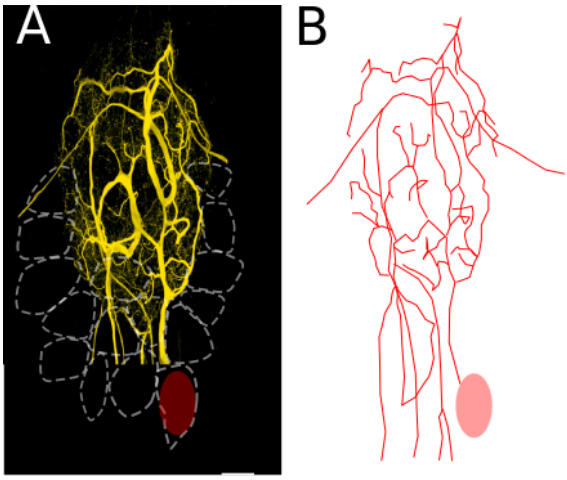
Quantifying STG neuronal morphology

Stomatogastric ganglion (STG) neurons control peristaltic muscles in the stomach of crustacea. STG somata are segregated from the neuropil; a single neurite from the unipolar somata enters the neuropil where it ramifies extensively to form electrical and chemical synapses.
Question: Are the branching structures of STG dendrites stereotyped in any way?



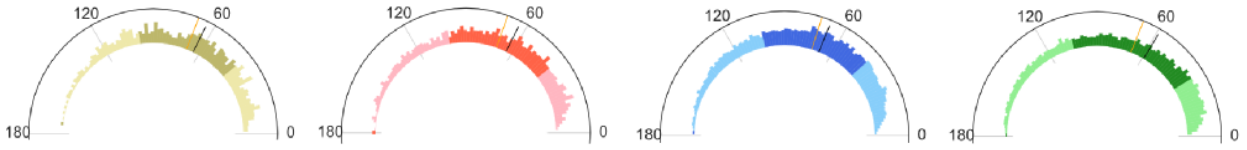
Neurons are reduced to skeletons for analysis by diligent tracers. A skeleton is a collection of *(x,y,z)* nodes with connectivity specified by edges. A Python-MATLAB toolbox called *NeuronGeoemtry* reads the skeletons and parses them into geometry objects containing segments, branches, nodes and other classes that facilitate analysis.

NeuronGeometry

- Geometry objects use graph theory to allow rapid and easy analysis of skeletons.
- The *structure* of geometry objects involves nested graphs. The top object is a *geometry* object, which is a neuron.
- A geometry object has many attributes, including
  1. *nodes* are the simplest unit, containing a single *(x,y,z)* tuple. Nodes are also graph objects and their neighbors are indexed.
  2. *segments* contain at least two nodes and are the fundamental edges of the geometry object. Segments are used to create paths and find distances in the geometry object.
  3. *branches* contain at least one segment and represent the collection of segments between branchpoints. Branches can be traversed in a somatofugal or centripetal manner to assign branch orders.

Extracting features from geometry objects

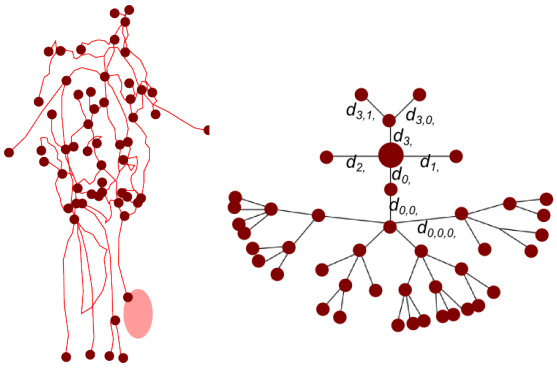
Branch Angles



Branch angles are identified each time a a branchpoint occurs and are determined as the angle between the daughter branch and the continuation of the parent branch. A 60-degree branch angle indicates 60 degrees deviation from the parent branch, which 0 degrees would be perfectly in line with the parent.

Using Dijkstra's algorithm to explore graph properties

We use an implementation of Dijkstra's algorithm to determine the shortest path length traversing segments and, as a result, the nodes and vertices that contribute to that path. We represent all edges equally, but they are weighted by their lengths (i.e.: $d_{0,1} \neq d_{0,0}$.



Dijkstra's algorithm iterates through nodes the edges that connect with its neighbors and keep the path that is the shortest to the target. Instead of a *directed* search to a target, my algorithm performs an *exhaustive* search, finding the distance and path to every other segment in the graph.

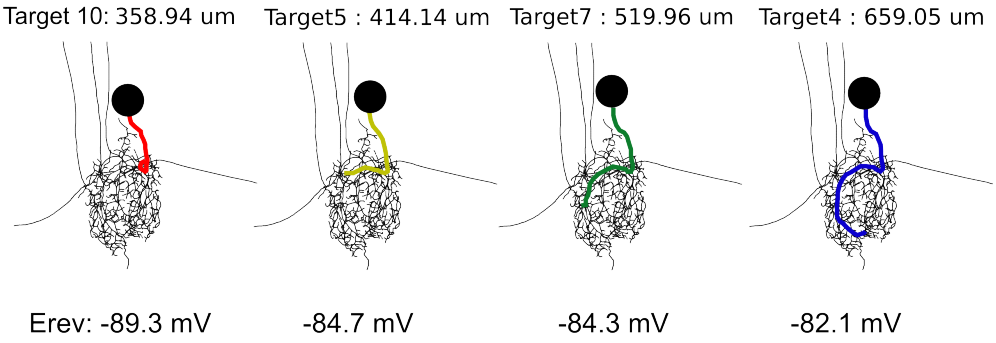
Code snippet: Dijkstra's algorithm

```
def _computeDistances(self):
    # use Dijkstras algorithm to find path distance from start to rest of
    # neuron.
    # Keep track of effect of startPos (starting position in startSegment)
    # Also keep track of effect of pos of each final segment
    segment, startPos = self.startSegment, self.startPos
    # distances is a dict object, with segments as keys
    # the values are a list of paths, each path described by a tuple
    #   (pathDistance, connecting location in the segment, path description,
    #    list of segments in path )
    distances = { segment : [(0.0, startPos,
                            segment.name + '(%.1f)' % startPos, [segment])] }
    branchOrders = { segment : 0 }
    openSegments = { segment, }
    while openSegments:
      segment = openSegments.pop()
      for currentD, startPos, segPathDesc, segPath in distances[segment]:
        branchOrderInc = 1
        if branchOrders[segment] > 0 and len(segment.neighbors) <= 2:
          branchOrderInc = 0
        #branchOrderInc = int(len(segment.neighbors) > 2)
        for neighbor, (connectLoc, nConnectLoc, node) \
            in zip(segment.neighbors, segment.neighborLocations):
          pathD = currentD + segment.length * abs(startPos - connectLoc)
          # check if neighbor in distances?
          if neighbor not in distances:
            # found path to new segment
            nPath = segPath + [neighbor]
            nPathDesc = segPathDesc + '->(%.1f)->' % connectLoc + \
                  neighbor.name + '(%.1f)' % nConnectLoc
            distances[neighbor] = [(pathD, nConnectLoc, nPathDesc, nPath)]
            openSegments.add(neighbor)
            branchOrders[neighbor] = branchOrders[segment] + branchOrderInc
          else:
            # Either there is a loop involving this segment, or the path is
            #  backtracking
```

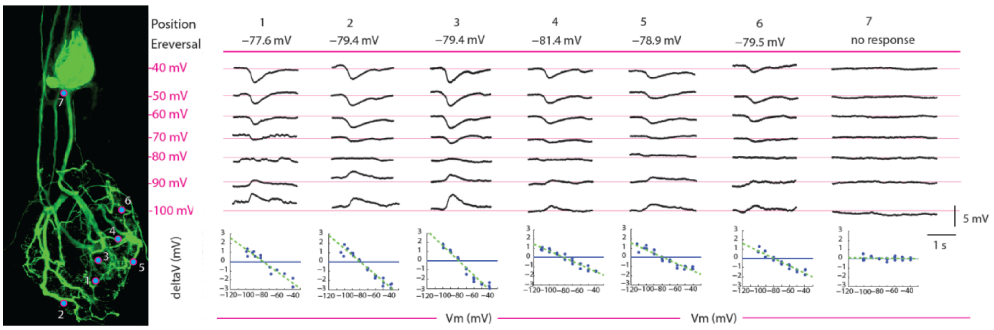
```
            # Check if the current path is an efficient route to the loop
            efficient = True
            insertInd = None
            loopDistances = distances[neighbor]
            for ind, (loopD, loopPos, loopPathDesc, loopPath) in \
                enumerate(loopDistances):
              traverse = neighbor.length * abs(loopPos - nConnectLoc)
              if pathD >= loopD + traverse:
                # the new path to neighbor is too slow to ever be useful
                efficient = False
                break
              elif pathD + traverse < loopD:
                # the new path to neighbor renders an old one obsolete
                loopDistances.pop(ind)
              if insertInd is None and pathD < loopD:
                insertInd = ind
            if efficient:
              nPath = segPath + [neighbor]
              nPathDesc = segPathDesc + '->(%.1f)->' % connectLoc + \
                    neighbor.name + '(%.1f)' % nConnectLoc
              pathInfo = (pathD, nConnectLoc, nPathDesc, nPath)
              if insertInd is None:
                loopDistances.append(pathInfo)
              else:
                loopDistances.insert(insertInd, pathInfo)
              distances[neighbor] = loopDistances
              openSegments.add(neighbor)
              branchOrders[neighbor] = branchOrders[segment] + branchOrderInc
              if self.warnLoops and len(loopDistances) > 1:
                warn('%d efficient paths to %s.' % (len(loopDistances), neighbor.name))
                for ind, loopPos, loopPath in loopDistances:
                  print(loopPath)

    return distances, branchOrders
```

We use Dijkstra's algorithm to calculate distances to **photo-uncaging** targets.



In this project, our goal was the identify regions that were electrotonically distal to the soma by photo-uncaging glutamate at various positions.



Tip identification

Using semi-automated axon identification we find the *main path*, the collection of segments connecting the soma to all axons. Every primary branch leaving the main path that does not terminate in an axon is a *subtree*. Below the subtree roots are indicated by dots.

For N subtrees, each subtree n_i will contain at least 1 tip, and the number of tips T is given by

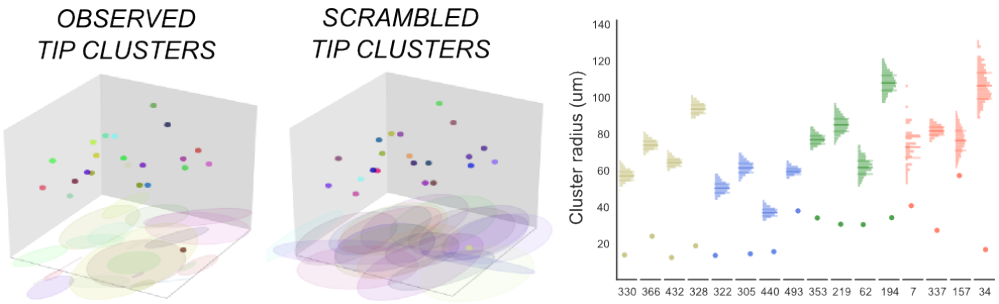
$$T = \sum_{i=1}^{n_i^B} 1$$

Where n_i^B is the number of branches of subtree i. The standard deviations of the tip positions in the $X-Y$ plane are shown as ellipses.

We determined tip distributions within each subtree were random by the following algorithm:

1. For each subtree, note how many tips it contains and the root position (x_r, y_r, z_r)
2. Add the displaced tip position $(x_t - x_r, y_t - y_r, z_t - z_r)$ to the list of tips
3. Randomnly assign tips to each subtree root, insisting that each subtree receive the same number of tips it had originally

This scrambling process is repeated as many times as possible, either $\binom{N_S}{N_S}$ where N_S is the number of subtrees (without repetition) or 2000, whichever is smaller. As a sample metric, I find the center of mass of each subtree and find the mean distance to each subtree's new displaced tips. This is averaged for each "neuron" instance and plotted. Note the large degree of tip cluster overlap and the much larger cluster radii on average.



Code snippet: randomizing subtrees

```
def randomize_subtrees(locations, N=2000, keep=0):
  # Assign tip clusters to different subtrees; repeat N times and send to
```

```
#   subtree_statistic for each tree.
from scipy.special import comb
# Condition the tree -- make each tip clustered mean-centered
import copy
tiplocs, rootlocs = [], []
for k in locations.keys():
  for tip in locations[k]['tips']:
    tiplocs.append([tip[0]-locations[k]['root'][0],
                    tip[1]-locations[k]['root'][1],
                    tip[2]-locations[k]['root'][2]])
  rootlocs.append([locations[k]['root'], len(locations[k]['tips'])]) # Location of tip AND how many it

# An original root with 7 tips can get ANY 7 new random tips, but only 7
# If there are >5 times fewer possible options than requested trials
numcombs = comb(len(rootlocs), len(rootlocs), repetition=False)
if N/5. > numcombs:
  print('Requested␣%i␣trials␣but␣only␣%.2f␣combinations␣are␣possible'
        %(N, float(numcombs)))
  N = int(numcombs*5)

# Run N times
stat, ret, retain = [], [int(i) for i in np.random.rand(keep)], []
for n in range(N):
  if n%200==0:
    print('%i␣/␣%i␣subtrees␣constructed␣so␣far' %(n, N))

  # Generate the dictionary
  trialtips = [int(i) for i in np.random.random(len(tiplocs))*len(tiplocs)] # Num tips
  trialroots = [int(i) for i in np.random.random(len(rootlocs))*len(rootlocs)] # Num roots
  # Make the randomized test dictionary
  this_dict = {}
  for t in range(len(trialroots)): # For each root (NOT each tip)
    this_dict[t] = {'root': rootlocs[t][0],
                    'tips': [[i[0]+rootlocs[t][0][0], # Already root-mean-centered,
                              i[1]+rootlocs[t][0][1], # So add it back
                              i[2]+rootlocs[t][0][2]] for i in
                              # Only use the next tips, however many this cluster should have
                              [tiplocs[o] for o in trialtips[0:rootlocs[t][1]]]]}
                              #tiplocs[trialtips[0]:trialtips[rootlocs[t][1]]]]}

    # Pop the used nodes from trial tips
    for u in range(rootlocs[t][1]):
      _ = trialtips.pop(0)
  if n in ret:
    retain.append(this_dict)

  # Get the test statistic for this new dict
  try:
    stat.append(subtree_statistics(this_dict))
```

```
  except:
    pass
if keep > 0:
  return stat, retain
return stat
```