



basically there is an update u have to implement that is

while the points are getting generated and if the next point is not following the path then we have to set a threshold and consider the max of 3 peaks, from them consider the point which is nearer

explain this to me and also tell me what point refers to here

Absolutely, here's a breakdown of your update and what "point" refers to:

What you're describing

- When performing template matching (i.e., searching for a small template image inside a larger image), **multiple candidate locations or "+points+"** are found based on the similarity/match score.
- Sometimes, the next point (candidate location) does **not follow the expected path** — it can jump off the trajectory, e.g., due to noise, repeating textures, or ambiguity in the local image. [\[1\]](#) [\[2\]](#) [\[3\]](#) [\[4\]](#)
- To handle this, instead of always picking the absolute highest peak (best match score), you should:
 1. **Identify the top 3 peaks** in your template matching score map (that is, the 3 best candidate locations for the template). [\[2\]](#) [\[3\]](#)
 2. Out of these 3, **select the point that is closest to your predicted path** (for example, closest to where you expected the next template location).
 3. If none of these are close enough (beyond a threshold distance), you can discard them or fall back to some recovery logic.

What does "point" refer to?

- In this context, a **point** refers to the coordinate (often the top-left corner or center) of a location in the big image where the template matching algorithm has found a good match.
- Each point is a (x, y) pixel coordinate.
- For every frame/step, you generate candidate points from the similarity score map, and then you want to choose the one that follows the "same path" — meaning it stays near the predicted or previously found point, ideally forming a track or sequence under a certain slope/direction. [\[4\]](#) [\[1\]](#)

Example (step by step):

1. **Template matching** produces a score map for possible template locations in the image.
2. **Find local maxima (peaks):** Pick the top 3 locations with the highest matching scores as candidate points.
3. **Distance check:** For each of these peaks, compute distance from the last known position (or predicted path point).
4. **Threshold check:** Only keep those peaks whose distance is less than a set threshold.
5. **Pick the best:** Of those candidates, choose the one that is closest to the prediction/path. This becomes your accepted next point.

Why this works

- It combines **template matching score (confidence)** with **spatial consistency** — ensuring your detections stay on track rather than jumping to visually similar but spatially incorrect locations.
- This is a common strategy in tracking/registration applications where false positives can appear far from the real path. ^[3] ^[4]

Would you like code for this logic or help adding it to your project?

Let me know how you'd like to proceed!

If you tell me which part of your code handles the picking of detected locations/peaks, I can show you exactly how to add this logic and make your path more robust.

✱

1. <https://blog.roboflow.com/template-matching/>
2. https://scikit-image.org/docs/0.25.x/auto_examples/features_detection/plot_template.html
3. https://docs.opencv.org/4.x/d4/dc6/tutorial_py_template_matching.html
4. https://docs.adaptive-vision.com/4.7/studio/machine_vision_guide/TemplateMatching.html
5. <https://www.sciencedirect.com/science/article/abs/pii/S0263224123008631>
6. <https://www.st-andrews.ac.uk/~wjh/dataview/tutorials/event-template.html>
7. <https://www.scirp.org/journal/paperinformation?paperid=55307>
8. <https://ieeexplore.ieee.org/document/6717843/>