

# sorting points to form a continuous line

Asked 6 years ago   Modified 12 months ago   Viewed 12k times



40



I have a list of (x,y)-coordinates that represent a line skeleton. The list is obtained directly from a binary image:

```
import numpy as np
list=np.where(img_skeleton>0)
```



18

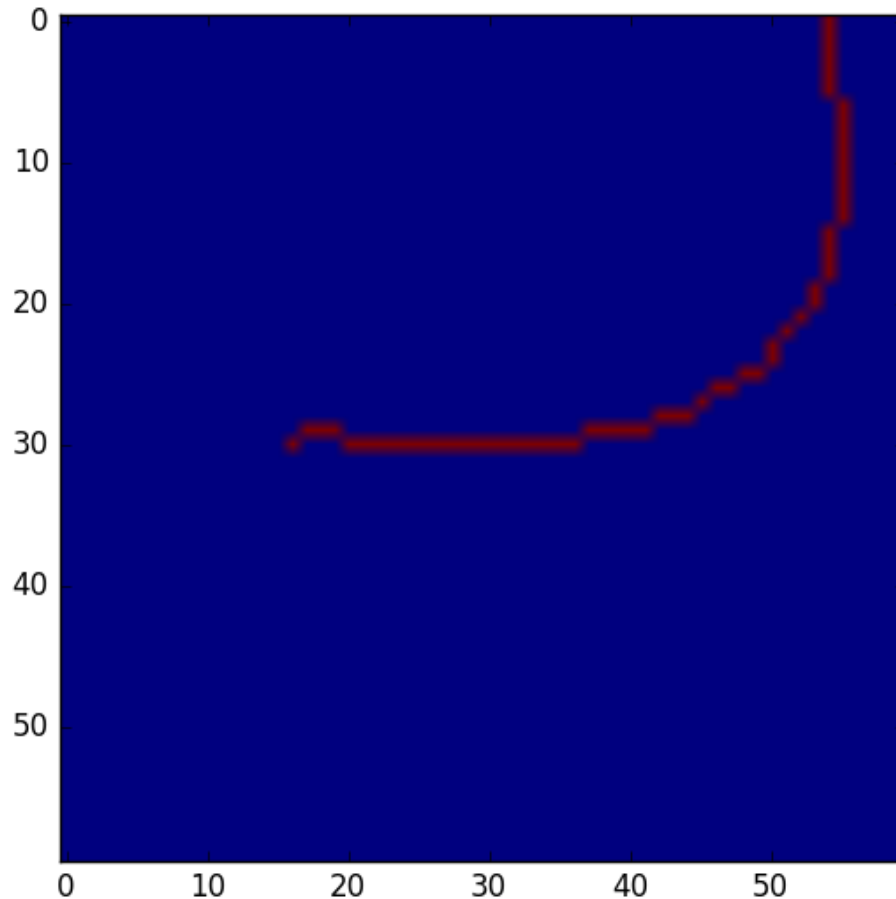


Now the points in the list are sorted according to their position in the image along one of the axes.

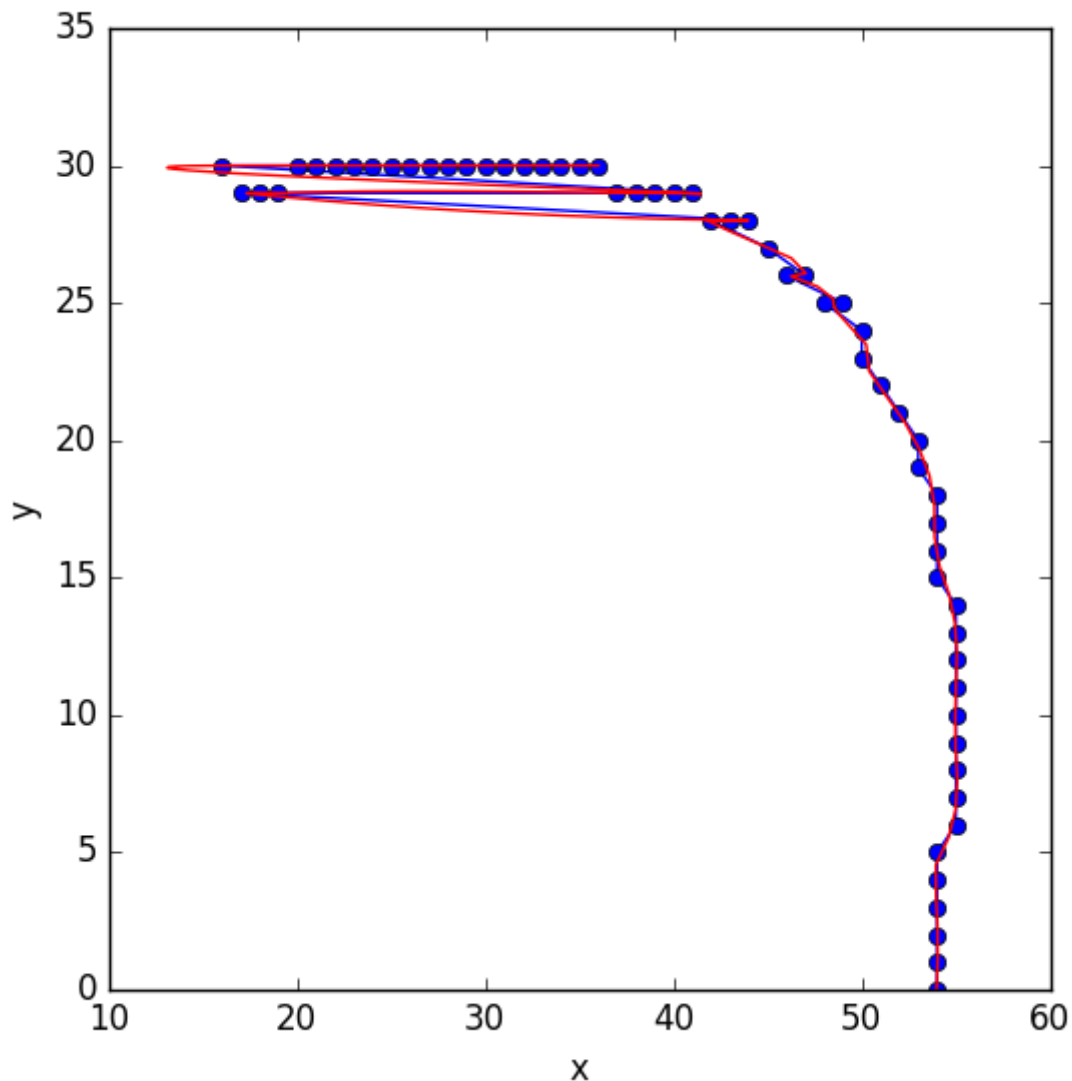
I would like to sort the list such that the order represents a smooth path along the line. (This is currently not the case where the line curves back). Subsequently, I want to fit a spline to these points.

A similar problem has been described and solved using arcpy [here](#). Is there a convenient way to achieve this using python, numpy, scipy, openCV (or another library?)

below is an example image. it results in a list of 59 (x,y)-coordinates.



when I send the list to scipy's spline fitting routine, I am running into a problem because the points aren't 'ordered' on the line:



[python](#) [opencv](#) [numpy](#) [image-processing](#) [scipy](#)

Share Improve this question

Follow

edited Apr 13, 2017 at 12:33

asked Jun 10, 2016 at 7:27



Community Bot

1 1



jlarsch

2,067

4 18 43

You probably want to store vectors in order and keep a starting point instead? Would that be possible?  
– [Torxed](#) Jun 10, 2016 at 7:34

1 You are provably looking at a "sort by nearest neighbor" function tho, that's a good search term :)  
– [Torxed](#) Jun 10, 2016 at 7:39

1 This problem is equivalent to finding the shortest path in a graph, where the graph is created as a fully connected graph (where your points are nodes) and edges are weighted by the euclidean distances between points. – [Imanol Luengo](#) Jun 10, 2016 at 8:26

1 None of the answers below address the actual problem here, which is the way that points are extracted

from the image. It is fairly simple to extract the points in the right order, leading to a much more efficient algorithm than any of the solutions below. – [Cris Luengo](#) Apr 23, 2019 at 19:12

1 What would be a starting point to achieve that? – [jlarsch](#) Apr 24, 2019 at 8:05

5 Answers

Sorted by:

Highest score (default)



I apologize for the long answer in advance :P (the problem is not *that* simple).

39



Lets start by rewording the problem. Finding a line that connects all the points, can be reformulated as a shortest path problem in a graph, where (1) the graph nodes are the points in the space, (2) each node is connected to its 2 nearest neighbors, and (3) the shortest path passes through each of the nodes **only once**. That last constrain is a very important (and quite hard one to optimize). Essentially, the problem is to find a permutation of length  $n$ , where the permutation refers to the order of each of the nodes ( $n$  is the total number of nodes) in the path.

Finding all the possible permutations and evaluating their cost is too expensive (there are  $n!$  permutations if I'm not wrong, which is too big for problems). Bellow I propose an approach that finds the  $n$  best permutations (the optimal permutation for each of the  $n$  points) and then find the permutation (from those  $n$ ) that minimizes the error/cost.

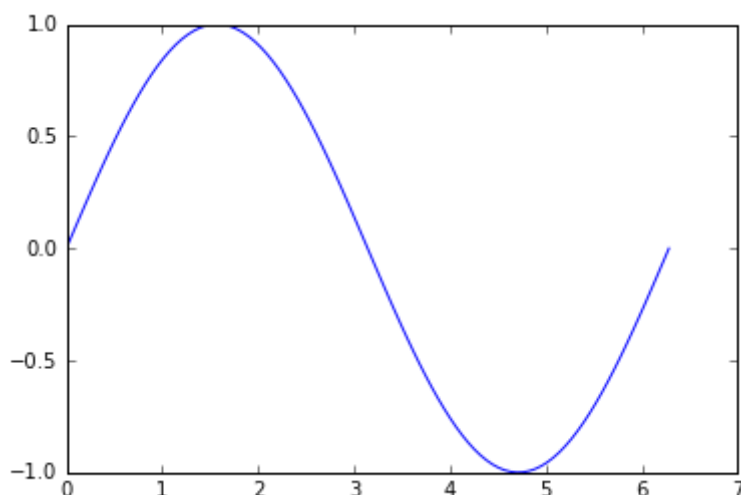
## 1. Create a random problem with unordered points

Now, lets start to create a sample problem:

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 2 * np.pi, 100)
y = np.sin(x)

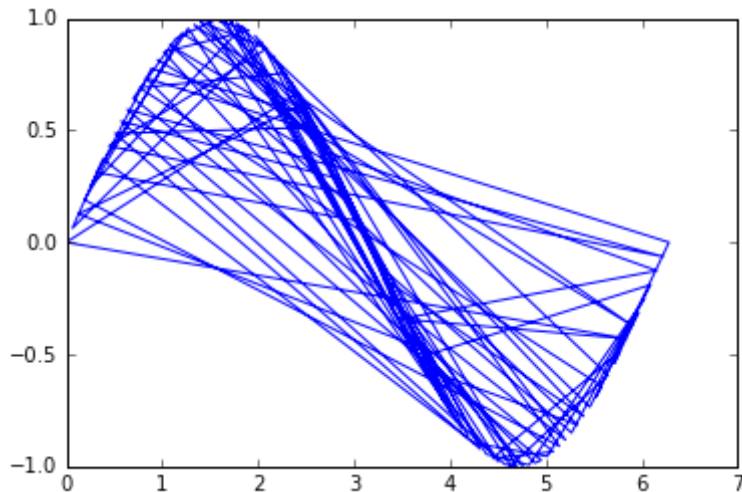
plt.plot(x, y)
plt.show()
```



And here, the unsorted version of the points  $[x, y]$  to simulate a random points in space connected in a line:

```
idx = np.random.permutation(x.size)
x = x[idx]
y = y[idx]

plt.plot(x, y)
plt.show()
```



The problem is then to order those points to recover their original order so that the line is plotted properly.

## 2. Create 2-NN graph between nodes

We can first rearrange the points in a  $[N, 2]$  array:

```
points = np.c_[x, y]
```

Then, we can start by creating a nearest neighbour graph to connect each of the nodes to its 2 nearest neighbors:

```
from sklearn.neighbors import NearestNeighbors

clf = NearestNeighbors(2).fit(points)
G = clf.kneighbors_graph()
```

$G$  is a sparse  $N \times N$  matrix, where each row represents a node, and the non-zero elements of the columns the euclidean distance to those points.

We can then use `networkx` to construct a graph from this sparse matrix:

```
import networkx as nx

T = nx.from_scipy_sparse_matrix(G)
```

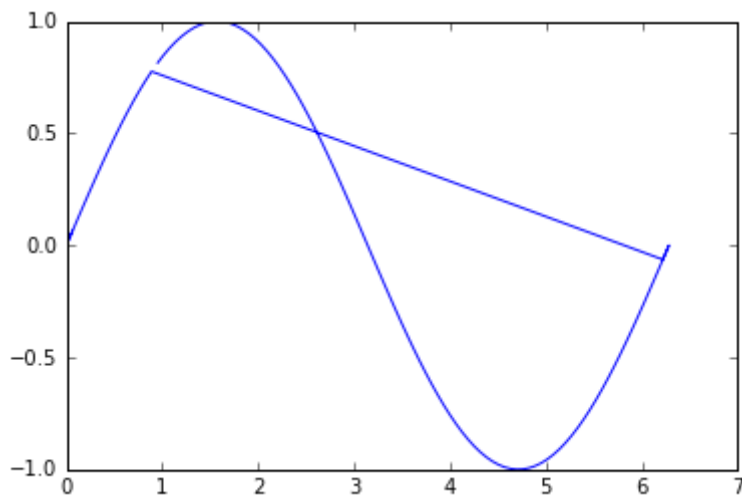
### 3. Find shortest path from source

And, here begins the *magic*: we can extract the paths using [dfs\\_preorder nodes](#), which will essentially create a path through all the nodes (passing through each of them exactly once) given a starting node (if not given, the 0 node will be selected).

```
order = list(nx.dfs_preorder_nodes(T, 0))

xx = x[order]
yy = y[order]

plt.plot(xx, yy)
plt.show()
```



Well, is not too bad, but we can notice that the reconstruction is not optimal. This is because the point 0 in the unordered list lays in the middle of the line, that is way it first goes in one direction, and then comes back and finishes in the other direction.

### 4. Find the path with smallest cost from all sources

So, in order to obtain the optimal order, we can just get the best order for all the nodes:

```
paths = [list(nx.dfs_preorder_nodes(T, i)) for i in range(len(points))]
```

Now that we have the optimal path starting from each of the  $N = 100$  nodes, we can discard them and find the one that minimizes the distances between the connections (optimization problem):

```
mindist = np.inf
minidx = 0

for i in range(len(points)):
    p = paths[i]          # order of nodes
    ordered = points[p]   # ordered nodes
    # find cost of that order by the sum of euclidean distances between points (i) and
    (i+1)
    cost = (((ordered[:-1] - ordered[1:])**2).sum(1)).sum()
```

```

if cost < mindist:
    mindist = cost
    minidx = i

```

The points are ordered for each of the optimal paths, and then a cost is computed (by calculating the euclidean distance between all pairs of points  $i$  and  $i+1$ ). If the path starts at the `start` or `end` point, it will have the smallest cost as all the nodes will be consecutive. On the other hand, if the path starts at a node that lies in the middle of the line, the cost will be very high at some point, as it will need to travel from the end (or beginning) of the line to the initial position to explore the other direction. The path that minimizes that cost, is the path starting in an optimal point.

```

opt_order = paths[minidx]

```

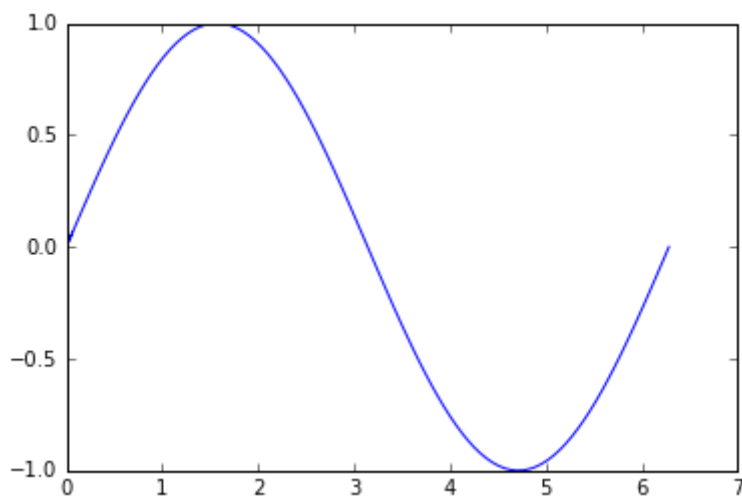
Now, we can reconstruct the order properly:

```

xx = x[opt_order]
yy = y[opt_order]

plt.plot(xx, yy)
plt.show()

```



Share Improve this answer Follow

edited Jun 10, 2016 at 12:55

answered Jun 10, 2016 at 9:21



**Imanol Luengo**


**14.5k** 2 43 66

looks cool. I like the strategy but I am getting an error on '`G = clf.kneighbors_graph()`'. error: `TypeError: kneighbors_graph() takes at least 2 arguments (1 given)`. – [jlarsch](#) Jun 10, 2016 at 11:42

@jlarsch which version of scikit-learn are you using? Because in the [last version](#) the parameters are optional. If not, try providing `clf.kneighbors_graph(points, 2)` or upgrading scikits-learn to the latest version. – [Imanol Luengo](#) Jun 10, 2016 at 12:45

- 1 @jlarsch I did edit the post to add headings to make it more clear. Essentially, if you already know either the `start` or `end` node, the section 3. gives you a 1-line solution using `nx.dfs_preorder_nodes(T, start_index)`. If you don't know (or don't want to provide manually) the initial node, section 4. computes all the possible minimum paths with each of the nodes as source,

and filters the one with minimum cost. – [Imanol Luengo](#) Jun 10, 2016 at 12:57

- 1 This line `paths = [list(nx.dfs_preorder_nodes(T, i)) for i in range(len(points))]` is generating paths of smaller size than `len(points)` in some cases, resulting in a wrong sorting. Why is this? – [learn2day](#) Sep 11, 2017 at 15:59 
- 1 This method is not robust for noisy data. It fails in graph construction stage. – [Eugene W.](#) Jan 10, 2021 at 15:08

7

One possible solution is to use a nearest neighbours approach, possible by using a KDTree. Scikit-learn has an nice interface. This can then be used to build a graph representation using networkx. This will only really work if the line to be drawn should go through the nearest neighbours:



```
from sklearn.neighbors import KDTree
import numpy as np
import networkx as nx

G = nx.Graph() # A graph to hold the nearest neighbours

X = [(0, 1), (1, 1), (3, 2), (5, 4)] # Some list of points in 2D
tree = KDTree(X, leaf_size=2, metric='euclidean') # Create a distance tree

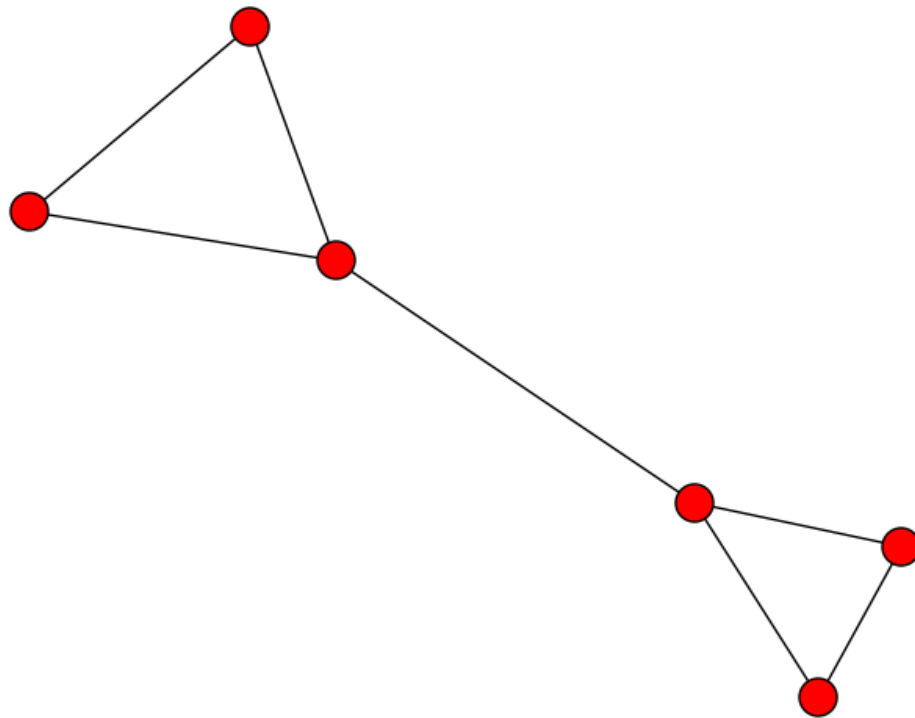
# Now loop over your points and find the two nearest neighbours
# If the first and last points are also the start and end points of the line you can
use X[1:-1]
for p in X:
    dist, ind = tree.query(p, k=3)
    print ind

    # ind Indexes represent nodes on a graph
    # Two nearest points are at indexes 1 and 2.
    # Use these to form edges on graph
    # p is the current point in the list
    G.add_node(p)
    n1, l1 = X[ind[0][1]], dist[0][1] # The next nearest point
    n2, l2 = X[ind[0][2]], dist[0][2] # The following nearest point
    G.add_edge(p, n1)
    G.add_edge(p, n2)

print G.edges() # A list of all the connections between points
print nx.shortest_path(G, source=(0,1), target=(5,4))
>>> [(0, 1), (1, 1), (3, 2), (5, 4)] # A list of ordered points
```

Update: If the start and end points are unknown and your data is reasonably well separated, you can find the ends by looking for cliques in the graph. The start and end points will form a clique. If the longest edge is removed from the clique it will create a free end in the graph which can be used as a start and end point. For example, the start and end points in this list appear in the middle:

```
X = [(0, 1), (0, 0), (2, 1), (3, 2), (9, 4), (5, 4)]
```

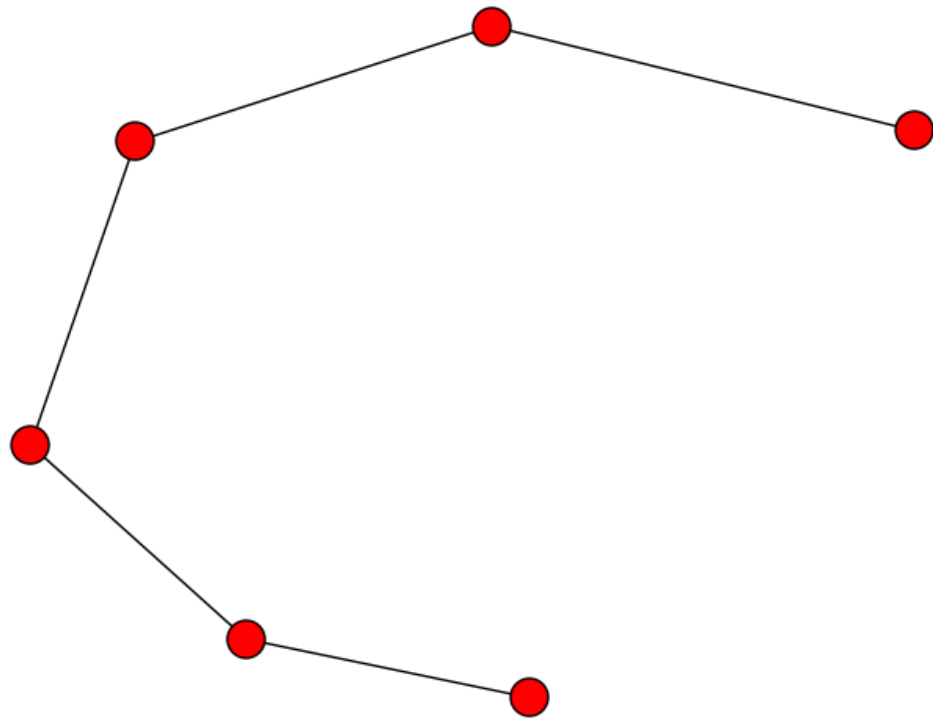


After building the graph, now its a case of removing the longest edge from the cliques to find the free ends of the graph:

```
def find_longest_edge(l):
    e1 = G[l[0]][l[1]]['weight']
    e2 = G[l[0]][l[2]]['weight']
    e3 = G[l[1]][l[2]]['weight']
    if e2 < e1 > e3:
        return (l[0], l[1])
    elif e1 < e2 > e3:
        return (l[0], l[2])
    elif e1 < e3 > e2:
        return (l[1], l[2])

end_cliques = [i for i in list(nx.find_cliques(G)) if len(i) == 3]
edge_lengths = [find_longest_edge(i) for i in end_cliques]
G.remove_edges_from(edge_lengths)
edges = G.edges()
```





```
start_end = [n for n,nbrs in G.adjacency_iter() if len(nbrs.keys()) == 1]
print nx.shortest_path(G, source=start_end[0], target=start_end[1])
>>> [(0, 0), (0, 1), (2, 1), (3, 2), (5, 4), (9, 4)] # The correct path
```

Share Improve this answer Follow

edited Jun 14, 2016 at 10:53

answered Jun 10, 2016 at 8:27



kezzos

2,793

3

18

34

You could essentially use sklearn's [NearestNeighbors](#) and its function `.kneighbors_graph()`, which would instantly give you a sparse representation of a graph (which is more clear and faster than using networkx to construct one). Anyway, after you have the graph of 2 nearest neighbors, you still have to order them, the problem is not yet solved. – [Imanol Luengo](#) Jun 10, 2016 at 8:44

I was going to say: now how am I sorting the list of graph edges? Perhaps there is a utility function in networkx that I am missing? – [jlarsch](#) Jun 10, 2016 at 8:47

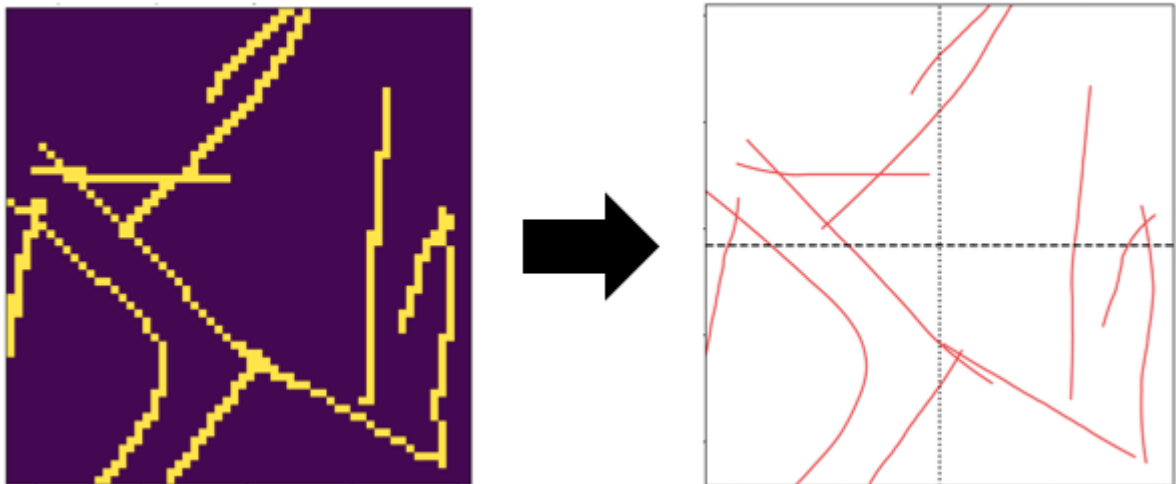
I didnt know about `kneighbour_graph()`, thanks! You dont need to sort the edges to draw a line. Just draw a line between nodes which have an edge and you will have a line, all joined up. But there is probably also a function which gives you the path in networkx. Sorry im not that familiar with networkx – [kezzos](#) Jun 10, 2016 at 9:04

maybe I wasn't clear but the point is not to draw the line but to obtain a sorted list of coordinates that I can feed into a spline fitting algorithm. for that, the list must be sorted as far as I understand it. – [jlarsch](#) Jun 10, 2016 at 9:07

- 1 I recommend looking at `.kneighbors_graph()` as suggested by Imanol, or look at `nx.shortest_path()` function: `print nx.shortest_path(G, source=(0,1), target=(5,4))` – [kezzos](#) Jun 10, 2016 at 9:11

4

I had the exact same problem. If you have two arrays of scattered x and y values that are not too curvy, then you can transform the points into PCA space, sort them in PCA space, and then transform them back. (I've also added in some bonus smoothing functionality).



```
import numpy as np
from scipy.signal import savgol_filter
from sklearn.decomposition import PCA

def XYclean(x,y):

    xy = np.concatenate((x.reshape(-1,1), y.reshape(-1,1)), axis=1)

    # make PCA object
    pca = PCA(2)
    # fit on data
    pca.fit(xy)

    #transform into pca space
    xypca = pca.transform(xy)
    newx = xypca[:,0]
    newy = xypca[:,1]

    #sort
    indexSort = np.argsort(x)
    newx = newx[indexSort]
    newy = newy[indexSort]

    #add some more points (optional)
    f = interpolate.interp1d(newx, newy, kind='linear')
    newX=np.linspace(np.min(newx), np.max(newx), 100)
    newY = f(newX)

    #smooth with a filter (optional)
    window = 43
    newY = savgol_filter(newY, window, 2)

    #return back to old coordinates
    xyclean = pca.inverse_transform(np.concatenate((newX.reshape(-1,1),
    newY.reshape(-1,1)), axis=1) )
    xc=xyclean[:,0]
    yc = xyclean[:,1]

    return xc, yc
```

I agree with Imanol\_Luengo [Imanol Luengo](#)'s solution, but **if** you know the index of the first point, then there is a considerably easier solution that uses only NumPy:

3

```
def order_points(points, ind):
    points_new = [ points.pop(ind) ] # initialize a new list of points with the known
    first point
    pcurr      = points_new[-1]      # initialize the current point (as the known
    point)
    while len(points)>0:
        d      = np.linalg.norm(np.array(points) - np.array(pcurr), axis=1) #
        distances between pcurr and all other remaining points
        ind     = d.argmin()          # index of the closest point
        points_new.append( points.pop(ind) ) # append the closest point to points_new
        pcurr   = points_new[-1]      # update the current point
    return points_new
```

This approach appears to work well with the sine curve example, especially because it is easy to define the first point as either the leftmost or rightmost point.

For the `img_skeleton` data cited in the question, it would be similarly easy to algorithmically obtain the first point, for example as the topmost point.

```
# create sine curve:
x      = np.linspace(0, 2 * np.pi, 100)
y      = np.sin(x)

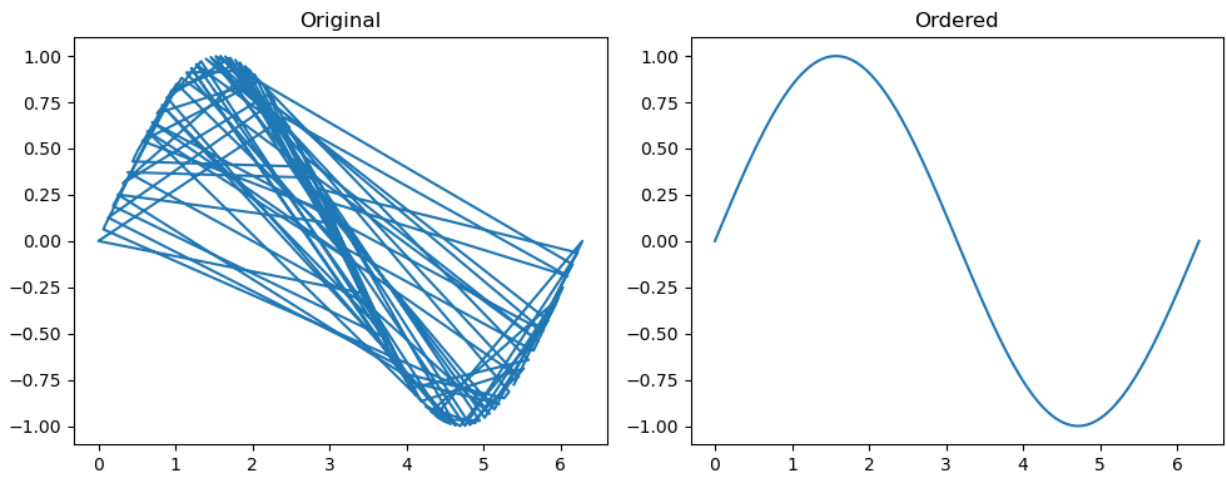
# shuffle the order of the x and y coordinates:
idx     = np.random.permutation(x.size)
xs,ys   = x[idx], y[idx] # shuffled points

# find the leftmost point:
ind     = xs.argmin()

# assemble the x and y coordinates into a list of (x,y) tuples:
points = [(xx,yy) for xx,yy in zip(xs,ys)]

# order the points based on the known first point:
points_new = order_points(points, ind)

# plot:
fig,ax = plt.subplots(1, 2, figsize=(10,4))
xn,yn   = np.array(points_new).T
ax[0].plot(xs, ys) # original (shuffled) points
ax[1].plot(xn, yn) # new (ordered) points
ax[0].set_title('Original')
ax[1].set_title('Ordered')
plt.tight_layout()
plt.show()
```



Share Improve this answer Follow

answered Jun 25, 2021 at 9:26



ToddP

574 10 17

Note that the left-most point may not always be an end-point. Consider a sine wave rotated 90 degrees -- i.e. we couldn't assume the "top-most" location was an end-point. – [georgedeath](#) Dec 13, 2021 at 14:06

- 1 Agreed. This solution requires that the index of the first point is known. If you can find the first point algorithmically or manually, then this numpy-only solution seems simplest. – [ToddP](#) Dec 14, 2021 at 16:20

1 I am working on a similar problem, but it has an important constraint (much like the example given by the OP) which is that each pixel has either one or two neighboring pixel, in the 8-connected sense. With this constraint, there is a very simple solution.

```
def sort_to_form_line(unsorted_list):
    """
    Given a list of neighboring points which forms a line, but in random order,
    sort them to the correct order.
    IMPORTANT: Each point must be a neighbor (8-point sense)
    to a least one other point!
    """
    sorted_list = [unsorted_list.pop(0)]

    while len(unsorted_list) > 0:
        i = 0
        while i < len(unsorted_list):
            if are_neighbours(sorted_list[0], unsorted_list[i]):
                #neighbours at front of list
                sorted_list.insert(0, unsorted_list.pop(i))
            elif are_neighbours(sorted_list[-1], unsorted_list[i]):
                #neighbours at rear of list
                sorted_list.append(unsorted_list.pop(i))
            else:
                i = i+1

        return sorted_list

def are_neighbours(pt1, pt2):
    """
    Check if pt1 and pt2 are neighbours, in the 8-point sense
```

```
pt1 and pt2 has integer coordinates
"""
```

```
return (np.abs(pt1[0]-pt2[0]) < 2) and (np.abs(pt1[1]-pt2[1]) < 2)
```

[Share](#) [Improve this answer](#) [Follow](#)

[edited Apr 23, 2019 at 19:01](#)



[DarkCygnus](#)

**6,732** 3 36 53

[answered Apr 9, 2018 at 10:04](#)



[redraider](#)

**11** 1

---