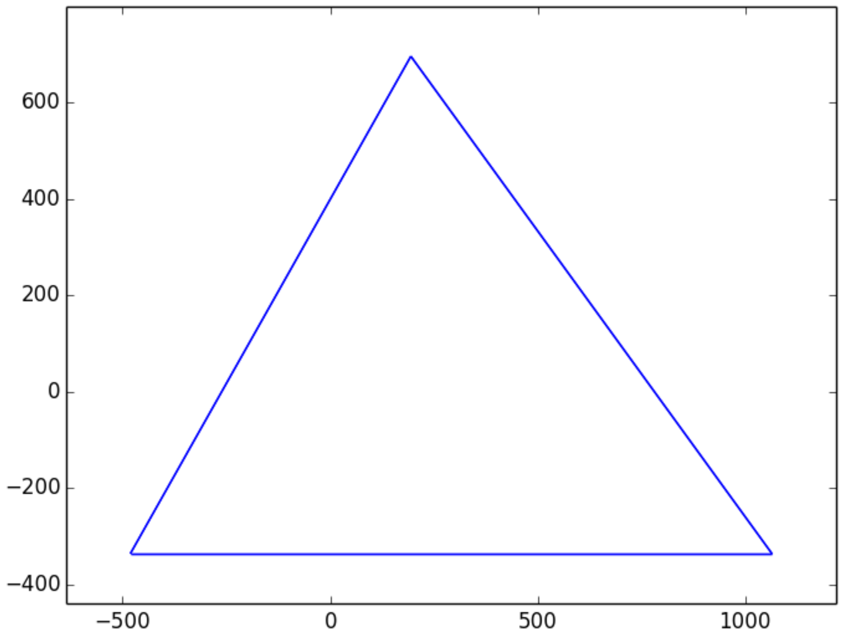
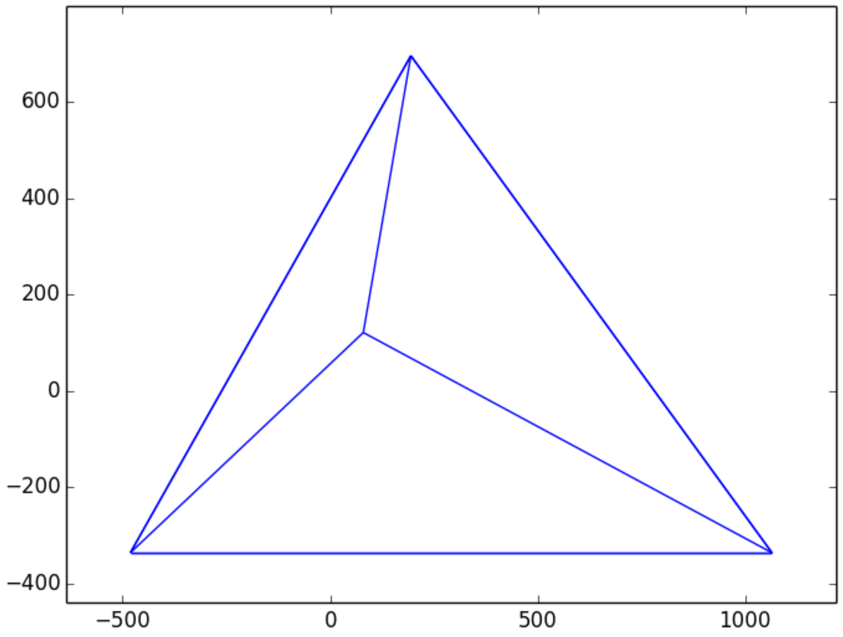
[Example 1]

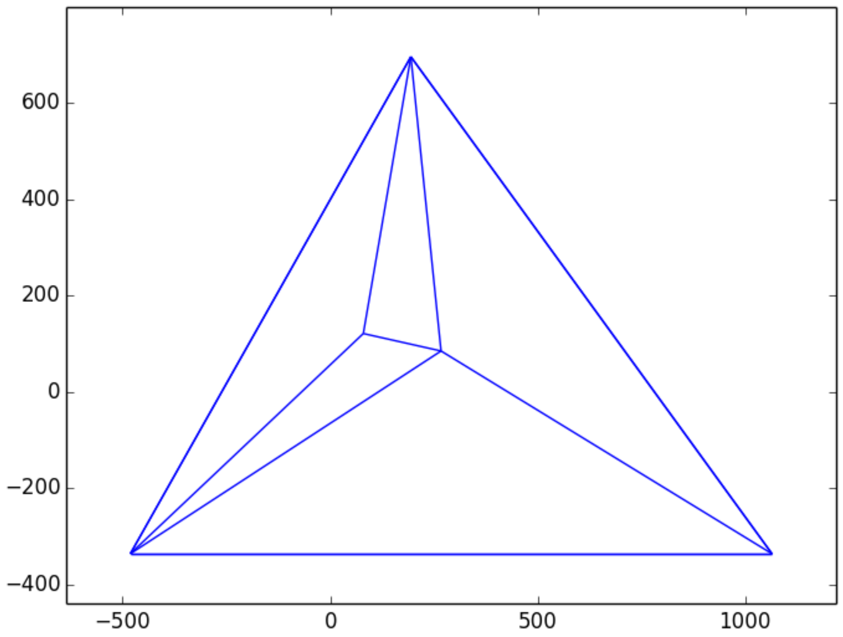
Input points:

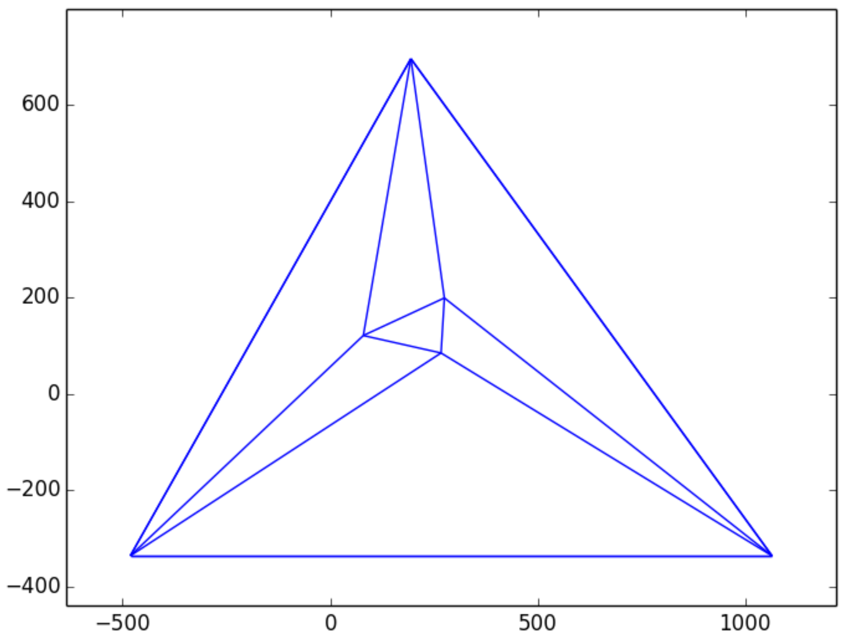
0: [233.67, 159.0]
1: [67.33, 179.47]
2: [89.0, 350.47]
3: [153.83, 217.97]
4: [266.67, 315.47]
5: [364.67, 185.0]
6: [274.67, 198.76]
7: [266.67, 84.53]
8: [145.0, 108.53]
9: [23.33, 8.53]
10: [79.5, 120.76]

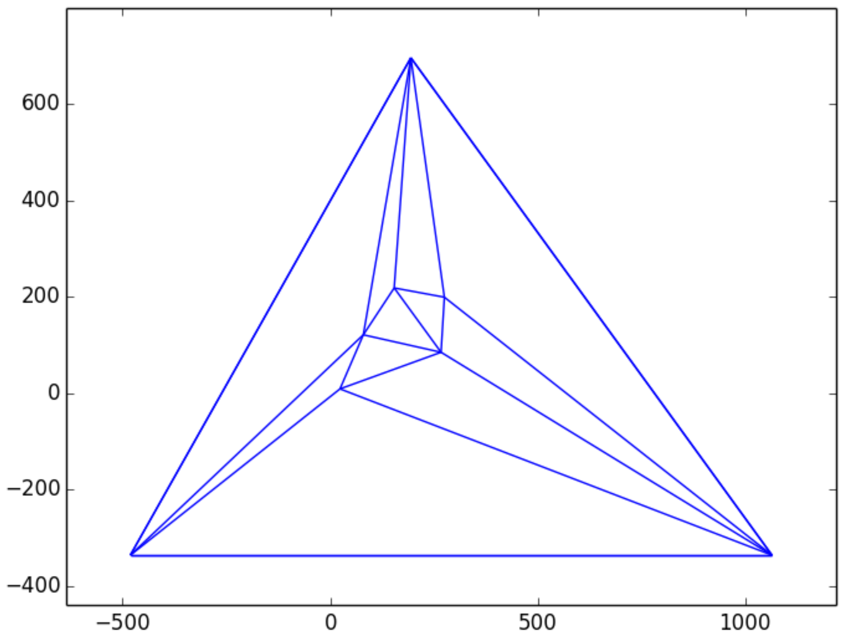
The next 7 images show the coarse-to-fine subdivisions and triangulations.

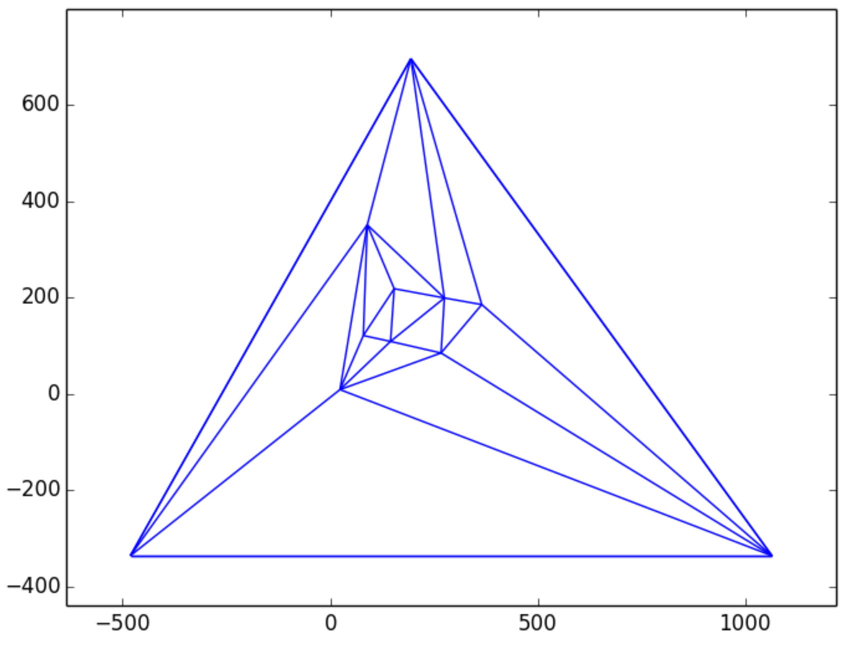


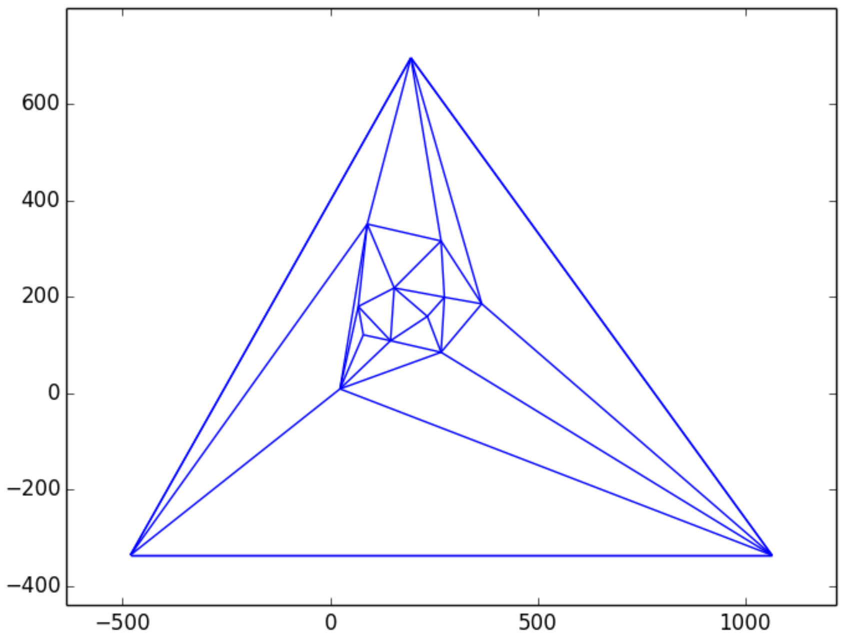

600
400
200
0
−200
−400
−500
0
500
1000


600
400
200
0
−200
−400
−500
0
500
1000


600
400
200
0
−200
−400
−500
0
500
1000

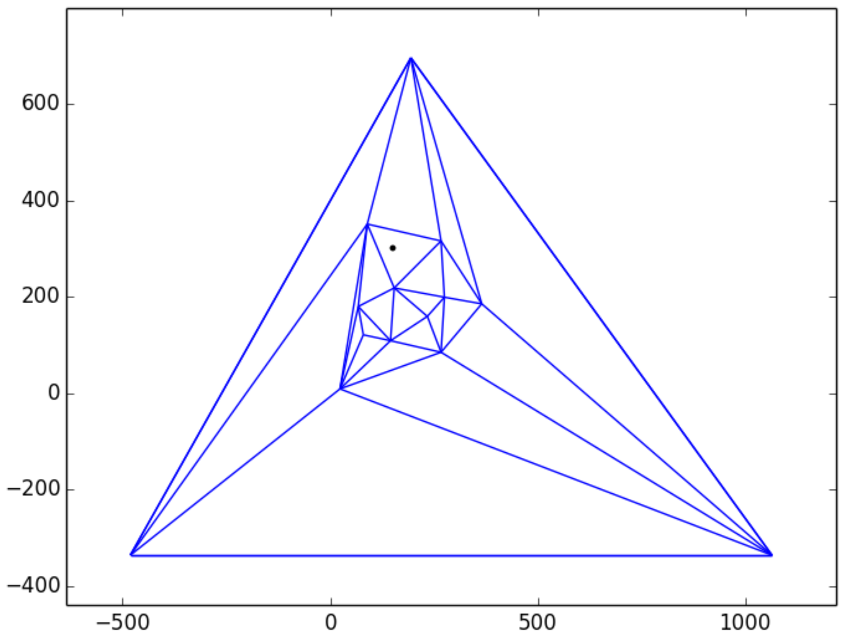

600
400
200
0
−200
−400
−500
0
500
1000



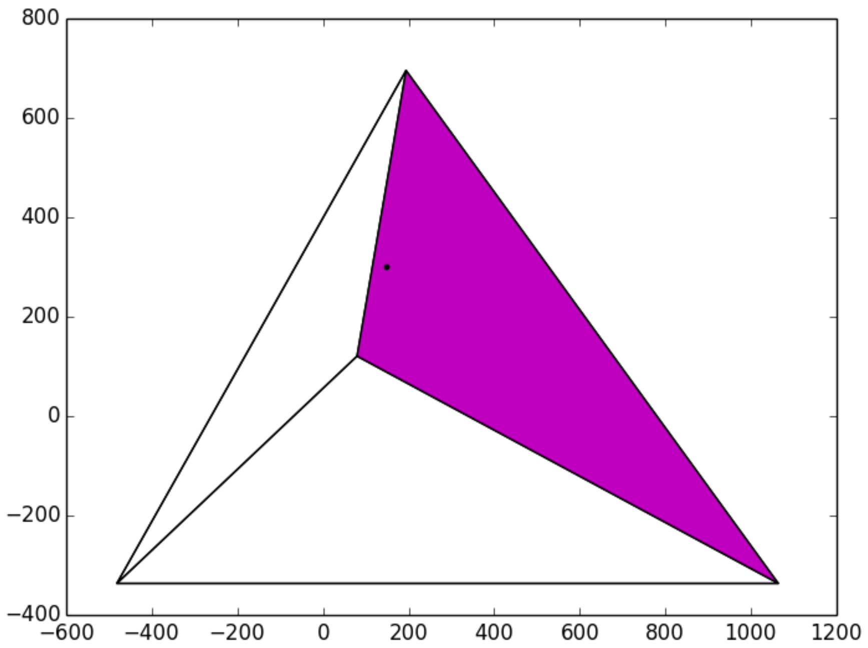


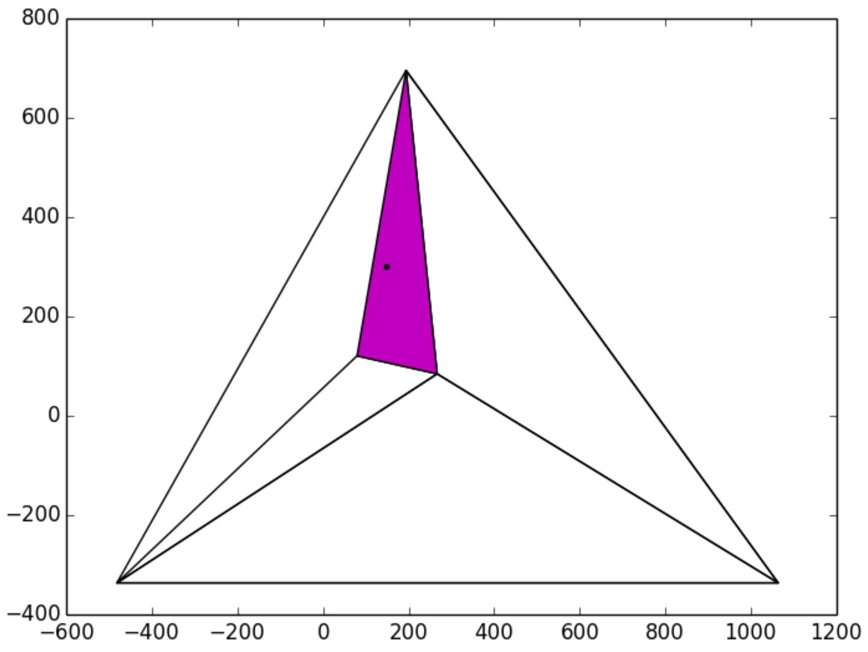
We also illustrate a point location query on the input points.
Query = (150, 300)

Query point overlaid on the fine subdivision to show which triangle should eventually be found:

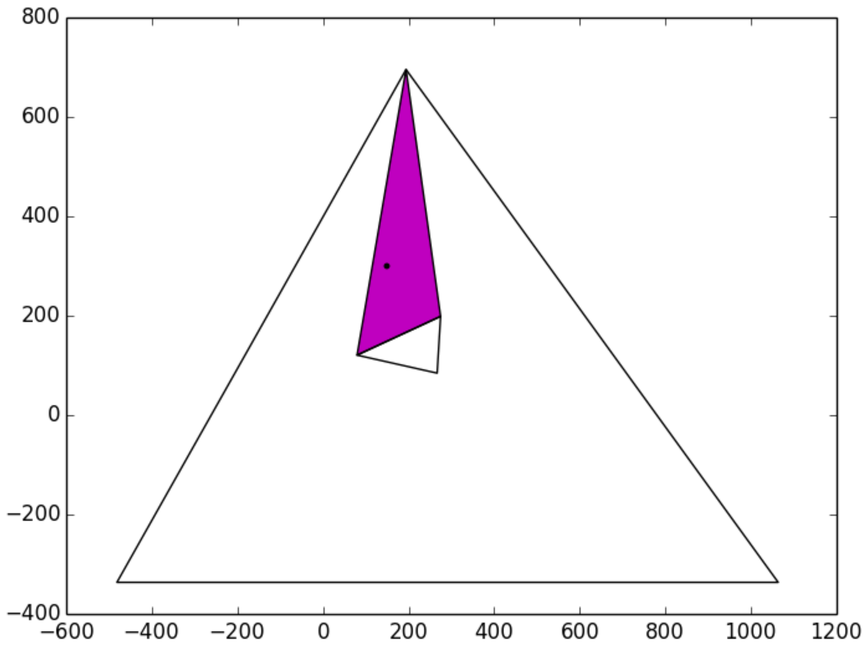


The next 6 images illustrate the point search as it starts from the coarse division and enters finer and finer divisions using overlapping child triangles.

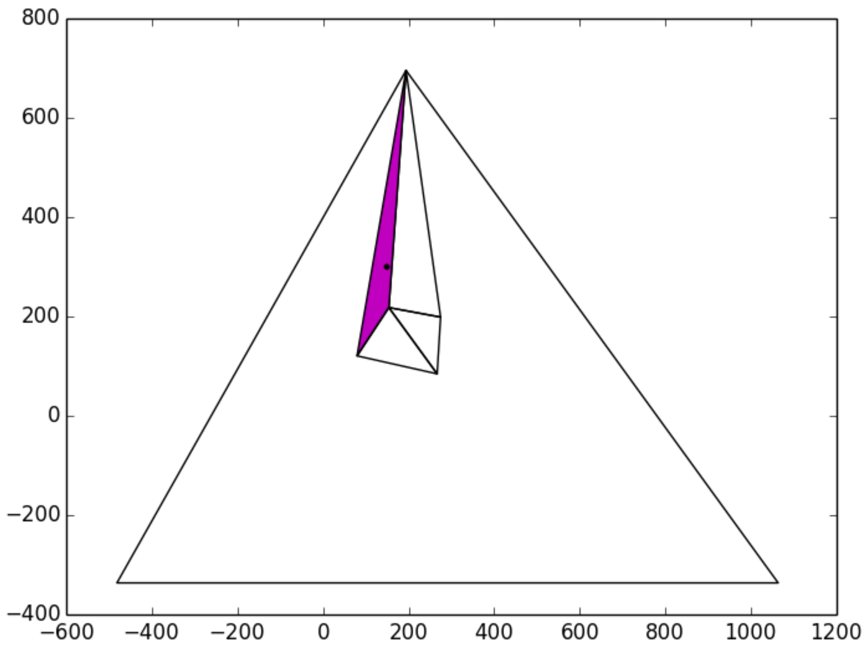


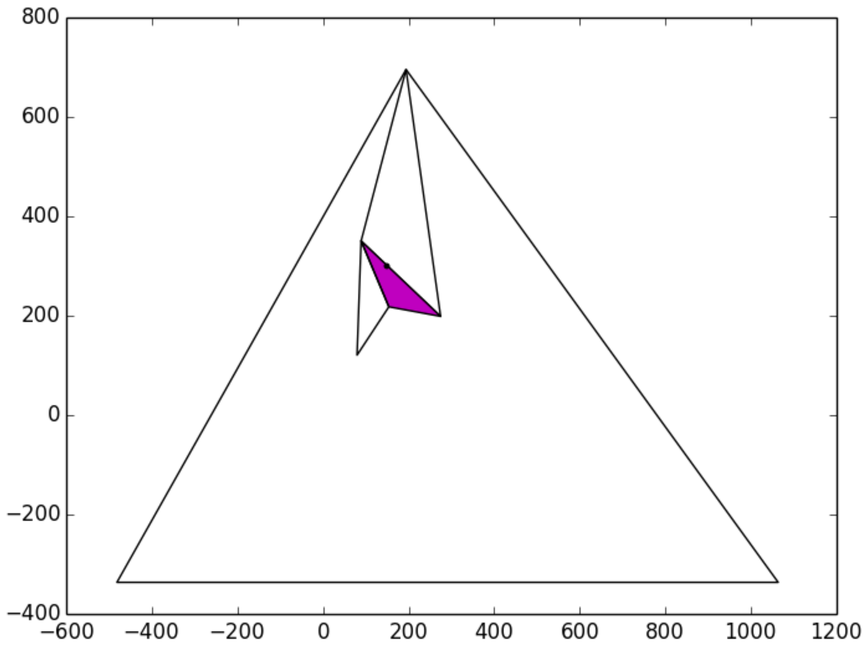


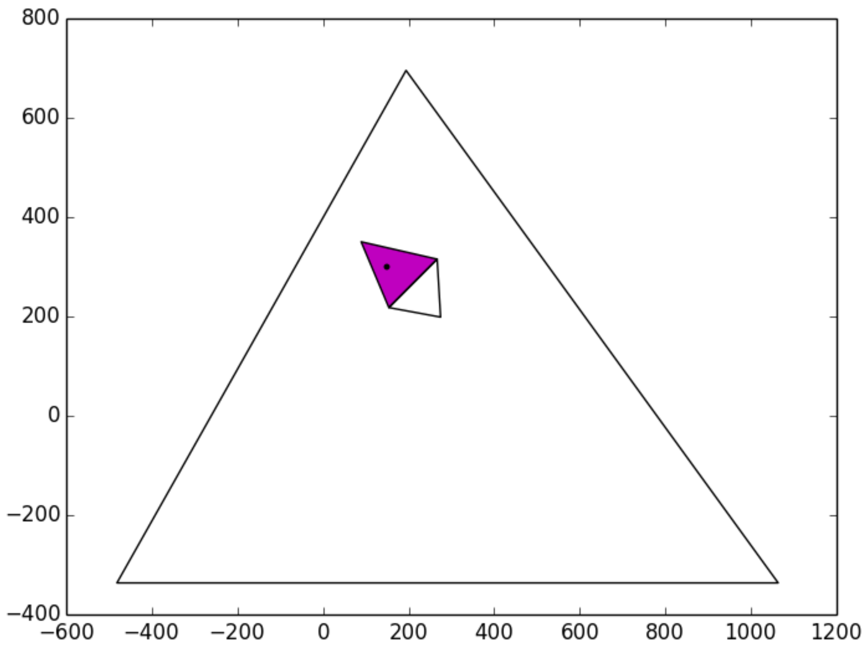
800
600
400
200
0
−200
−400
−600
−400
−200
0
200
400
600
800
1000
1200



800
600
400
200
0
−200
−400
−600
−400
−200
0
200
400
600
800
1000
1200


800
600
400
200
0
−200
−400
−600
−400
−200
0
200
400
600
800
1000
1200


800
600
400
200
0
−200
−400
−600
−400
−200
0
200
400
600
800
1000
1200



The final output of our algorithm, showing the coordinates of the final triangle and whether it was inside the convex hull of the input points:

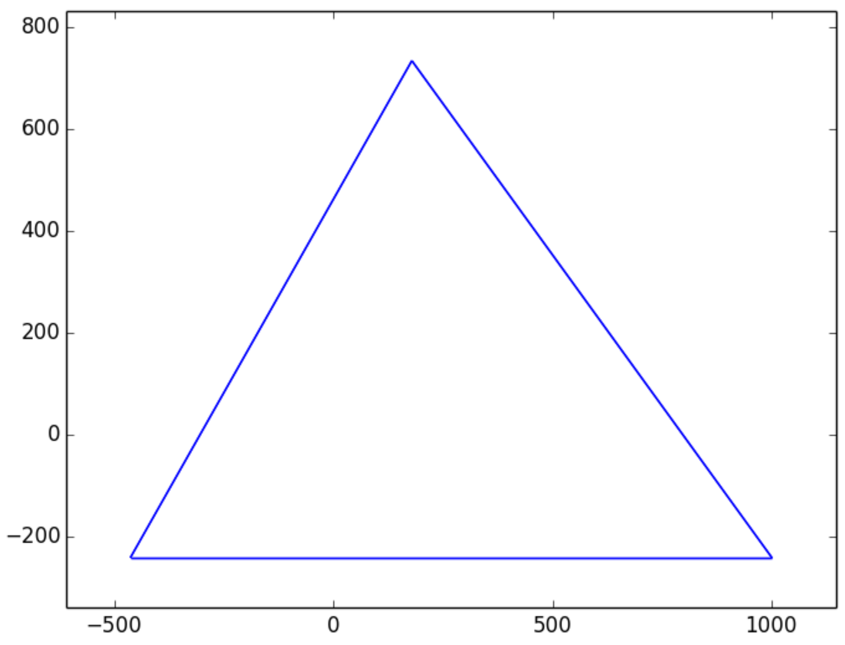
```
{'p2': [46.0, 725.0], 'p3': [95.0, 351.0], 'inside': True, 'p1':
[24.0, 85.0]}
```

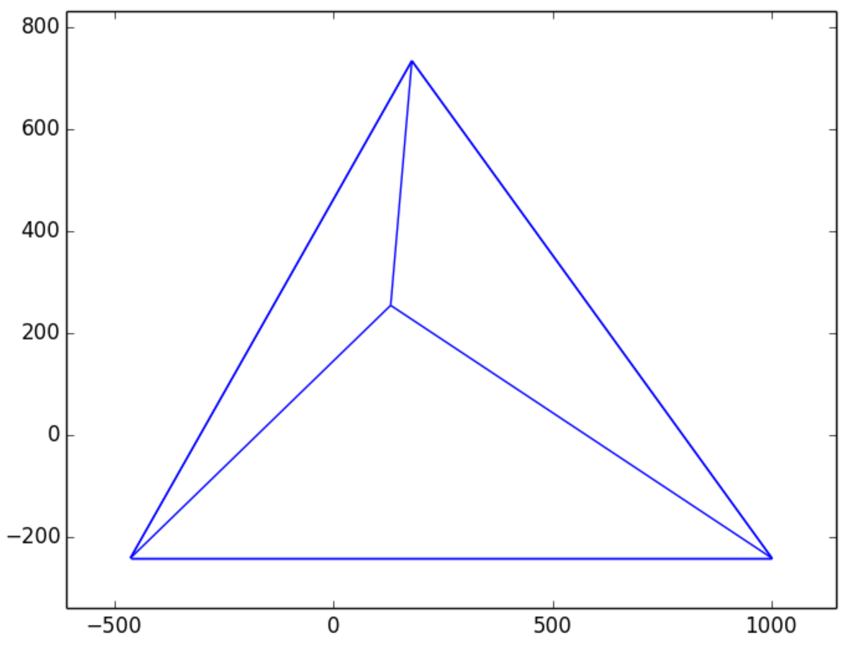
[Example 2]

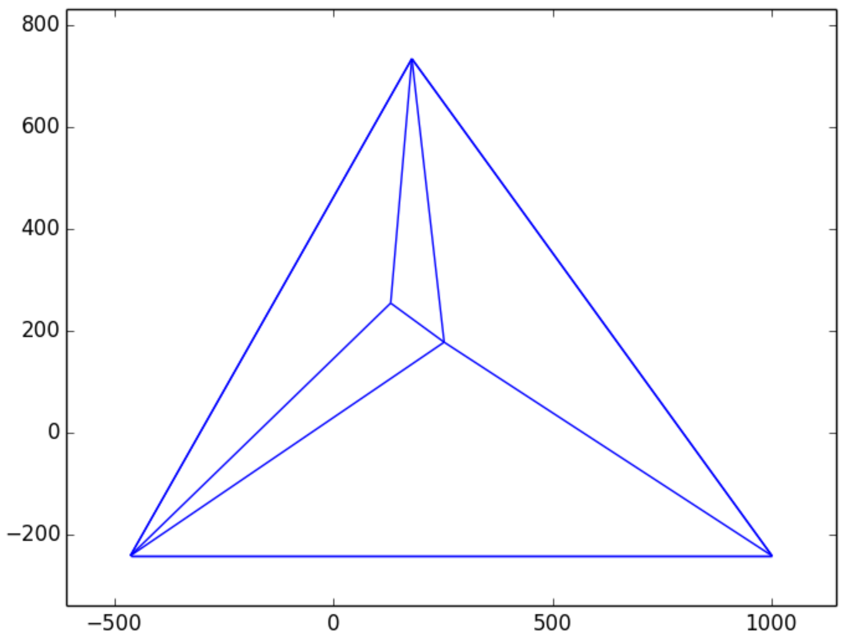
Input points:

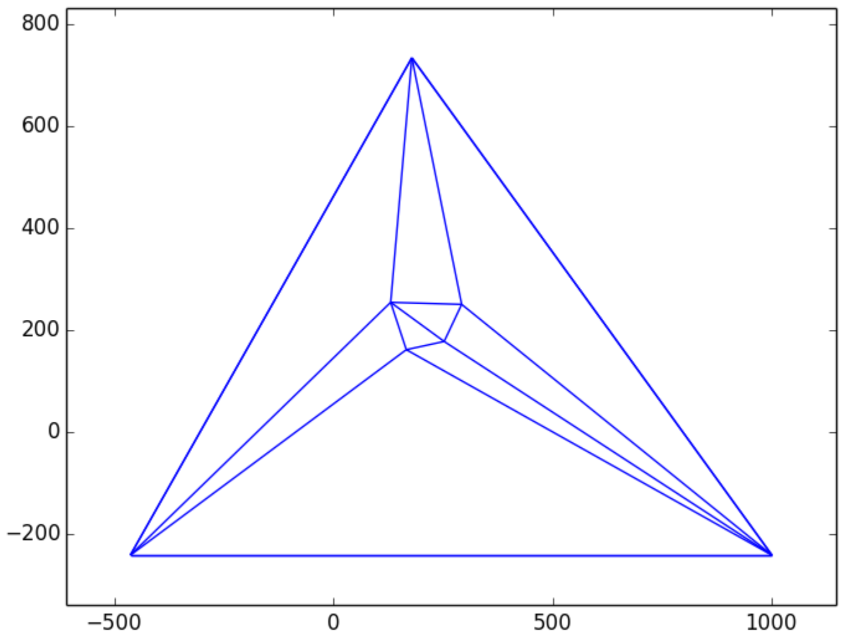
0: [17.67, 220.0]
1: [133.33, 315.47]
2: [194.0, 178.47]
3: [266.67, 315.47]
4: [341.0, 407.74]
5: [333.33, 200.0]
6: [293.0, 250.26]
7: [266.67, 84.53]
8: [253.0, 177.53]
9: [133.33, 84.53]
10: [166.67, 161.4]
11: [131.0, 254.26]

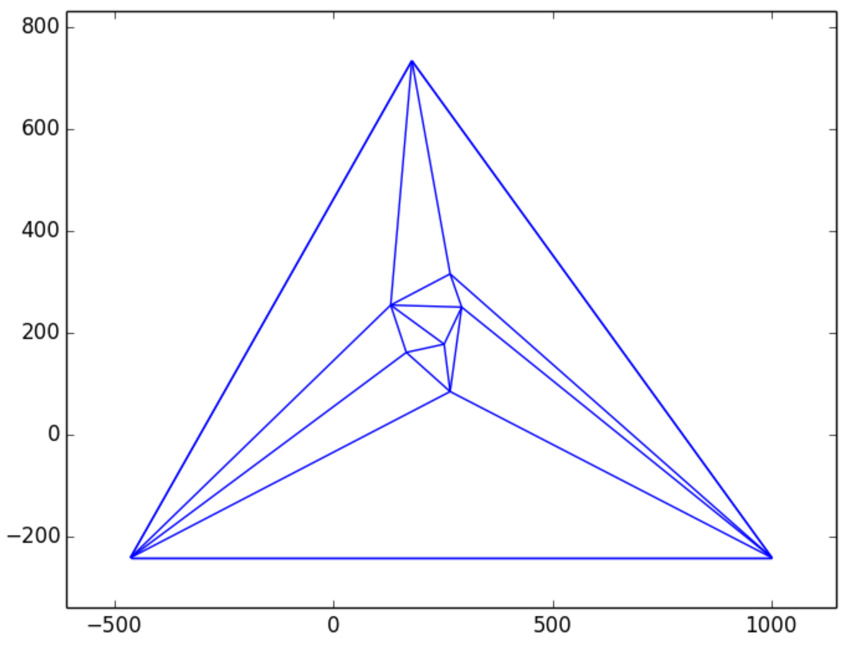
The next 7 images show the coarse-to-fine subdivisions and triangulations.

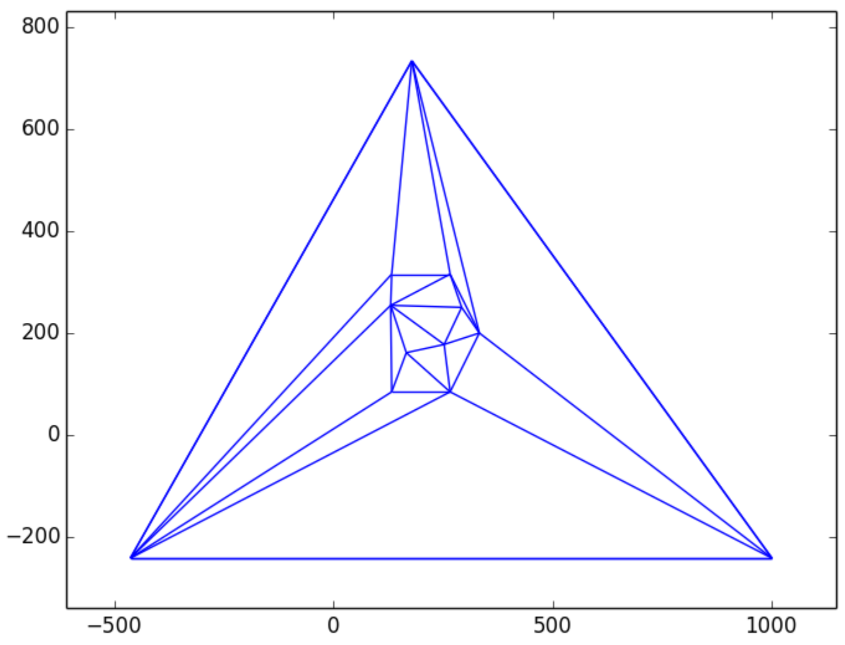

800
600
400
200
0
−200
−500
0
500
1000

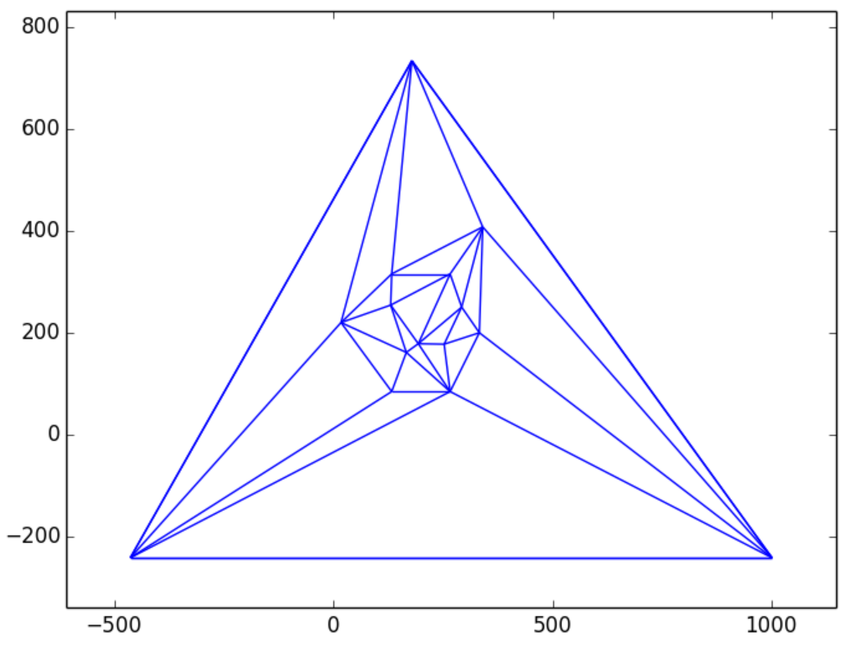

800
600
400
200
0
−200
−500
0
500
1000


800
600
400
200
0
−200
−500
0
500
1000


800
600
400
200
0
−200
−500
0
500
1000

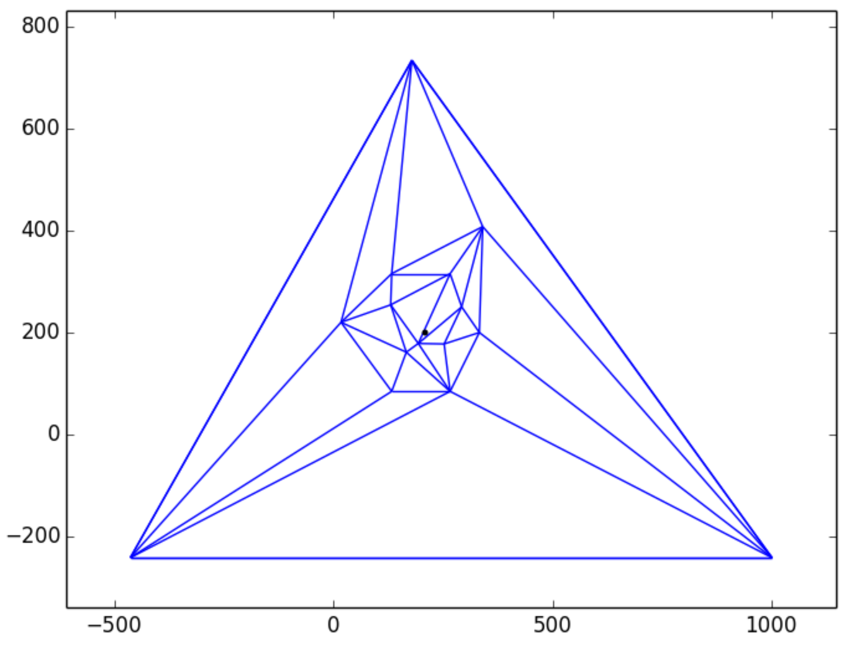

800
600
400
200
0
−200
−500
0
500
1000


800
600
400
200
0
−200
−500
0
500
1000

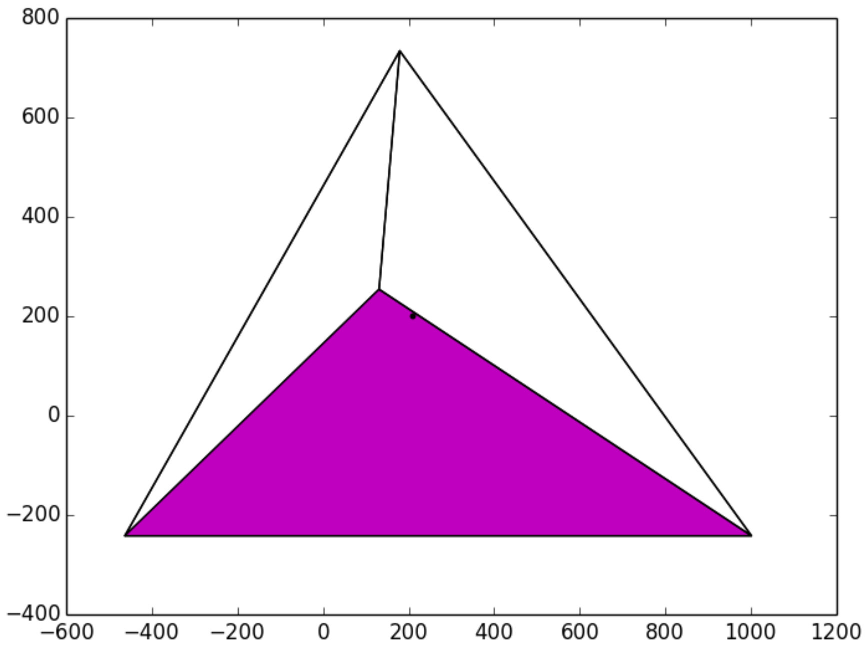


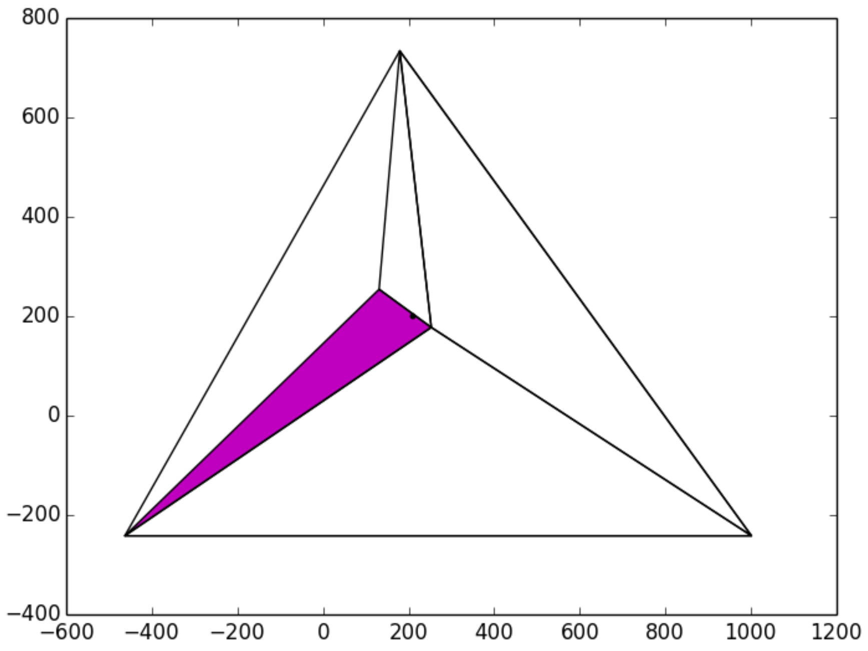
We also illustrate a point location query on the input points.
Query = (210, 200)

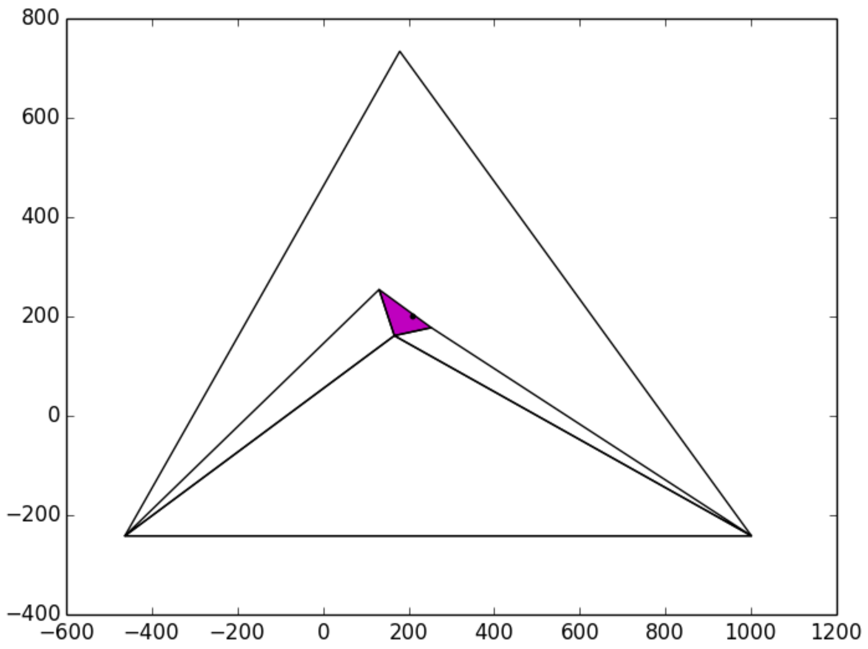
Query point overlaid on the fine subdivision to show which triangle should eventually be found:

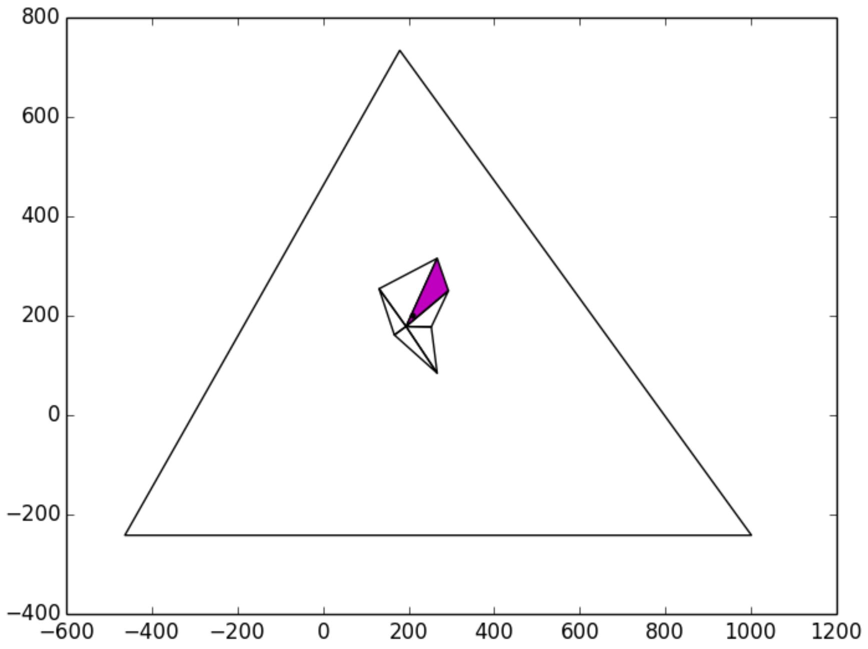


The next 4 images illustrate the point search as it starts from the coarse division and enters finer and finer divisions using overlapping child triangles.









The final output of our algorithm, showing the coordinates of the final triangle and whether it was inside the convex hull of the input points:

```
{'p2': [266.67, 315.47], 'p3': [293.0, 250.26], 'inside': True,
                    'p1': [194.0, 178.47]}
```

Timing Benchmarks

Finally, we provide some timing benchmarks for DAG data-structure construction and point location. To generate the benchmarks, we generate N random points (x and y coordinates between 0 and 100,000) for the input points into the data structure. For location times, we generate a random point q after generating N random points (for each time, we use a new set of N points).

Data-structure construction:

N (# points)	T(# seconds)
100	0.317
250	1.15
500	3.508
1000	11.95
2000	46.10

Point Location for random query point on N random input points

N (# points)	T(# seconds)
1000	0.000129
2000	0.000172
3000	0.000154
5000	0.000199