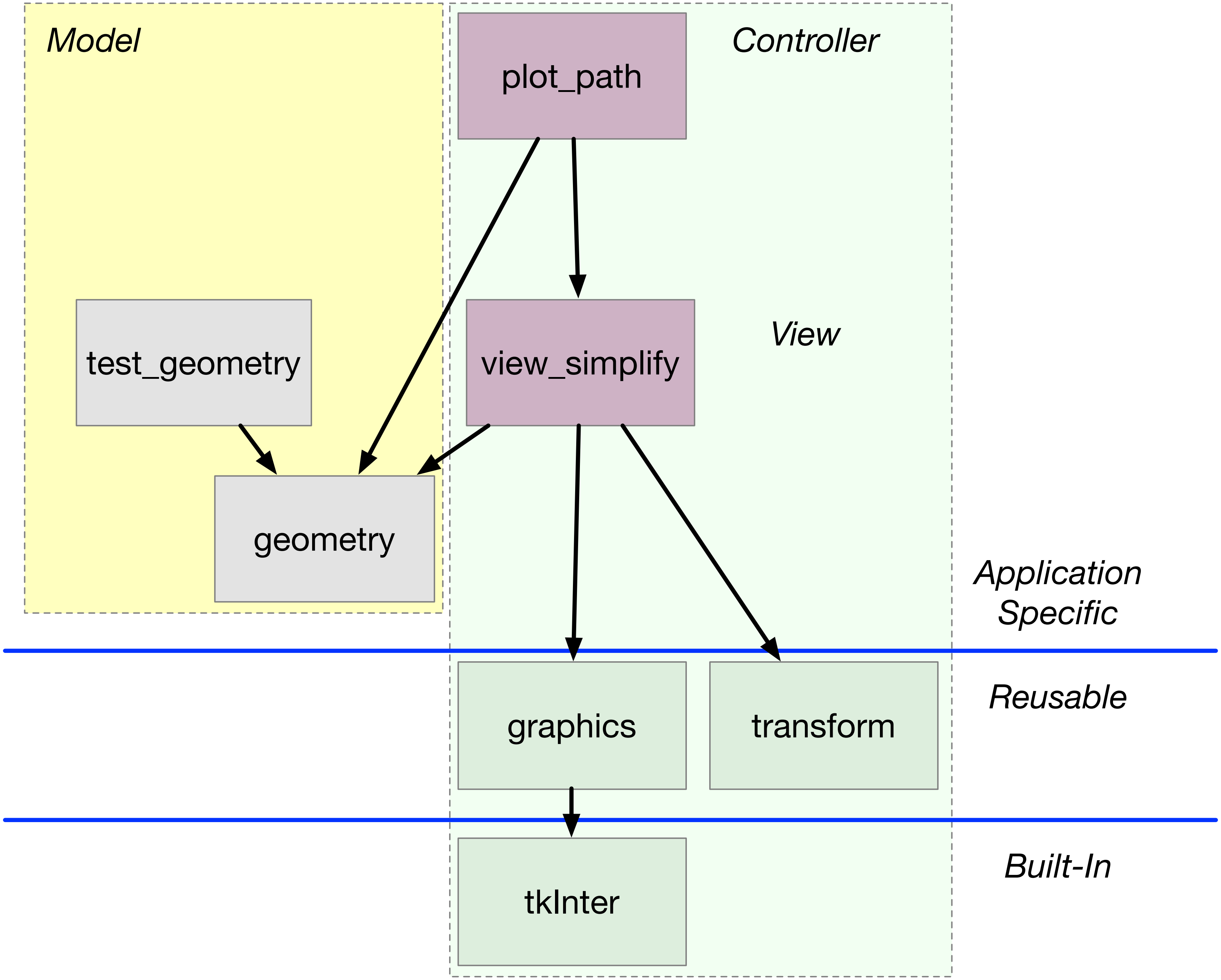
Modularity demands separation of concerns: Geometry algorithms in geometry.py, graphical display in a different module.

We can do this by following a *design pattern* called Model-View-Controller.

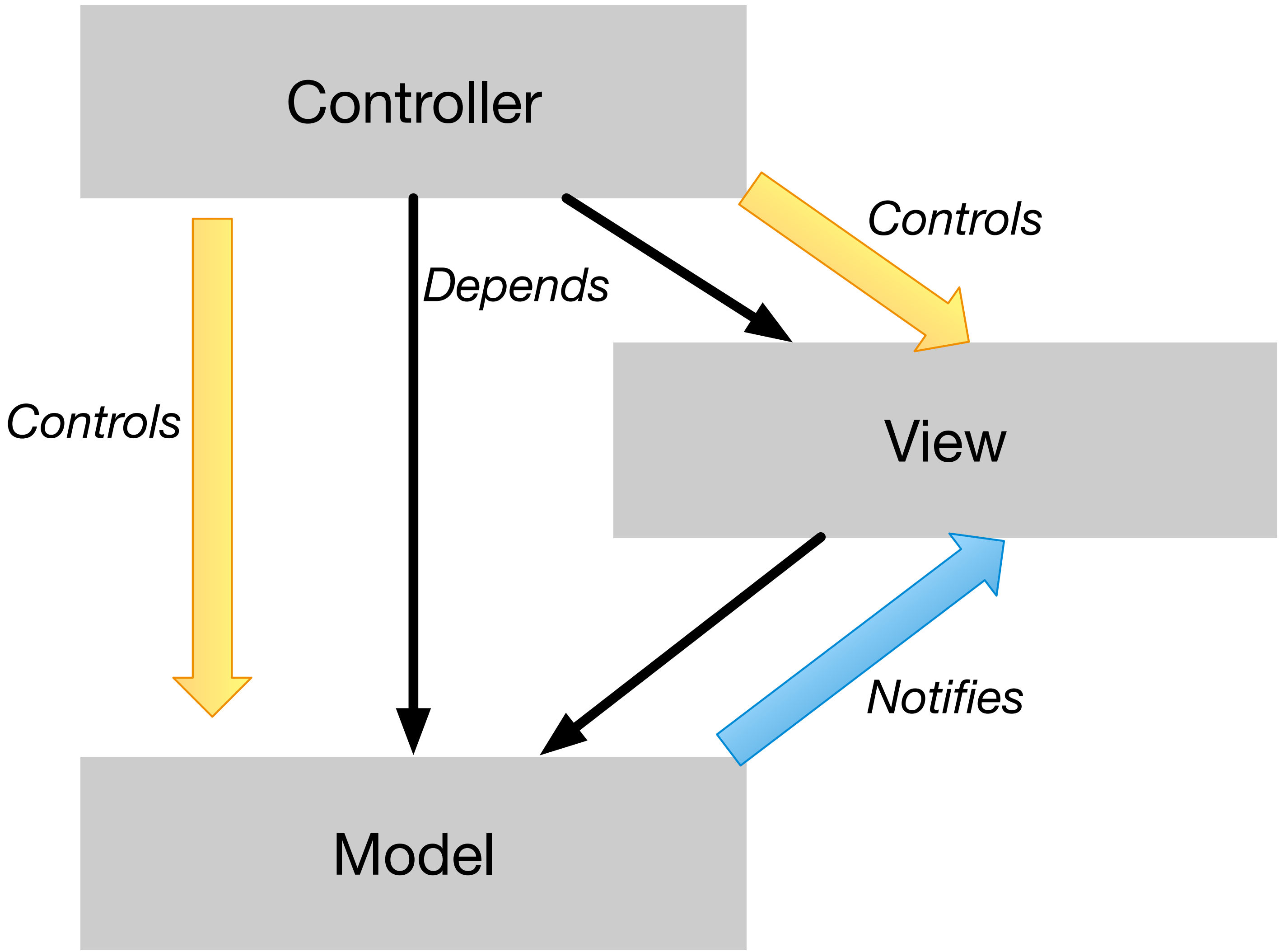
Dependence


plot_path
test_geometry
view_simplify
geometry
graphics
transform
tkInter

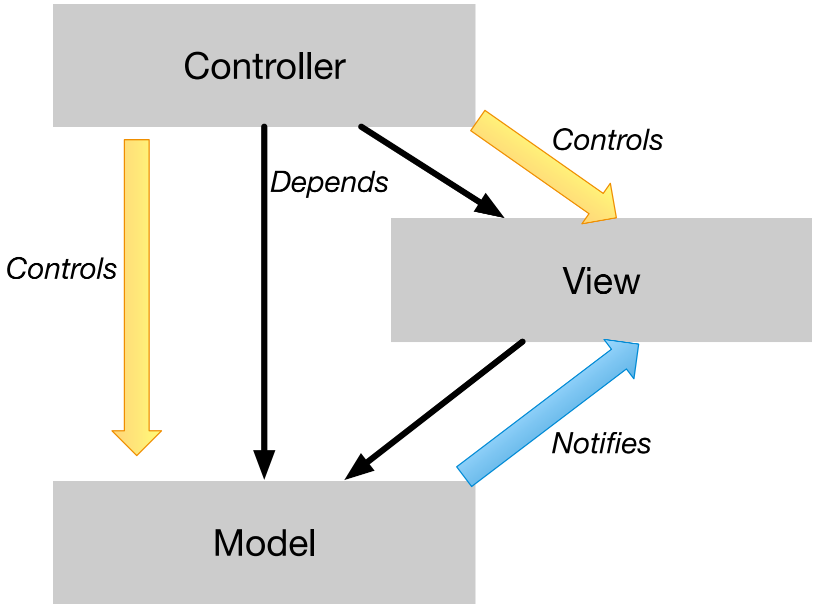
Separation of Concerns

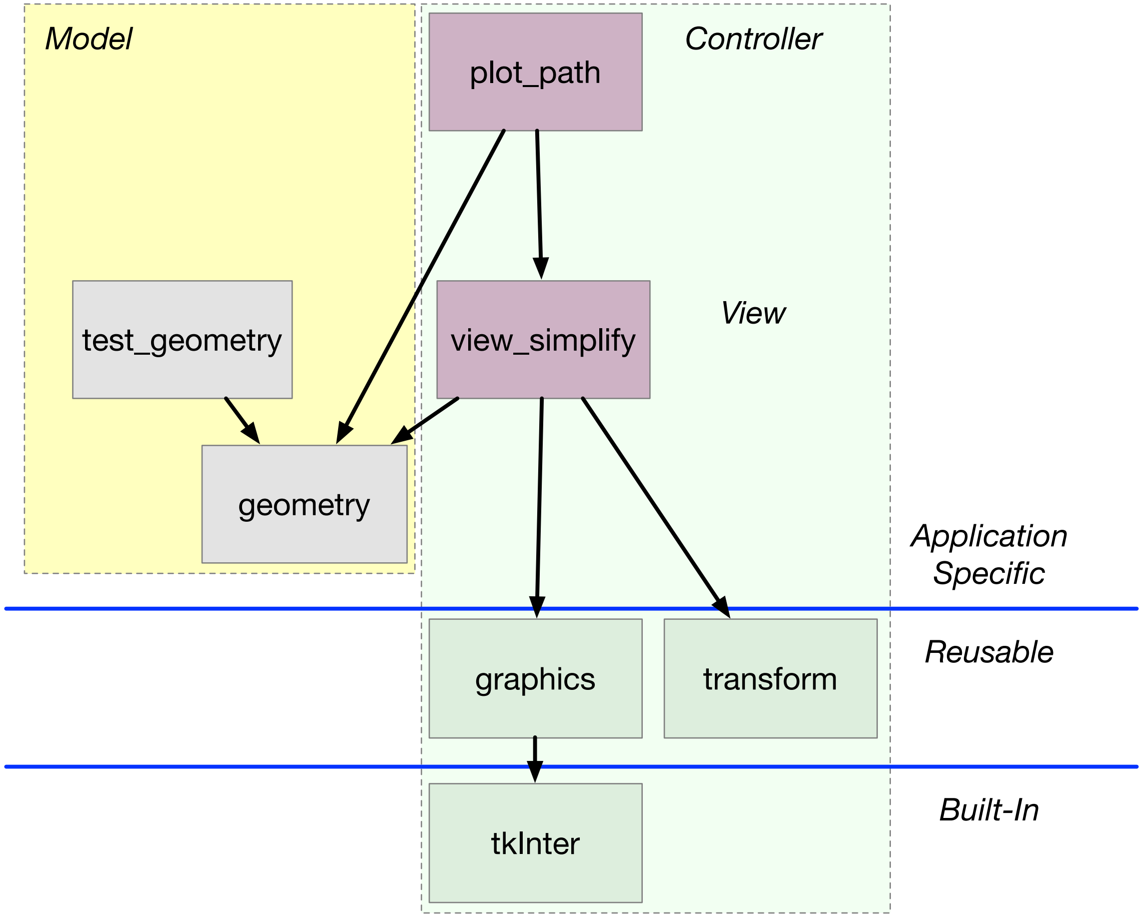


Model-View-Controller pattern



Our use of MVC pattern





Controller/driver attaches view to model

plot_path.py

```
import view_simplify
import geometry
…
path = geometry.PolyLine(pts)
view = view_simplify.View(win, path.points)
path.add_listener(view)
…
```

geometry.py

```
class PolyLine:
  …
  def add_listener(self, listener):
       self.listeners.append(listener)
```

Model sends events to listening view(s)

```
class PolyLine:


…
def notify_all(self, event_name, options={}):
    for listener in self.listeners:
        listener.notify(event_name, options=options)


…


def _dp_simplify(self, from_index, to_index):
    self.notify_all("trial_approx",
                    options = { "p1": self.points[from_index],
                                "p2": self.points[to_index] })
   …
```

View receives notification, updates display

```
class View(object):
    """A view of line simplification"""

    def notify(self, event_name, options):
        """Event notifications from simplification process."""
        if event_name == "trial_approx":
            p1 = options["p1"]
            p2 = options["p2"]
            self.draw_segment(p1, p2, color="grey")
        elif event_name == "final_approx_seg":
            p1 = options["p1"]
            p2 = options["p2"]
            self.draw_segment(p1, p2, color="red")
        else:
            raise Exception("Unknown event {}".format(event_name))
```

Why all this trouble for separation of concerns?

Incremental development: Work on one part at a time

Communication (to other developers): Well-known, standard pattern

Cost-effective change: Isolate what the developer must read, understand, and change

Note: *Cost of a software change is proportional not to how much the developer must write (often very little), but to how much the developer must read*