How to find the junction points or segments in a skeletonized image Python OpenCV?

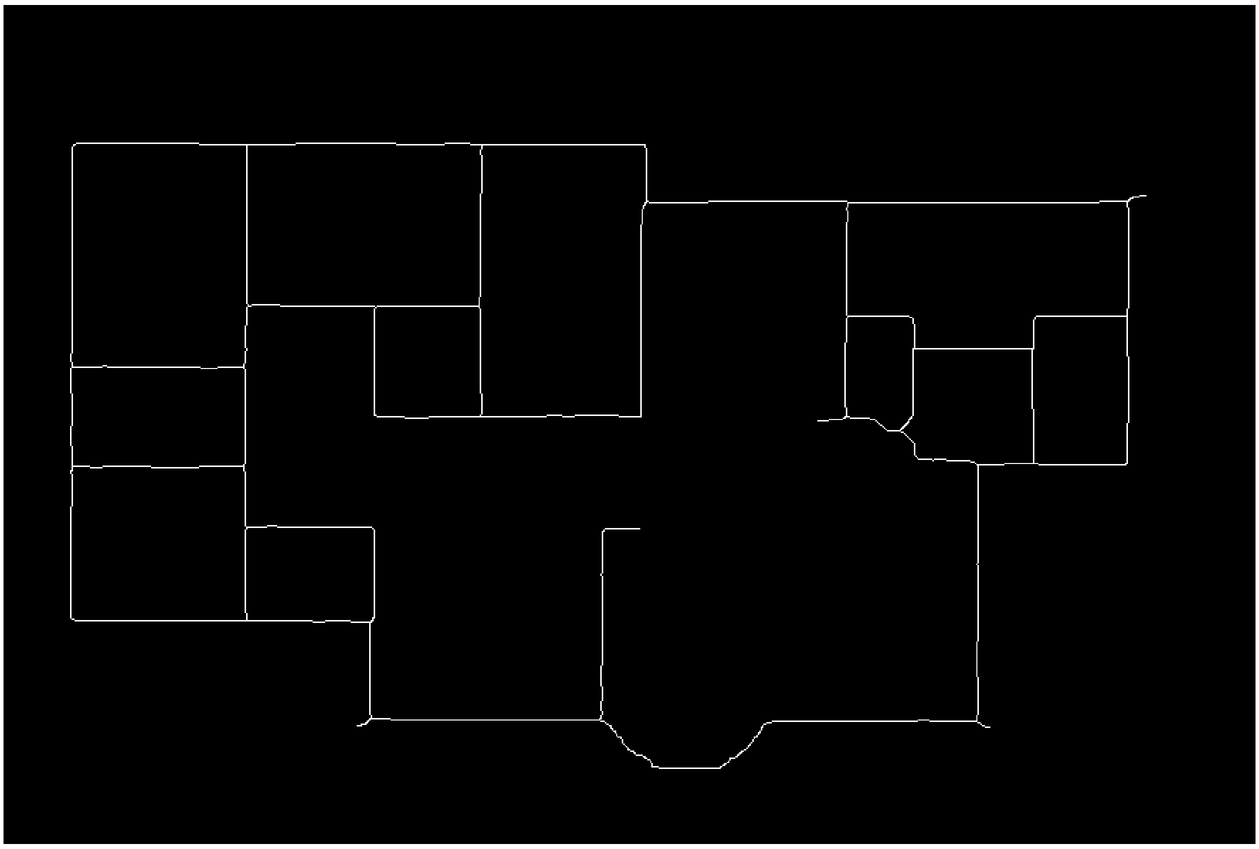
Asked 1 month ago Modified 1 month ago Viewed 113 times

2

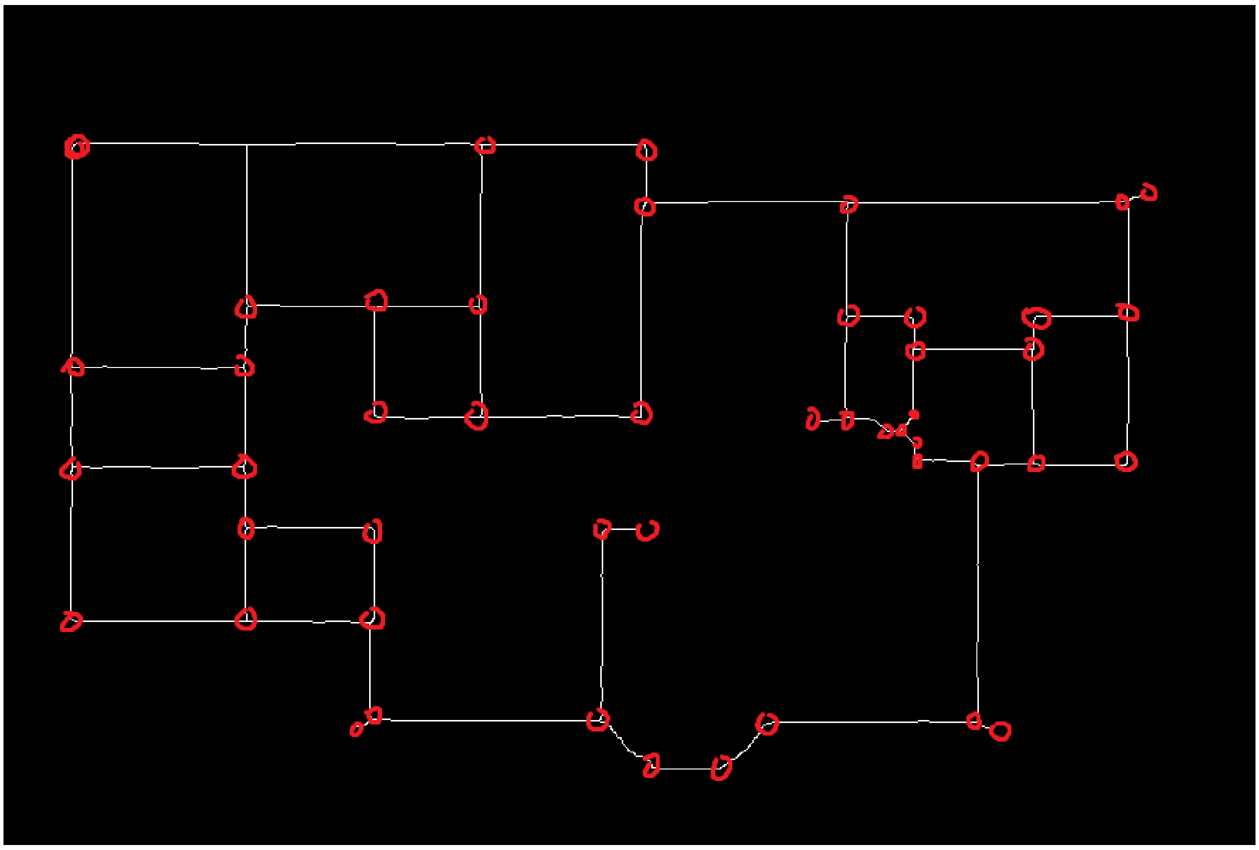
1

I am trying to convert the result of a skeletonization into a set of line segments, where the vertices correspond to the junction points of the skeleton. The shape is not a closed polygon and it may be somewhat noisy (the segments are not as straight as they should be).

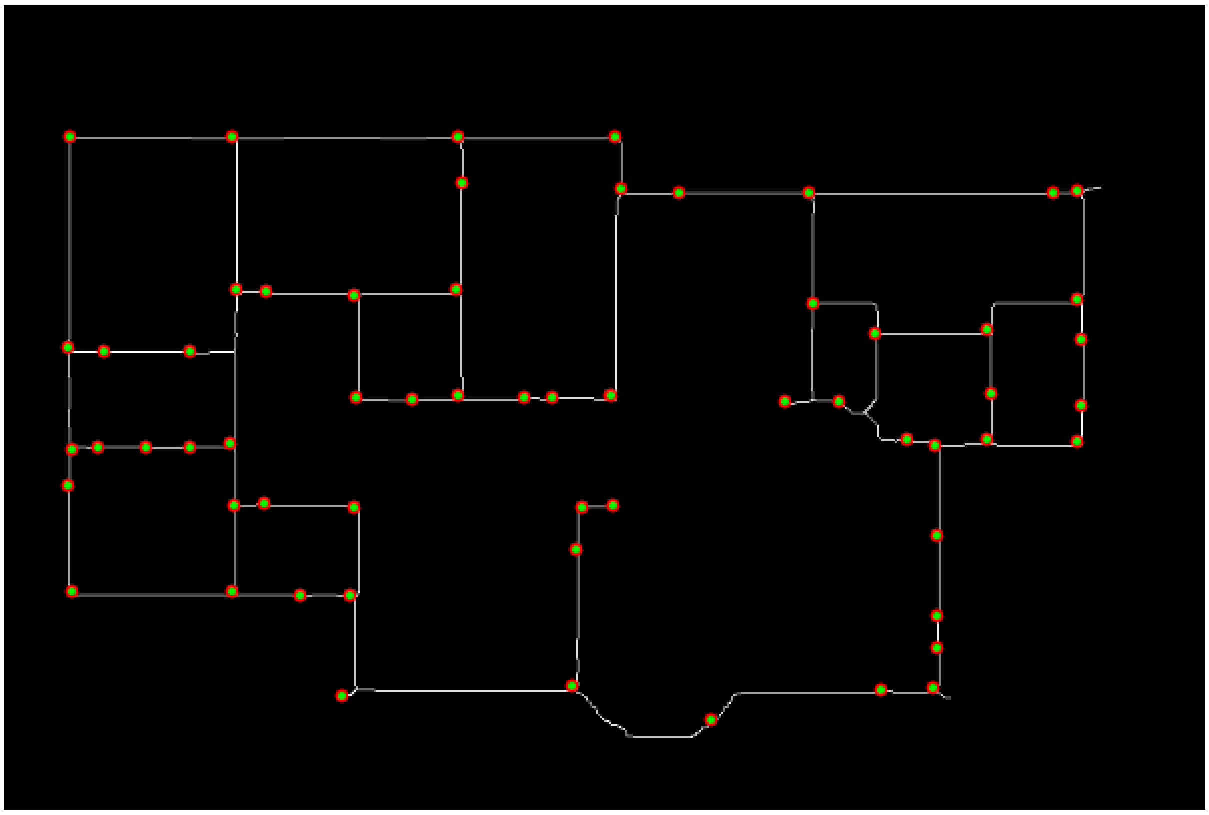
Here is an example input image:



And here are the points I want to retrieve:



I have tried using the harris corner detector, but it has trouble in some areas even after trying to tweak the algorithm's parameters (such as the angled section on the bottom of the image). Here are the results:



Do you know of any method capable of doing this? I am using python with mostly OpenCV and Numpy but I am not bound to any library. Thanks in advance.

Edit: I've gotten some good responses regarding the junction points, I am really grateful. I would also appreciate any solutions regarding extracting line segments from the junction points. I think @nathancy's answer could be used to extract line segments by subtracting the masks with the intersection mask, but I am not sure.

python image opencv image-processing computer-vision

Share Follow

edited May 9 at 0:19

asked May 8 at 20:10
João David
132 2 8

2 Answers

Sorted by:
Highest score (default)

2

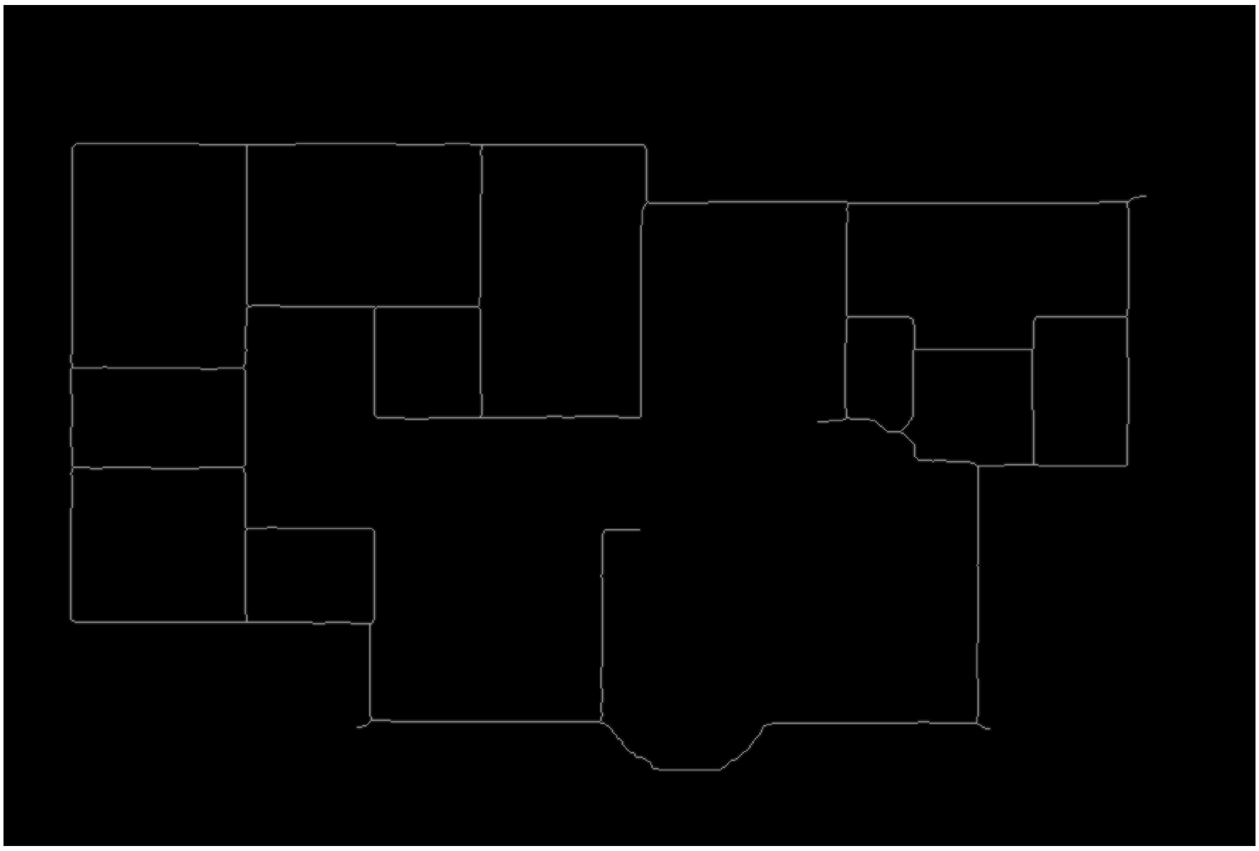
My approach is based on my previous answer here. It involves **convolving** the image with a *special kernel*. This convolution identifies the **end-points** of the lines, as well as the **intersections**. This will result in a points mask containing the pixel that matches the points you are looking for. After that, apply a little bit of **morphology** to join possible duplicated points. The method is sensible to the corners produced by the skeleton.

This is the code:

```
import cv2
import numpy as np

# image path
path = "D://opencvImages//"
fileName = "Repn3.png"

# Reading an image in default mode:
inputImage = cv2.imread(path + fileName)
inputImageCopy = inputImage.copy()

# Convert to grayscale:
grayscaleImage = cv2.cvtColor(inputImage, cv2.COLOR_BGR2GRAY)

# Compute the skeleton:
skeleton = cv2.ximgproc.thinning(grayscaleImage, None, 1)

# Threshold the image so that white pixels get a value of 10 and
# black pixels a value of 0:
_, binaryImage = cv2.threshold(skeleton, 128, 10, cv2.THRESH_BINARY)

# Set the convolution kernel:
h = np.array([[1, 1, 1],
              [1, 10, 1],
              [1, 1, 1]])

# Convolve the image with the kernel:
imgFiltered = cv2.filter2D(binaryImage, -1, h)
```

So far I convolved the skeleton image with my *special kernel*. You can inspect the image produced and search for the numerical values at the corners and intersections.

This is the output so far:



Next, identify a corner or an intersection. This bit is tricky, because the threshold value depends directly on the skeleton image, which sometimes doesn't produce good (close to straight) corners:

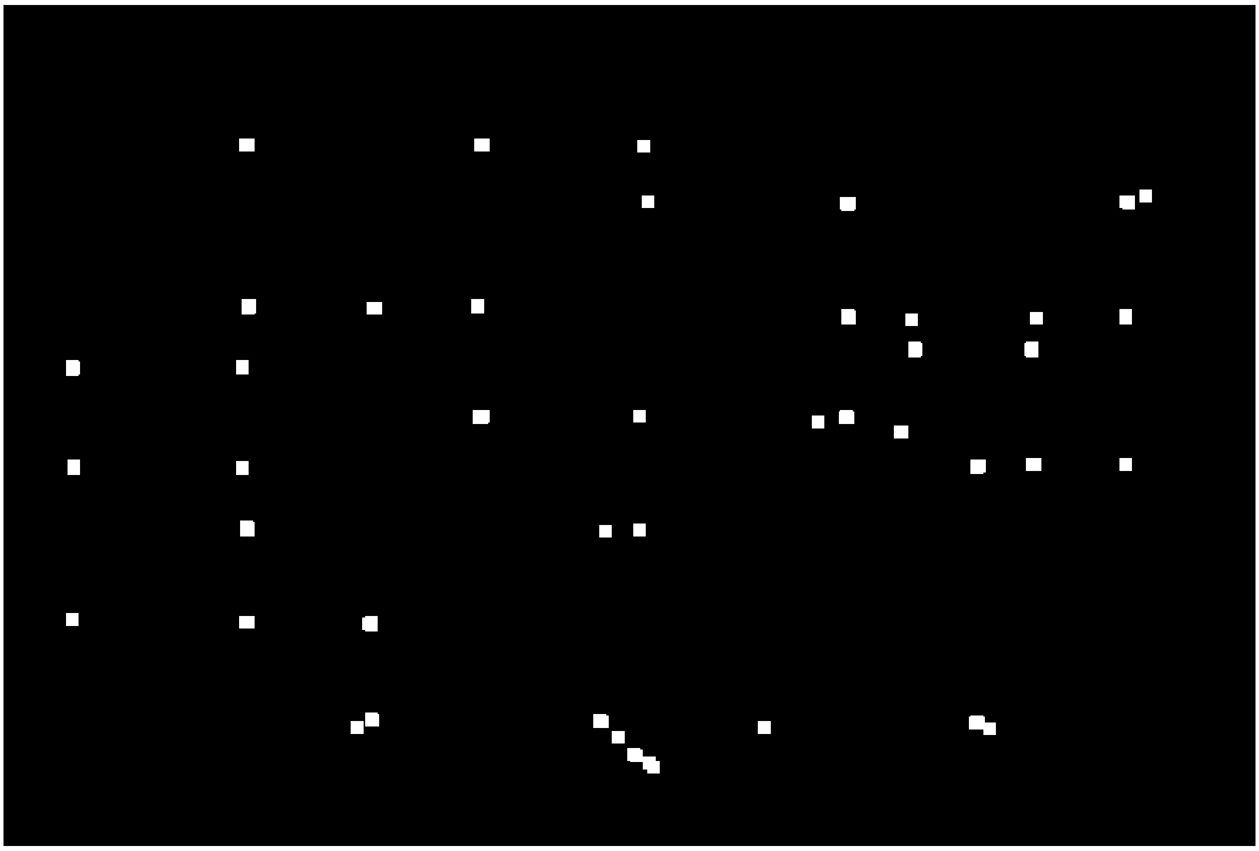
```
# Create list of thresholds:
thresh = [130, 110, 40]

# Prepare the final mask of points:
(height, width) = binaryImage.shape
pointsMask = np.zeros((height, width, 1), np.uint8)

# Perform convolution and create points mask:
for t in range(len(thresh)):
    # Get current threshold:
    currentThresh = thresh[t]
    # Locate the threshold in the filtered image:
    tempMat = np.where(imgFiltered == currentThresh, 255, 0)
    # Convert and shape the image to a uint8 height x width x channels
    # numpy array:
    tempMat = tempMat.astype(np.uint8)
    tempMat = tempMat.reshape(height,width,1)
    # Accumulate mask:
    pointsMask = cv2.bitwise_or(pointsMask, tempMat)
```

This is the binary mask:

Let's dilate to join close points:

```
# Set kernel (structuring element) size:
kernelSize = 3
# Set operation iterations:
opIterations = 4
# Get the structuring element:
morphKernel = cv2.getStructuringElement(cv2.MORPH_RECT, (kernelSize, kernelSize))
# Perform Dilate:
pointsMask = cv2.morphologyEx(pointsMask, cv2.MORPH_DILATE, morphKernel, None, None,
opIterations, cv2.BORDER_REFLECT101)
```

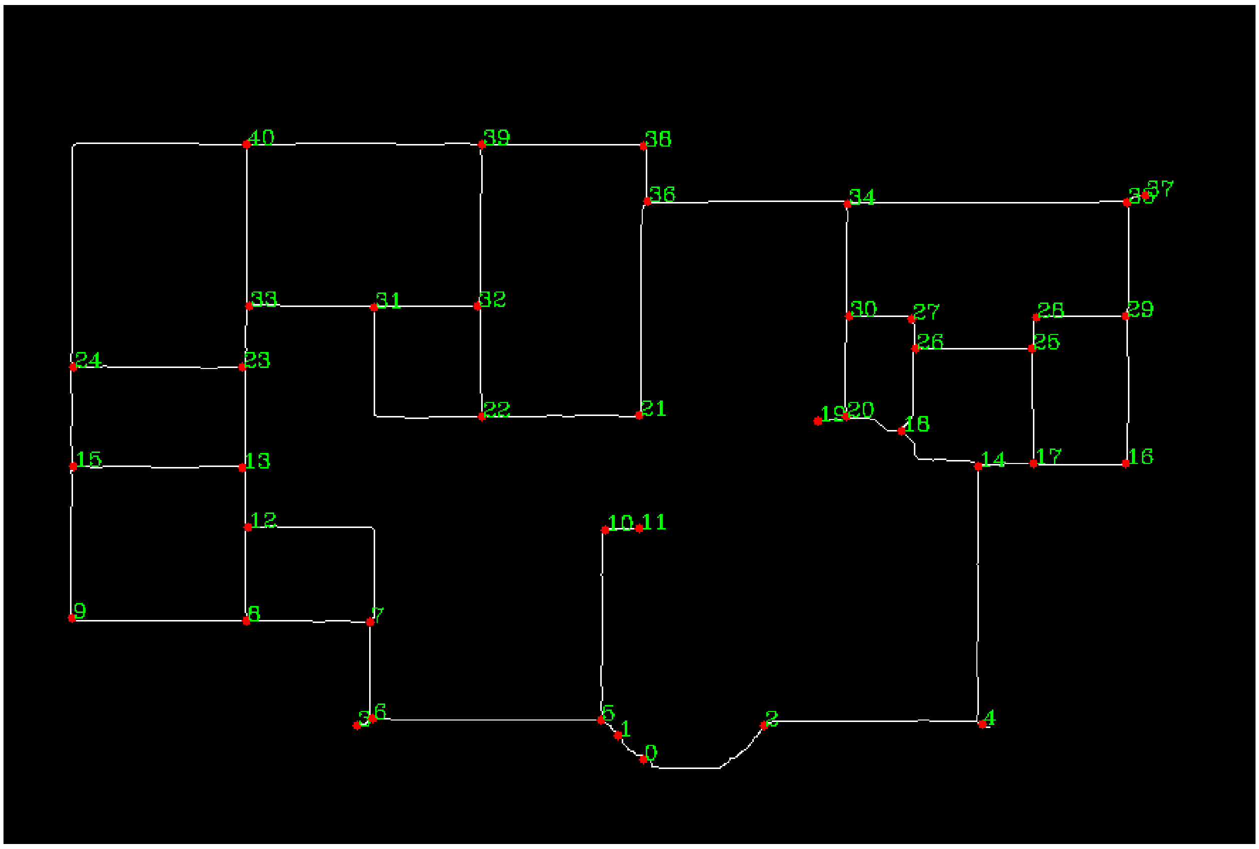
This is the output:



Now simple extract external contours. Get their bounding boxes and calculate their centroid:

```
# Look for the outer contours (no children):
contours, _ = cv2.findContours(pointsMask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

# Store the points here:
pointsList = []

# Loop through the contours:
for i, c in enumerate(contours):

    # Get the contours bounding rectangle:
    boundRect = cv2.boundingRect(c)

    # Get the centroid of the rectangle:
    cx = int(boundRect[0] + 0.5 * boundRect[2])
    cy = int(boundRect[1] + 0.5 * boundRect[3])

    # Store centroid into list:
    pointsList.append( (cx,cy) )

    # Set centroid circle and text:
    color = (0, 0, 255)
    cv2.circle(inputImageCopy, (cx, cy), 3, color, -1)
    font = cv2.FONT_HERSHEY_COMPLEX
    cv2.putText(inputImageCopy, str(i), (cx, cy), font, 0.5, (0, 255, 0), 1)

    # Show image:
    cv2.imshow("Circles", inputImageCopy)
    cv2.waitKey(0)
```

This is the result. Some corners are missed, you might one to improve the solution before computing the skeleton.



Share Follow

answered May 8 at 22:07
stateMachine
4,190 3 11 25

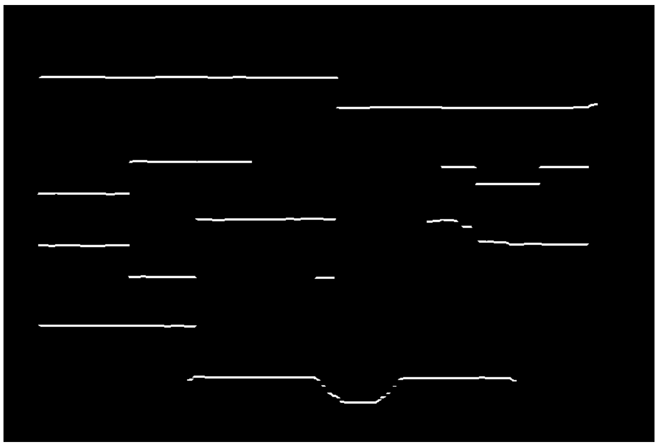
Thank you, great response! It seems to miss some points that are important, but mixing this for the endpoints with the response from @nathancy might yield good results. – João David May 9 at 0:12

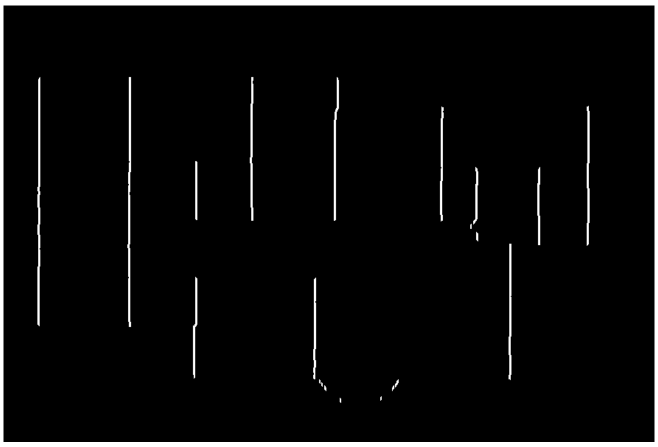
1

Here's a simple approach, the idea is:

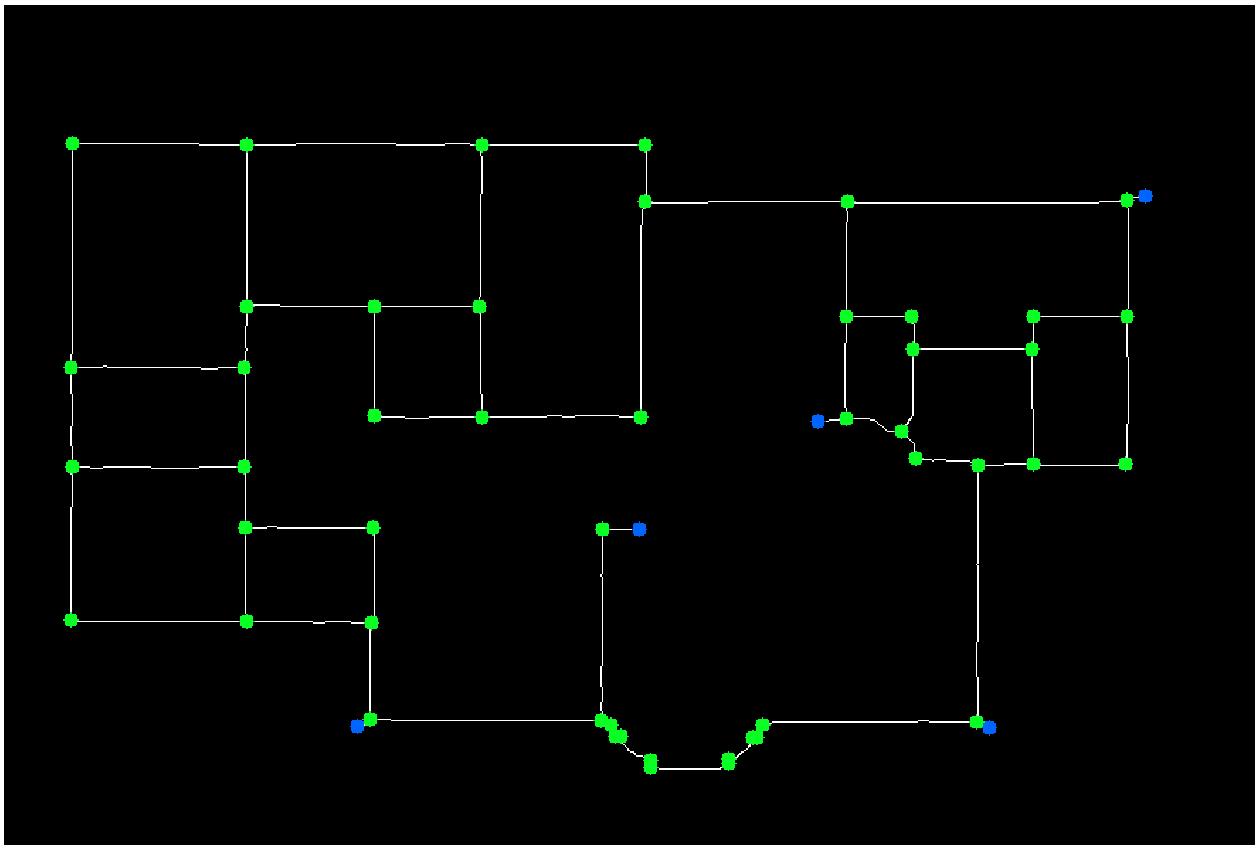
1. **Obtain binary image.** Load image, convert to grayscale, Gaussian blur, then Otsu's threshold.
2. **Obtain horizontal and vertical line masks.** Create horizontal and vertical structuring elements with `cv2.getStructuringElement` then perform `cv2.morphologyEx` to isolate the lines.
3. **Find joints.** We `cv2.bitwise_and` the two masks together to get the joints. The idea is that the intersection points on the two masks are the joints.
4. **Find centroid on joint mask.** We find contours then calculate the centroid.
5. **Find leftover endpoints.** Endpoints do not correspond to an intersection so to find those, we can use the Shi-Tomasi Corner Detector

Horizontal and vertical line masks





Results (joints in green and endpoints in blue)



Code

```
import cv2
import numpy as np
```

```python
# Load image, grayscale, Gaussian blur, Otsus threshold
image = cv2.imread('1.png')
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
blur = cv2.GaussianBlur(gray, (3,3), 0)
thresh = cv2.threshold(blur, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)[1]

# Find horizonal lines
horizontal_kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (5,1))
horizontal = cv2.morphologyEx(thresh, cv2.MORPH_OPEN, horizontal_kernel, iterations=1)

# Find vertical lines
vertical_kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (1,5))
vertical = cv2.morphologyEx(thresh, cv2.MORPH_OPEN, vertical_kernel, iterations=1)

# Find joint intersections then the centroid of each joint
joints = cv2.bitwise_and(horizontal, vertical)
cnts = cv2.findContours(joints, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
cnts = cnts[0] if len(cnts) == 2 else cnts[1]
for c in cnts:
    # Find centroid and draw center point
    x,y,w,h = cv2.boundingRect(c)
    centroid, coord, area = cv2.minAreaRect(c)
    cx, cy = int(centroid[0]), int(centroid[1])
    cv2.circle(image, (cx, cy), 5, (36,255,12), -1)

# Find endpoints
corners = cv2.goodFeaturesToTrack(thresh, 5, 0.5, 10)
corners = np.int0(corners)
for corner in corners:
    x, y = corner.ravel()
    cv2.circle(image, (x, y), 5, (255,100,0), -1)

cv2.imshow('thresh', thresh)
cv2.imshow('joints', joints)
cv2.imshow('horizontal', horizontal)
cv2.imshow('vertical', vertical)
cv2.imshow('image', image)
cv2.waitKey()
```

Share Follow

edited May 9 at 1:41

answered May 9 at 0:00

nathancy
35.9k 13 89 114

Really simple approach and the results seem to hit more points than the other answer, I find it curious how it actually handles the angled area properly. However, it does not handle the endpoints (that do not correspond to an intersection), but that can be patched in. I also mention that the line segments would be helpful, I'll edit my question to give more emphasis to that part, any idea on how those could be retrieved? – João David May 9 at 0:15

To handle endpoints, you can use `cv2.goodFeaturesToTrack` . See the update. You may need to adjust the parameters depending on the image – nathancy May 9 at 1:42

That seems to work, however, this would fail for segments angled 45º, right? My guess is that those would not appear in the vertical or horizontal mask. – João David May 9 at 9:13

No, it would work for any angle since it's searching on the Otsu image not the V or H mask. The only downside is that it's a pain to manually adjust the parameters. – nathancy May 9 at 10:06

1 I'm not sure I understand, I'll leave that step to you – nathancy May 9 at 16:48