GUIDE FOR MVT ON MANIFOLDS PYTHON CODE

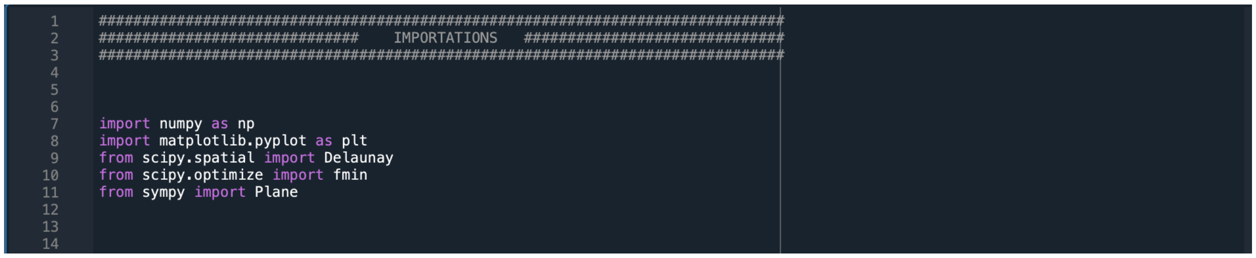
Connor Painter

The purpose of this document is to guide a new or returning researcher through the progress made implementing Python methods to refine the non-contact set in solutions to the obstacle problem with Green's function defined on a cylinder. I'll show you around each of the various functions, detail how they work, and suggest some exercises to try. The code is accessible in the "`MVT on Manifolds 2.py`" and inspiration is taken from the following paper, among others:

A shape optimization approach for simulating contact of elastic membranes with rigid obstacles, Sharma and Rangarajan

Let's begin at the top of the .py document.

IMPORTATIONS

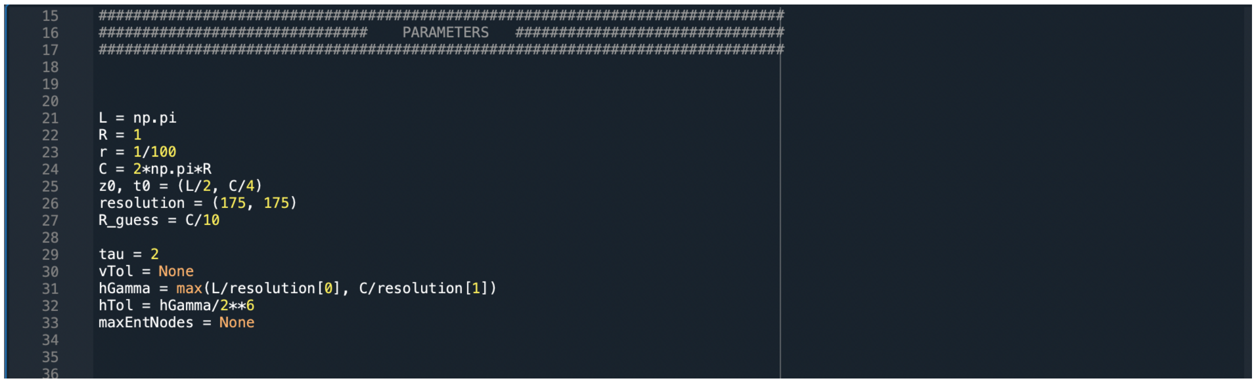


```
################################################################################
#############################    IMPORTATIONS   ################################
################################################################################



import numpy as np
import matplotlib.pyplot as plt
from scipy.spatial import Delaunay
from scipy.optimize import fmin
from sympy import Plane


```

Python documents come with a limited number of built-in functions, but to do most scientific computing you'll need to import packages of methods. `numpy` is a critically important package to any numerical calculations and, thankfully, its methods appear to mimic those of MATLAB. For most scientific researchers using Python, `matplotlib.pyplot` is the go-to package for visualizing data. It has an immense library of possibilities and just might surprise you with its versatility. `scipy` is another hugely popular package with more specialized functions, which we will use for finite element analysis and function optimization. Finally, `sympy` is a package to help make geometry more intuitive.

Importing a package "`as`" another name allows you to reference that package by the abbreviation of your choice. `np` and `plt` are standard abbreviations for their respective packages.

PARAMETERS



```
##############################################################################
##############################    PARAMETERS   ###############################
##############################################################################



L = np.pi
R = 1
r = 1/100
C = 2*np.pi*R
z0, t0 = (L/2, C/4)
resolution = (175, 175)
R_guess = C/10

tau = 2
vTol = None
hGamma = max(L/resolution[0], C/resolution[1])
hTol = hGamma/2**6
maxEntNodes = None
```

In the PARAMETERS section, you'll find definitions of global variables which act as important parameters throughout the document. Whenever you'd like to change something about the shape optimization, all of your parameters can be found in one place. These variables are "visible" inside all the functions we will create in the document, so they can be referenced anywhere. If you introduce more parameters, consider defining them in this space. Let's see what these variables mean:

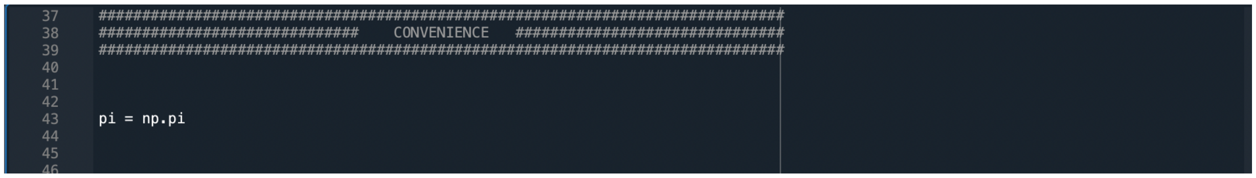
- `L` - the length of the cylinder (don't change this)
- `R` - the radius of the cylinder
- `r` - variable in the differential equation for the solution to the obstacle problem in the non-contact set. I.e., controls the "force" pushing up on the elastic membrane.
- `C` - the circumference of of the cylinder
- `z0, t0` - the (z, θ) coordinates of the singularity in Green's function defined on the cylinder
- `resolution` - the number of grid points (along each axis) at which to compute values of Green's function, the signed distance function, and solutions to the differential equations. Total number of points is approximately `resolution[0]*resolution[1]`.
- `R_guess` - we will initially guess that the shape of the noncontact region is a circle. Its radius is manipulable with this parameter.

The other parameters have not yet been implemented, as they are involved with the process of refining the shape of the noncontact set, which is left to you! Detailed explanations of these parameters can be found in Sharma and Rangarajan.

<u>Interact</u>

- Compile the code. If you're using Spyder 5, like I do, That's the green play button.
- In your console, type any of the parameter names and Return. Are they reading their correct values?
- Change a parameter value by deleting the old value from the document and typing a new value. Recompile, then make sure that the value has actually changed.
- Ask Python the question "`C == 2*pi*R`". Does it return what you expected?

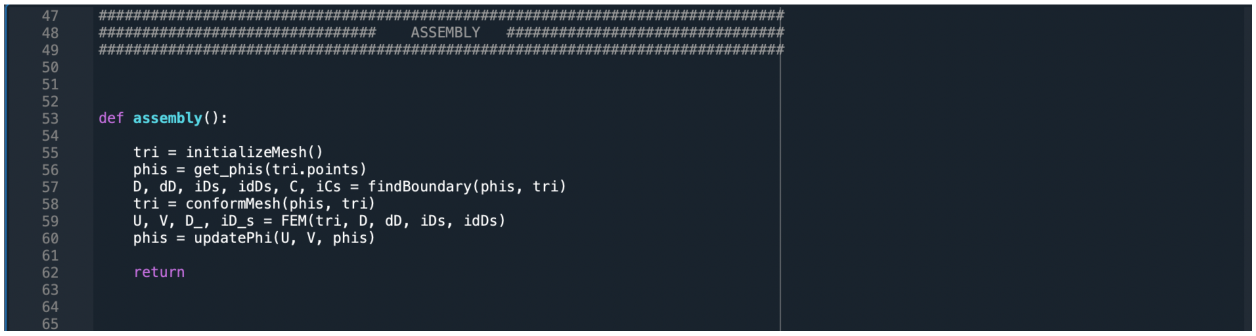
CONVENIENCE



```
###############################################################################
##############################    CONVENIENCE   ###############################
###############################################################################



pi = np.pi

```

I needed to write π a lot, so I made it two characters instead of five.

ASSEMBLY



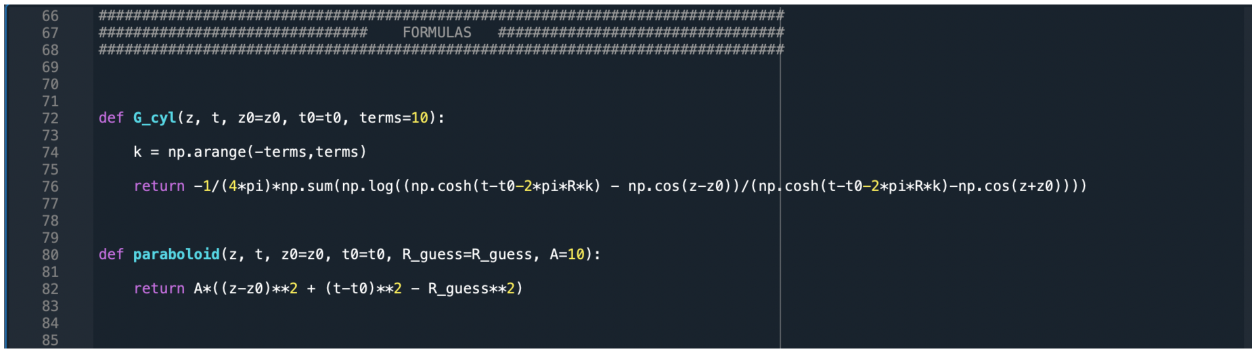
```
###############################################################################
################################    ASSEMBLY   ################################
###############################################################################



def assembly():

    tri = initializeMesh()
    phis = get_phis(tri.points)
    D, dD, iDs, idDs, C, iCs = findBoundary(phis, tri)
    tri = conformMesh(phis, tri)
    U, V, D_, iD_s = FEM(tri, D, dD, iDs, idDs)
    phis = updatePhi(U, V, phis)

    return

```

The `assembly()` function should execute the main steps of the shape optimization algorithm in the correct sequence. It's not even close to finished! Some of the methods are currently just trivial (they return what is inputted) and for the finished product you will likely need a `while` loop over the methods that should be iterated until satisfactory shape resolution is achieved. I think the general framework is correct for the first iteration, at least.

<u>Interact</u>

- In your console, try calling `assembly()` and observe the figures currently being produced. Vary the parameter `r` and try to discern what changes in these figures (one figure will remain unchanged).

FORMULAS



```
################################################################################
###############################    FORMULAS   ##################################
################################################################################



def G_cyl(z, t, z0=z0, t0=t0, terms=10):

    k = np.arange(-terms,terms)

    return -1/(4*pi)*np.sum(np.log((np.cosh(t-t0-2*pi*R*k) - np.cos(z-z0))/(np.cosh(t-t0-2*pi*R*k)-np.cos(z+z0))))


def paraboloid(z, t, z0=z0, t0=t0, R_guess=R_guess, A=10):

    return A*((z-z0)**2 + (t-t0)**2 - R_guess**2)


```

In this section, you'll find two important formulas which define Green's function and a paraboloid on a cylinder. The former is Dr. LeCrone's masterpiece and the latter will define the initial signed distance function ϕ, designed to be negative in the noncontact region and positive in the contact region. Both functions contain necessary and optional parameters. The necessary parameters z, t are the coordinates at which you'd like to know the value of the function. z0, t0 are, again, the location of the singularity in Green's function. By default, they are the values from the PARAMETERS section, but can be changed locally as well for testing purposes.

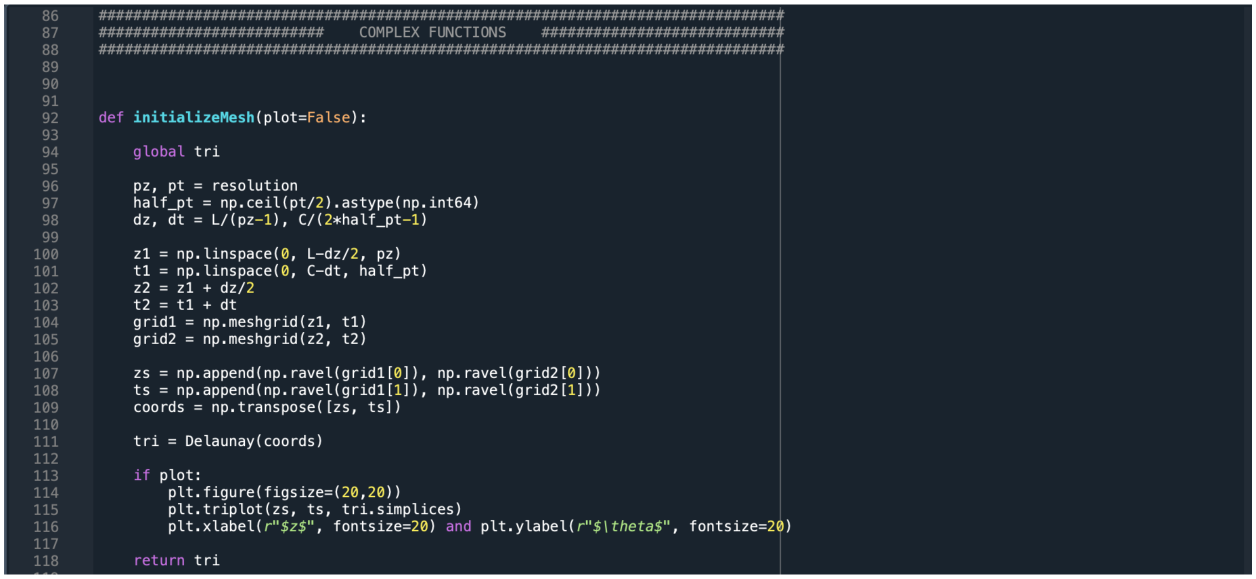
Interact

- Ask for the value of Green's function at some various locations on the cylinder. You may want to use `L` and `C`. Make sure to try asking for the value at the singularity, too.
- Do the same thing for the `paraboloid` function.
- In the `G_cyl` function, what is the purpose of the variable `k`? Try exploring the `np.arange` function by typing `np.arange(-10,10)`. Now, do you see `np.sum` in the return statement? Try to make sense of what's going on here.
- Try to compute 5^3, $\cosh(\pi)$, and $\frac{C}{2\pi}$ in the console.

COMPLEX FUNCTIONS

These functions do the heavy lifting in this project. When I say "COMPLEX", I mean they took me a while to figure out how to code properly and they do rather intricate tasks. No imaginary numbers in this project, yet! Let's go through them one-by-one.

`initializeMesh(plot=False)`



```
 86  ###############################################################################
 87  ##########################    COMPLEX FUNCTIONS    ############################
 88  ###############################################################################
 89
 90
 91
 92  def initializeMesh(plot=False):
 93
 94      global tri
 95
 96      pz, pt = resolution
 97      half_pt = np.ceil(pt/2).astype(np.int64)
 98      dz, dt = L/(pz-1), C/(2*half_pt-1)
 99
100      z1 = np.linspace(0, L-dz/2, pz)
101      t1 = np.linspace(0, C-dt, half_pt)
102      z2 = z1 + dz/2
103      t2 = t1 + dt
104      grid1 = np.meshgrid(z1, t1)
105      grid2 = np.meshgrid(z2, t2)
106
107      zs = np.append(np.ravel(grid1[0]), np.ravel(grid2[0]))
108      ts = np.append(np.ravel(grid1[1]), np.ravel(grid2[1]))
109      coords = np.transpose([zs, ts])
110
111      tri = Delaunay(coords)
112
113      if plot:
114          plt.figure(figsize=(20,20))
115          plt.triplot(zs, ts, tri.simplices)
116          plt.xlabel(r"$z$", fontsize=20) and plt.ylabel(r"$\theta$", fontsize=20)
117
118      return tri
```

The first of the complex functions creates a Delaunay triangulation object `tri` from the `scipy.spatial` class, spanning a cylinder with length and radius set previously in PARAMETERS. Essentially, the function creates lists of evenly spaced numbers in z and θ (lines 96-103), uses them to create grids of points (lines 104-105), then combines the grid and "triangulates" the surface of the cylinder into acute triangular elements (lines 107-111).

Triangulation objects have many important attributes. You should become very comfortable with the documentation at the following link, which details parameters you can pass into the Delaunay function and the attributes to the output object:
https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.Delaunay.html

Notice that I declare `tri` a global variable in the first line of this function. This means that during any particular execution of the code, `tri` is accessible from all other places in the code once `initializeMesh()`. For instance, if in a single execution I call `initializeMesh()`, then call `findBoundary()`, `tri` can be referenced, used, and manipulated inside findBoundary(). I will continue this organizational technique throughout the functions that I wrote, just for the sake of clarity.

The optional `plot` parameter allows you to visualize the triangulated mesh on the "unraveled" cylinder.

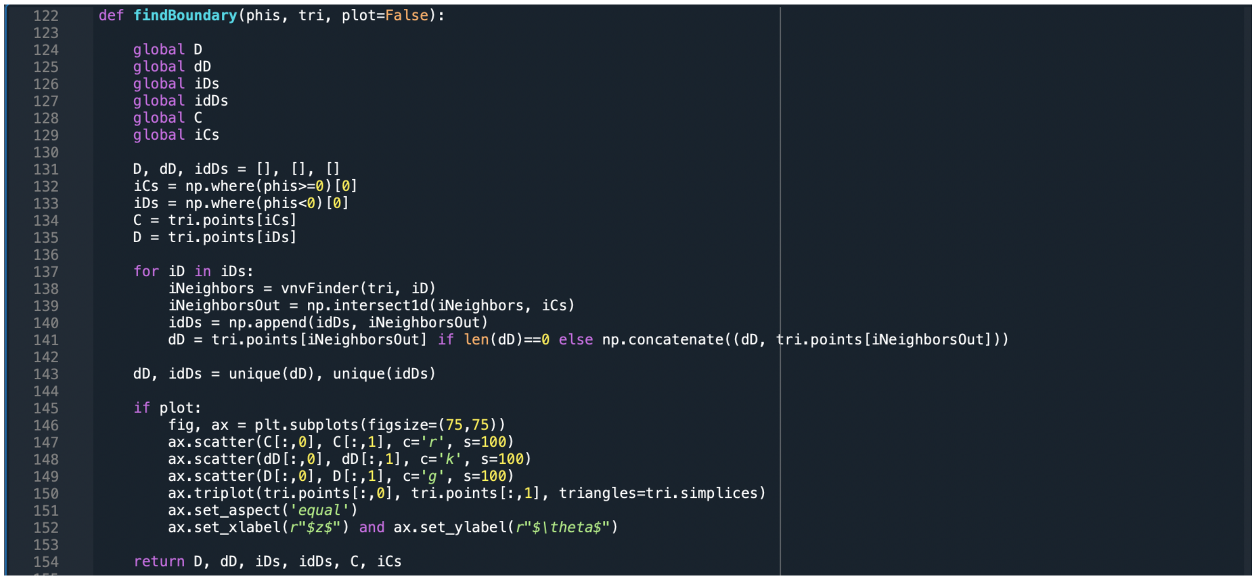
`tri` is returned for convenience with manual manipulation, even though returning it is sort of redundant after declaring it a global variable.

Interact

- In the console, create and store a triangulation object using `initializeMesh()`. (Hint: see line 55 in `assembly`).

- Print out some of the attributes of your triangulation object. If you called it `tri`, like I did, do this by typing `tri.points`, `tri.simplices`, `tri.vertex_neighbor_vertices`, etc. The attributes that I listed were the most important ones to me, thus far. If you want to learn about the others, absolutely go for it!
    - Further understand the dimensions of the attributed arrays by asking for their *shape*. I use numpy's shape function endlessly: https://numpy.org/doc/stable/reference/generated/numpy.shape.html
- Compare the number of points in the triangulation with the expected number, given your `resolution` parameter. Any differences come from the fact that the mesh is not rectangular, but triangular. I'm sure there's a better way to code it, but at least I got close!
- Test out the optional `plot` parameter by asking for `initializeMesh(True)` or `initializeMesh(plot=True)`. Change the `resolution` and try again!
- Create your own custom triangulation object by following the structure of the Example within the Delaunay documentation page. Create your list of points, triangulate them, and plot them.
- In a new Python document, create two functions `first()` and `second()`. In `first()`, declare an integer variable `a = 143`. Then, in `second()`, try to print out that variable with `print(a)`. Call both functions in sequence in the same execution. Now, declare `a` a global variable by changing the line to `global a = 143`. Try executing both functions in sequence again.

`findBoundary(phis, tri, plot=False)`



```python
def findBoundary(phis, tri, plot=False):

    global D
    global dD
    global iDs
    global idDs
    global C
    global iCs

    D, dD, idDs = [], [], []
    iCs = np.where(phis>=0)[0]
    iDs = np.where(phis<0)[0]
    C = tri.points[iCs]
    D = tri.points[iDs]

    for iD in iDs:
        iNeighbors = vnvFinder(tri, iD)
        iNeighborsOut = np.intersect1d(iNeighbors, iCs)
        idDs = np.append(idDs, iNeighborsOut)
        dD = tri.points[iNeighborsOut] if len(dD)==0 else np.concatenate((dD, tri.points[iNeighborsOut]))

    dD, idDs = unique(dD), unique(idDs)

    if plot:
        fig, ax = plt.subplots(figsize=(75,75))
        ax.scatter(C[:,0], C[:,1], c='r', s=100)
        ax.scatter(dD[:,0], dD[:,1], c='k', s=100)
        ax.scatter(D[:,0], D[:,1], c='g', s=100)
        ax.triplot(tri.points[:,0], tri.points[:,1], triangles=tri.simplices)
        ax.set_aspect('equal')
        ax.set_xlabel(r"$z$") and ax.set_ylabel(r"$\theta$")

    return D, dD, iDs, idDs, C, iCs
```

This function figures out which nodes in the triangulation can be considered part of the non-contact set/domain D, the boundary ∂D, and the contact set C. Lists of all points in each of these subdomains are found in global variables `D`, `dD`, and `C`. Indices that locate these points within the entire list of points `tri.points` are found in global variables `iDs`, `idDs`, and `iCs`.

`findBoundary` takes as required inputs `phis` and `tri`, the latter of which should be our friend the Delaunay triangulation object. `phis` is the list of values of the signed distance function ϕ at each node of the triangulation, in the correct order. In other words, `phis[n]` should equal
ϕ(`tri.points[n]`). The function should be general enough to accept `phis` and `tri` after every iteration of updating, as `phis` will be changed in `updatePhi` and tri will be changed in `conformMesh`.

Essentially, the function uses `np.where` to locate the indices at which `phis` has values greater or less than zero. Then, to define a boundary, it loops through each node in `D`, retrieves its immediate neighboring nodes using the `vertex_neighbor_vertices` attribute, and isolates the neighboring nodes which are *not* a part of `D` (or equivalently a part of `C`). It's possible for the loop to identify the same boundary point more than once, so line 143 takes care of duplicates.

The optional `plot` parameter again cleanly visualizes the each determined region of the cylinder.

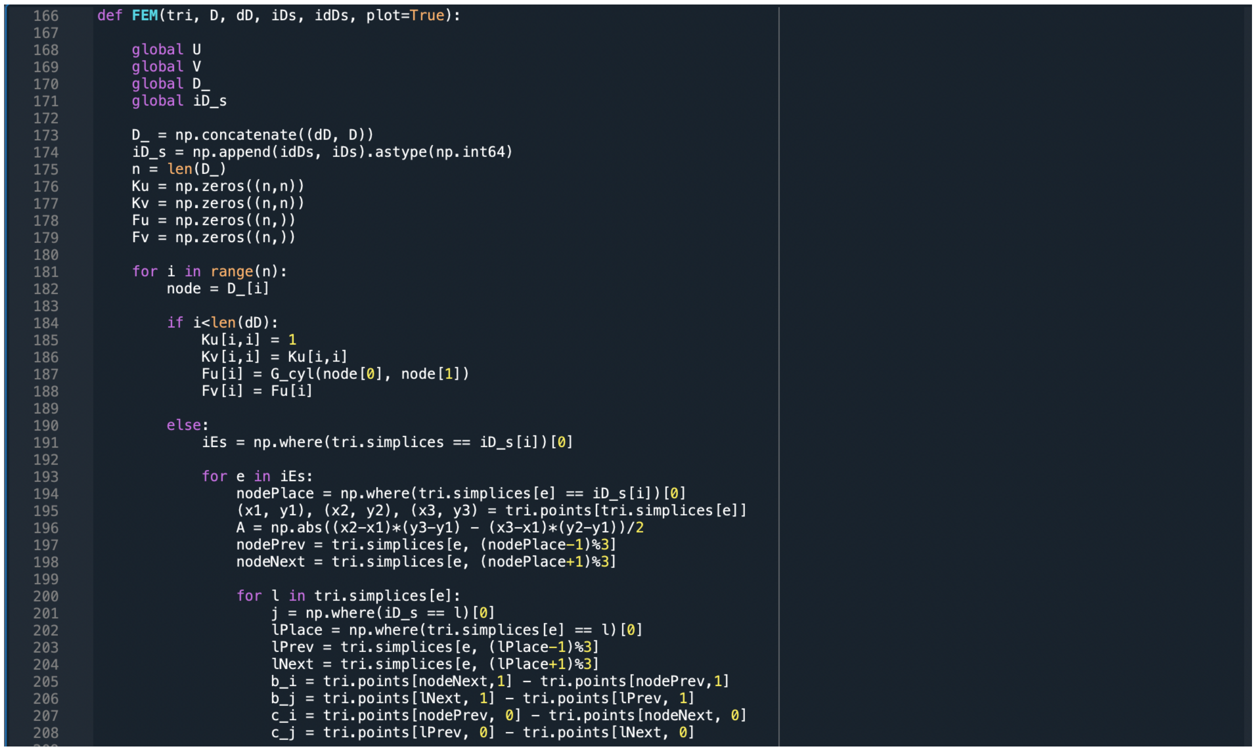
<u>Interact</u>

- Copy lines 55-57 from assembly and execute them in the console, either one by one or all together. Make one minor change: go ahead and set `plot=True`.
- Observe the output visual. The red dots are located at nodes in C, green dots in D, and black dots in ∂D. If you're using Spyder, make sure to uncheck "Fit plots to window" or just Undock the figures so that you can zoom in and out.
- Take a look at each of the new variables (`D`, `dD`, `idDs`, etc.) to make sense of them.
- Experiment with numpy's where function a bit. Create an array `x = np.arange(-10,10)` and ask `np.where(x < 0)`. Now ask for `np.where(x < 0, x, "Positive")`. Reshape the array with `x = x.reshape(5,4)`. Ask the same questions. Useful, huh?

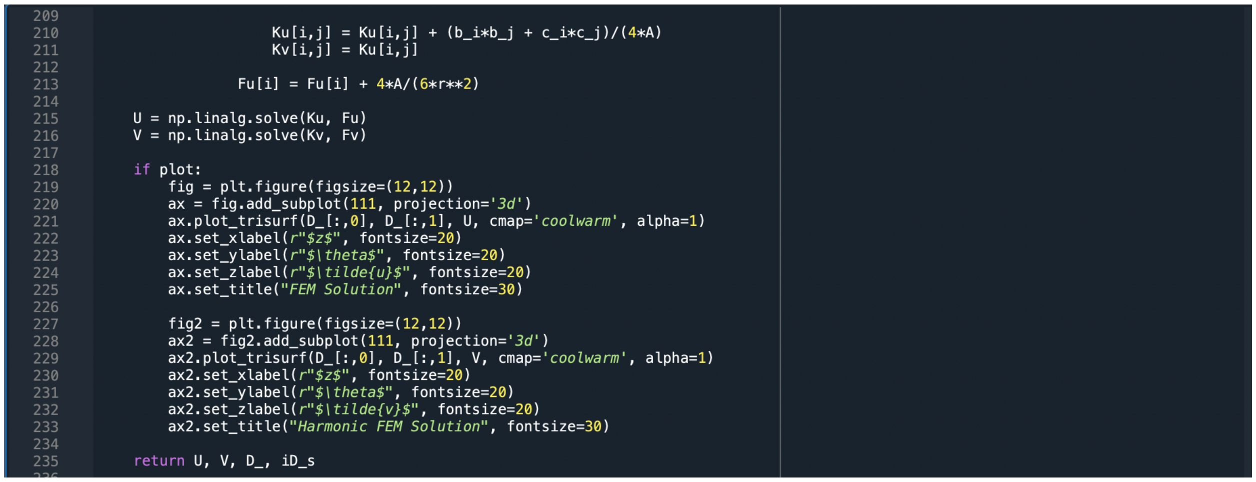
`conformMesh(phis, tri)`

The first complex function that you'll have to construct is one which adjusts the locations of the boundary nodes in the triangulation so that they approximate the true boundary a little better, but algorithmically preserves the "quality" of the other elements as best as possible. The inputs are just a suggestion/prediction; you might need to change them.

`FEM(tri, D, dD, iDs, idDs, plot=True)`



```
166  def FEM(tri, D, dD, iDs, idDs, plot=True):
167  
168      global U
169      global V
170      global D_
171      global iD_s
172  
173      D_ = np.concatenate((dD, D))
174      iD_s = np.append(idDs, iDs).astype(np.int64)
175      n = len(D_)
176      Ku = np.zeros((n,n))
177      Kv = np.zeros((n,n))
178      Fu = np.zeros((n,))
179      Fv = np.zeros((n,))
180  
181      for i in range(n):
182          node = D_[i]
183  
184          if i<len(dD):
185              Ku[i,i] = 1
186              Kv[i,i] = Ku[i,i]
187              Fu[i] = G_cyl(node[0], node[1])
188              Fv[i] = Fu[i]
189  
190          else:
191              iEs = np.where(tri.simplices == iD_s[i])[0]
192  
193              for e in iEs:
194                  nodePlace = np.where(tri.simplices[e] == iD_s[i])[0]
195                  (x1, y1), (x2, y2), (x3, y3) = tri.points[tri.simplices[e]]
196                  A = np.abs((x2-x1)*(y3-y1) - (x3-x1)*(y2-y1))/2
197                  nodePrev = tri.simplices[e, (nodePlace-1)%3]
198                  nodeNext = tri.simplices[e, (nodePlace+1)%3]
199  
200                  for l in tri.simplices[e]:
201                      j = np.where(iD_s == l)[0]
202                      lPlace = np.where(tri.simplices[e] == l)[0]
203                      lPrev = tri.simplices[e, (lPlace-1)%3]
204                      lNext = tri.simplices[e, (lPlace+1)%3]
205                      b_i = tri.points[nodeNext,1] - tri.points[nodePrev,1]
206                      b_j = tri.points[lNext, 1] - tri.points[lPrev, 1]
207                      c_i = tri.points[nodePrev, 0] - tri.points[nodeNext, 0]
208                      c_j = tri.points[lPrev, 0] - tri.points[lNext, 0]
```



```
209  
210                      Ku[i,j] = Ku[i,j] + (b_i*b_j + c_i*c_j)/(4*A)
211                      Kv[i,j] = Ku[i,j]
212  
213                  Fu[i] = Fu[i] + 4*A/(6*r**2)
214  
215      U = np.linalg.solve(Ku, Fu)
216      V = np.linalg.solve(Kv, Fv)
217  
218      if plot:
219          fig = plt.figure(figsize=(12,12))
220          ax = fig.add_subplot(111, projection='3d')
221          ax.plot_trisurf(D_[:,0], D_[:,1], U, cmap='coolwarm', alpha=1)
222          ax.set_xlabel(r"$z$", fontsize=20)
223          ax.set_ylabel(r"$\theta$", fontsize=20)
224          ax.set_zlabel(r"$\tilde{u}$", fontsize=20)
225          ax.set_title("FEM Solution", fontsize=30)
226  
227          fig2 = plt.figure(figsize=(12,12))
228          ax2 = fig2.add_subplot(111, projection='3d')
229          ax2.plot_trisurf(D_[:,0], D_[:,1], V, cmap='coolwarm', alpha=1)
230          ax2.set_xlabel(r"$z$", fontsize=20)
231          ax2.set_ylabel(r"$\theta$", fontsize=20)
232          ax2.set_zlabel(r"$\tilde{v}$", fontsize=20)
233          ax2.set_title("Harmonic FEM Solution", fontsize=30)
234  
235      return U, V, D_, iD_s
```

FEM stands for Finite Element Method, so this function provides the artillery to solve complicated differential equations in meshed regions using linear algebra. `FEM` takes as input the triangulation `tri`, the non-contact set `D`, the boundary `dD`, and the locators `iDs` and `idDs`. It uses this information to find the solution to the obstacle problem `U` and the solution to the harmonic equation `V`.

`FEM` starts by concatenating the lists of boundary nodes and non-contact nodes into `D_`, which is supposed to resemble “the closure of D” (lines 173-174). The remainder of the function is focused on populating the *shape matrices* `Ku`, `Kv` and *force matrices* `Fu`, `Fv` with the correct values.

This code was mostly created by Dr. LeCrone in MATLAB and adapted to Python by me, so he is by far the most knowledgeable about the details here. Notice, though, the overarching nested loop structure. The function loops through every index `i` corresponding to points in `D_`. If `i` corresponds with a boundary point, it populates the corresponding elements of the matrices a certain way (lines 184-188). However, if `i` corresponds to a non-contact point, the corresponding elements of the matrices are populated a different way (lines 190-213). Eventually, the matrices are used to solve two matrix equations, as is traditionally done in Finite Element Method, yielding solutions to two different differential equations related to the obstacle problem.

Use the optional `plot` input to visualize both solutions.

`updatePhi(U, V, phis)`

The second function you'll need to create has to take as input the solutions to the differential equations and make algorithmic adjustments to the signed distance function ϕ to refine the approximation of the boundary of the non-contact set (hence the title "shape optimization" from Sharma-Rangarajan). The last four functions should be compatible in an iterative, looping format, general enough to accept and handle each updated `phis` and `tri`.

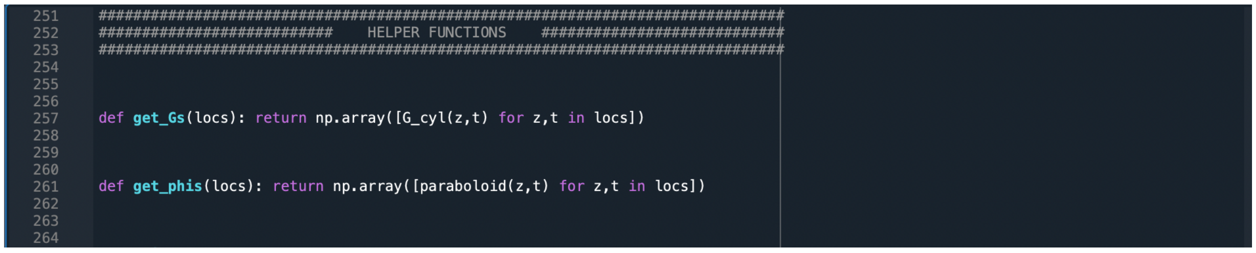
Interact

- Call `assembly()` again and ponder the sequence in which the functions are currently executed. With Dr. LeCrone, discuss the modifications to the overall structure of the assembly necessary to complete an iterative, looping optimization process (instead of a single initialization). The loop will probably be a while loop, executing until a certain condition or level of tolerance is satisfied.

HELPER FUNCTIONS

Helper functions are quicker algorithms that will be useful in coding the complex functions. They make a task easier by reducing a common problem into a few short inputs.

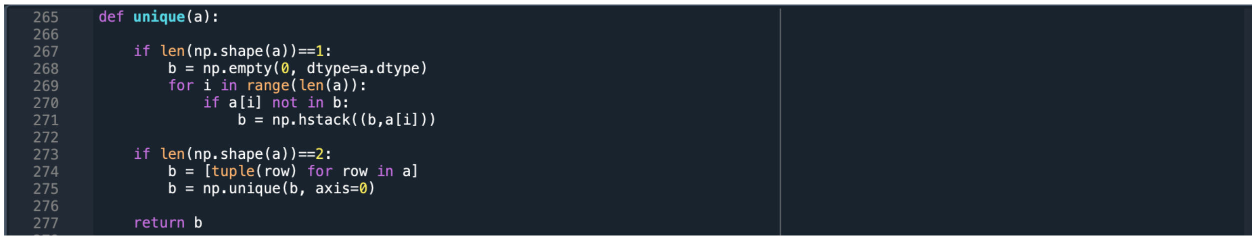
`get_Gs(locs)` and `get_phis(locs)`



```
################################################################################
###########################    HELPER FUNCTIONS    #############################
################################################################################



def get_Gs(locs): return np.array([G_cyl(z,t) for z,t in locs])


def get_phis(locs): return np.array([paraboloid(z,t) for z,t in locs])


```

These functions take as input a list of locations on the cylinder and outputs the values of Green's function or paraboloid at those locations using list comprehension (fast!).

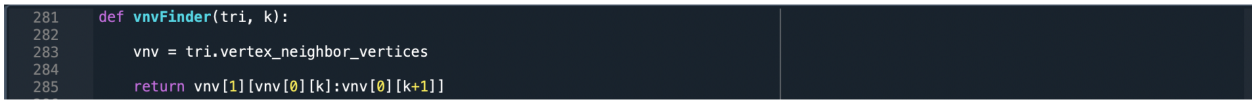
`unique(a)`



```python
def unique(a):

    if len(np.shape(a))==1:
        b = np.empty(0, dtype=a.dtype)
        for i in range(len(a)):
            if a[i] not in b:
                b = np.hstack((b,a[i]))

    if len(np.shape(a))==2:
        b = [tuple(row) for row in a]
        b = np.unique(b, axis=0)

    return b
```

Takes a 1 (2)-dimensional array `a` and outputs an array of equal dimension with duplicate elements (rows) removed. There's a numpy routine for this (numpy.unique), but I found it difficult to figure out, so I drafted up my own. Feel free to use my method or the one made by experts at your leisure!

`vnvFinder(tri, k)`



```python
def vnvFinder(tri, k):

    vnv = tri.vertex_neighbor_vertices

    return vnv[1][vnv[0][k]:vnv[0][k+1]]
```

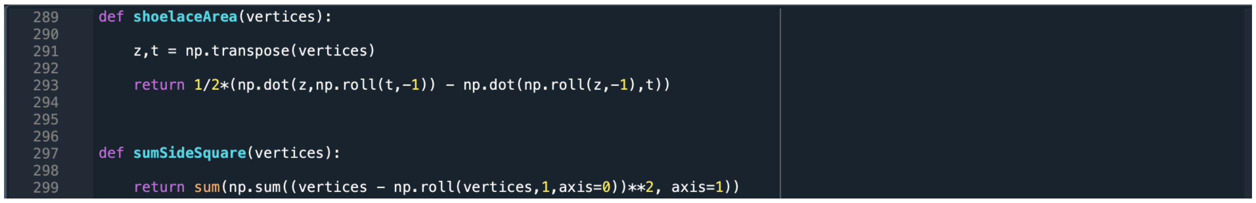
Delaunay triangulation objects have a useful attribute called `vertex_neighbor_vertices` which is simple in theory but difficult to code. The method allows you to input a particular node index and get the indices of all neighboring nodes. Luckily, `scipy` has taken care of most of the hard work for us, but they left some gymnastics for the user. Luckily for *you*, I've taken care of the rest of those gymnastics so that you can use the feature quickly.

Input the triangulation object `tri` and the index `k` of the node in question. Outputted are the indices of the neighboring vertices.

<u>Interact</u>

- Create and store a triangulation object `tri` using `initializeMesh()`. Use `vnvFinder()` to find the indices of the neighbor vertices to vertex number 1000. Print `tri.points[1000]` and `tri.points[vnvFinder(tri, 1000)]`.

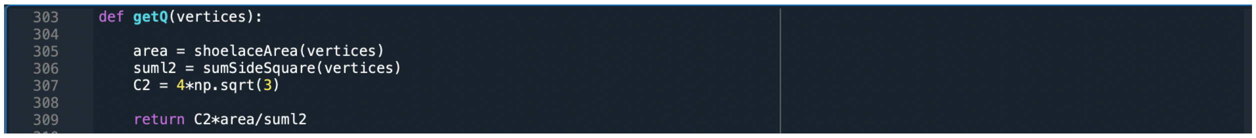
`shoelaceArea(vertices)` and `sumSideSquare(vertices)`



```
def shoelaceArea(vertices):

    z,t = np.transpose(vertices)

    return 1/2*(np.dot(z,np.roll(t,-1)) - np.dot(np.roll(z,-1),t))


def sumSideSquare(vertices):

    return sum(np.sum((vertices - np.roll(vertices,1,axis=0))**2, axis=1))
```

Blazing fast algorithms to find the *signed area* and *sum of squared side lengths* of a simple polygon given the coordinate locations of its vertices. Input vertices as a 2-dimensional array and get the quantities you want. It's that simple. I used these when coding the next few helper methods, so these are helpers to helpers.

`getQ(vertices)`



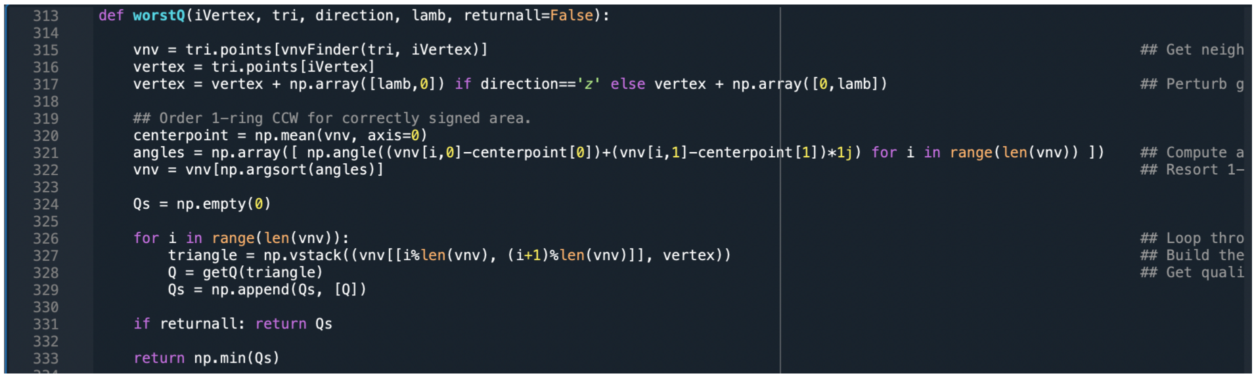
```
def getQ(vertices):

    area = shoelaceArea(vertices)
    suml2 = sumSideSquare(vertices)
    C2 = 4*np.sqrt(3)

    return C2*area/suml2
```

Gets the "quality factor" Q of a triangle, a sort of measure of the area-to-perimeter ratio of the triangle. A high quality triangle is close to equilateral while a low quality triangle is very scalene/obtuse. This method is necessary in conjunction with the next few in updating the triangulation in an algorithmically clean way.

Interact

- Create any set of three vertices `v` defining a triangle (as a 3 x 2 matrix). Call `shoelaceArea(v)`, `sumSideSquare(v)`, and `getQ(v)`. Do this with a few triangles and get a sense for the functions.

`worstQ(iVertex, tri, direction, lamb, returnall=False)`

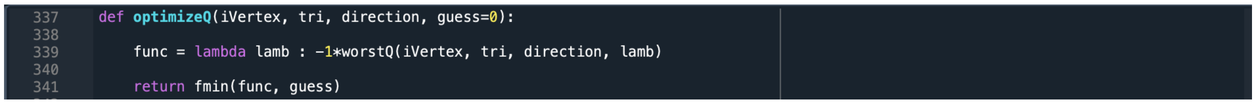


```
def worstQ(iVertex, tri, direction, lamb, returnall=False):

    vnv = tri.points[vnvFinder(tri, iVertex)]                                                                      ## Get neigh
    vertex = tri.points[iVertex]
    vertex = vertex + np.array([lamb,0]) if direction=='z' else vertex + np.array([0,lamb])                        ## Perturb g

    ## Order 1-ring CCW for correctly signed area.
    centerpoint = np.mean(vnv, axis=0)
    angles = np.array([ np.angle((vnv[i,0]-centerpoint[0])+(vnv[i,1]-centerpoint[1])*1j) for i in range(len(vnv)) ])   ## Compute a
    vnv = vnv[np.argsort(angles)]                                                                                   ## Resort 1-

    Qs = np.empty(0)

    for i in range(len(vnv)):                                                                                       ## Loop thro
        triangle = np.vstack((vnv[[i%len(vnv), (i+1)%len(vnv)]], vertex))                                           ## Build the
        Q = getQ(triangle)                                                                                          ## Get quali
        Qs = np.append(Qs, [Q])

    if returnall: return Qs

    return np.min(Qs)
```

Pretty dang cool little method that may be my most treasured piece of code that I created in this document. Input the index of any node `iVertex`, the triangulation `tri`, then the `direction` (‘z’ or ‘t’) and distance `lamb` that you’d like to perturb the vertex. The method temporarily changes the location of some node in the triangulation and calculates the quality factor of each element containing that node, i.e. each element in the *1-ring*. The lowest quality factor calculated among those elements is returned. I know it sounds niche, but it turns out to be useful for adjusting the triangulation in smart ways.

What the heck am I doing in this method? It’s a little complicated and unenlightening to explain, but I’ve left my original personal comments in the code document for future reference. Hopefully the next function makes the idea a little clearer.

`optimizeQ(iVertex, tri, direction, guess=0)`



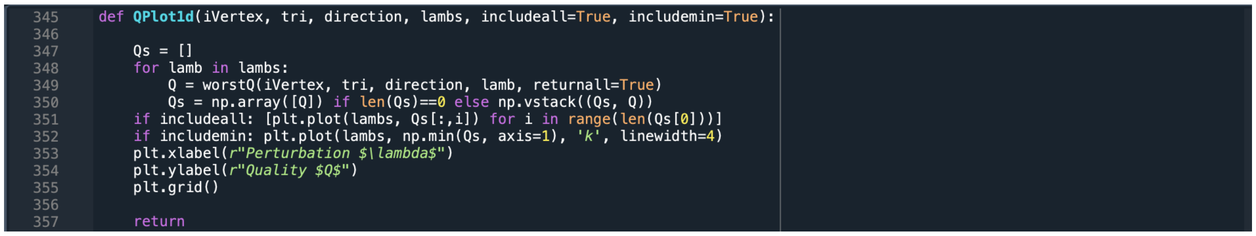
```python
def optimizeQ(iVertex, tri, direction, guess=0):

    func = lambda lamb : -1*worstQ(iVertex, tri, direction, lamb)

    return fmin(func, guess)
```

Now, I’m using `worstQ` to my advantage with `fmin`, a `scipy` function used to find a local minimum of a continuous function. As required by `fmin`, I create a quick function using `worstQ`, fixing `iVertex`, `tri`, and `direction`, but allowing `lamb` to fluctuate. I multiply in -1 so that my `fmin` is really an “fmax” (does not exist, don’t try to find it in `scipy`). `fmin` then uses an optimization algorithm to determine the local extremum of the function I pass in, which should be somewhere around the optional parameter `guess`.

Think about what this is doing for a second: we’re perturbing the location of a node in the triangulation to figure out the location at which the *worst quality triangle in the 1-ring is at a maximum*. It is in this configuration that the elements are closest to equilateral, in total.

This will be useful in `conformMesh()`. You’ll be moving vertices around to best fit an estimation of the non-contact boundary, then “controlling the damage” to the local equilateral-ness of elements, requiring slight changes to elements in the entire mesh.

`QPlot1d(iVertex, tri, direction, lambs, includeAll=True, includeMin=True)`



```python
def QPlot1d(iVertex, tri, direction, lambs, includeall=True, includemin=True):

    Qs = []
    for lamb in lambs:
        Q = worstQ(iVertex, tri, direction, lamb, returnall=True)
        Qs = np.array([Q]) if len(Qs)==0 else np.vstack((Qs, Q))
    if includeall: [plt.plot(lambs, Qs[:,i]) for i in range(len(Qs[0]))]
    if includemin: plt.plot(lambs, np.min(Qs, axis=1), 'k', linewidth=4)
    plt.xlabel(r"Perturbation $\lambda$")
    plt.ylabel(r"Quality $Q$")
    plt.grid()

    return
```

I wanted to visualize this complicated idea, and if you do too, you've come to the right place.

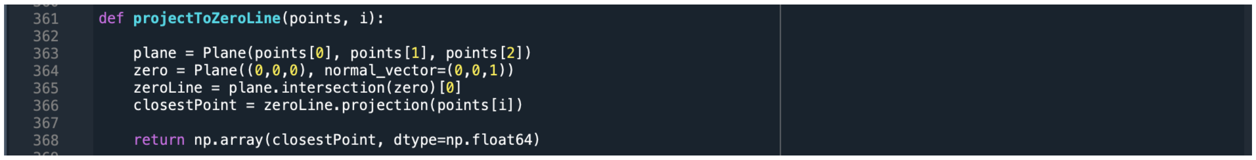
Side note: this is a helper function which doesn't provide elegance to another part of the code, but instead actually helps the user understand what's going on! Feel free to include your own methods of this sort during your coding journey.

Same inputs as `worstQ`, except that you supply a bunch of values lambs at which you'd like to measure the worst quality. In a more human way, you're visually doing what `optimizeQ` is doing. `QPlot1d` can plot the quality of each triangle in the 1-ring at each perturbation value (with optional parameter `includeall`) and can bold the worst quality at each `lamb` (with parameter `includemin`).

<u>Interact</u>

- Get a triangulation `tri` from `initializeMesh()`. Call `QPlot1d`, centralizing node number 1000, perturbing along the 't' axis, with 50 `lamb` values between -0.08 and 0.08. Try to make sense of what you see by adjusting some of these parameters. It's really cool stuff!

`projectToZeroLine(points, i)`



```
def projectToZeroLine(points, i):

    plane = Plane(points[0], points[1], points[2])
    zero = Plane((0,0,0), normal_vector=(0,0,1))
    zeroLine = plane.intersection(zero)[0]
    closestPoint = zeroLine.projection(points[i])

    return np.array(closestPoint, dtype=np.float64)
```

Quick function that does a series of geometric calculations using Python's `sympy` package.

Input `points`, a 3 x 3 array which defines coordinate locations of vertices of a triangle in 3-dimensional $(z, \theta, \phi(z, \theta))$ space. One or two points have $\phi > 0$ while the other(s) have $\phi < 0$. That means the z, θ plane intersects the triangle formed by points, and the intersection is a line segment. Finally, input index `i` and find the closest point projection onto that line segment.

This method should, again, be useful when you code `conformMesh`. I've done my best to give a headstart into the development of that method with this helper and `optimizeQ`.

Disclaimer: it's actually not possible to perform a closest point projection onto a line segment with `sympy`, and I'm not sure you can find a publicly available package with that functionality. What I've done is a closest point projection onto the *line* formed by intersection the whole plane formed by `points` and the z, θ plane.

<u>Interact</u>

- (Advanced) Resolve the disclaimer: modify this method to actually project onto the line segment produced by intersecting the triangle formed by points and the z, θ plane. Do this by first projecting onto the line (as I've already done), then checking to see if your projection is contained within the segment encompassed by the triangle (this will take work). If so, leave the projection be. If not, move the projection to the closer endpoint of the segment.