EN.500.113 Gateway Computing: Python

Project D

Project Due: Friday, December 9 at 23:59 ET

Build Your Own Carbon Nanotube

The last three decades have seen the advent of engineering on the atomic scale: nanotechnology. One ignition point for this explosion of activity was the first imaging of a carbon nanotube by a research group at NEC in Japan in 1991. Their microscopic images generated a great deal of excitement because they revealed that carbon, in addition to forming the backbone of life, was also able to form a large number of other interesting geometric structures that could comprise the building blocks for new materials and structures. By 1996 researchers at Rice University in the US had been able to produce these materials in a controlled manner, opening up the possibility to test the properties of these new carbon structures. The results have been wonderfully surprising. Depending on the structure of these objects they can act as metals or as semiconductors. So, they could potentially play a wide variety of roles in a new generation of electronics built on the molecular scale. They are also extremely stiff and strong per unit weight, leading to the conjecture that they could serve to make new super strong composite materials. While most of these applications remain unrealized, a good deal of active work is going on to bring these materials to market.

Nanotubes have a very interesting structure. Essentially you can think of a nanotube as a rolled-up graphene sheet. Consider the diagram below:
The hexagonal pattern represents the graphene sheet. Each carbon atom sits at a vertex and is bonded to 3 other carbon atoms. To make your nanotube you must cut out a rectangle (the shaded area in the diagram) from the graphene sheet, and then roll it up to create a tube aligned along one of the cuts (T in the diagram). The trick is that you cannot choose just any rectangle; the side that will comprise the mouth of the tube must start at a carbon atom and end at a carbon atom. The result is that there exist an infinite, but countable, number of nanotube structures that can be made. To denote what kind of tube they are considering, scientists draw two vectors denoted a_1 and a_2 in the diagram. Each vector represents a hop from one carbon atom to another. The side that will become the mouth of the tube is determined by two integers n and m. The mouth of the tube

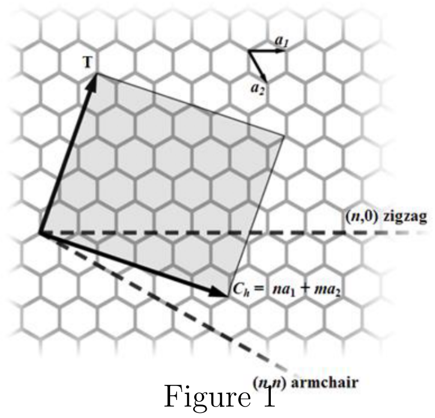

Figure 1

(C_h in the diagram) then extends from the carbon atom at the origin to the one that is reached by making n hops in the a_1 direction and m hops in the a_2 direction. It turns out that all the possible tubes can be represented by combinations of positive integers n and m such that m is less than or equal to n.

From a practical point of view the hexagonal sheet for graphene can be thought of as a rectangular array on which some atoms have been moved a little bit in the y-direction:

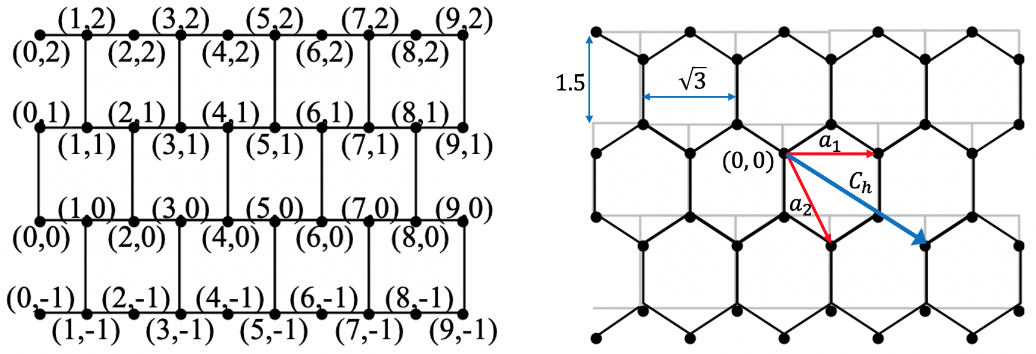

Figure 2

In the above diagram the numbers (p, q) on the left are not meant to be locations in space, but rather a numbering system in which each atom is assigned two integers that uniquely identify that atom. To get regular hexagons on the right the boxes on the left would have to be 1.5 carbon-carbon bonds high and $\sqrt{3}$ carbon-carbon bonds wide.

The Project Parts

This project is described below:

- You are asked to build a graphene sheet and roll it up in order to construct a carbon nanotube. For the graphene sheets, the orientation should be determined as illustrated and described on the previous page with origin at the atom that sits at the leftmost corner of the sheet. For the nanotube, the $z-$axis is passing through the center of the tube while the x-axis pass through the atom that was originally at the origin. All lengths will be measured in units of the carbon-carbon bond length; so, bonded atoms are a distance 1.0 apart.

In order to graphically depict the graphene sheet and the corresponding carbon nanotube you are given the function `plot` (located inside the python script `atomplot`). This function represents atoms as circles (for a graphene sheet) or spheres (for a carbon nanotube), connected by lines that represent the carbon-carbon bonds.

Python scripts to submit

For this project you will write one Python script (`solution.py`) to solve the problems above. I suggest you save the `template.py` script you are given as `solution.py` and start from there.

Description of `template.py`

In this file we have defined: i) a `class` called `CarbonFiller`, ii) a function `Graphene` and iii) the special method `__name__` == "`__main__`" which sets the name `__main__` to the special variable `__name__`.

1 Class `CarbonFiller`.

The `class CarbonFiller` is used in order to create an object of type "CarbonFIller" which has the following class and static methods, that you are asked to write:

METHODS

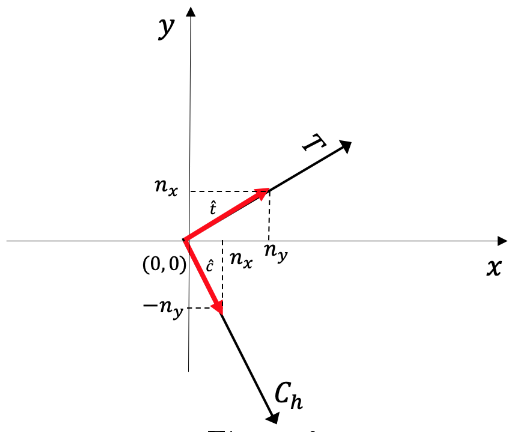

Figure 3

1.1 `__init__`

Constructor of the class. It takes three input parameters:

1. n (type: `int`): number of hops along the a_1 direction.
2. m (type: `int`): number of hops along the a_2 direction.
3. l (type: `int`): length of the sheet or tube as measured in carbon-carbon bond lengths

A CarbonFiller object is created as: `CarbonFiller(2, 1, 1)` and has the attributes `self.n=2, self.m=1, self.l=1, self.name='Carbon Filler'`.

1.2 `vector` (10 points)

This is an **instance** method that will be used for finding the coordinates of the vector C_h, which extends from the carbon at the origin to the one that is reached by making n hops in the a_1 direction and m hops in the a_2 direction (see Fig.1). that describe the orientation of the graphene sheet. The method should return a `numpy.array` of `shape=(2,)`.

```
def vector(self):
        return vec
```

You are required to round each element of this `numpy.array` to **three** decimal places. For example, `CarbonFiller(2, 0, 1).vector()` and `CarbonFiller(2, 1, 1).vector()` should return `[3.464, 0.0]` and `[4.33, -1.5]`, respectively.

1.3 `normVector` (6.5 points)

This is a **static** method of the class will be used to normalize vector C_h by dividing it by its length. The input to this method should be a `numpy.array` storing the coordinates of C_h. It should return **two** things:

1. `norm_vec` (type: `numpy.array of floats`): Coordinates of the normalized vector $\hat{c}$ (see Fig.1).
2. `norm` (type: `float`): Length of vector C_h.

You are required to round each element of this `numpy.array` to **three** decimal places. For example, `CarbonFiller.normVector([4.33, -1.5])` should return `(array([ 0.945, -0.327]), 4.582)`.

```
@staticmethod
def normVector(vec):
        return norm_vec, norm
```

The **@staticmethod** decorator is a built-in function decorator that gets evaluated after your function is defined. A static method does not receive an implicit first argument (i.e., `self`). A static method is bound to the class and not the object of the class and it cannot access or modify the class state.

1.4 `normTvector` (3.5 points)

This is a **static** method that will be used to find the coordinates of the vector perpendicular to a given vector. That being said, this method should take as input a `numpy.array` with the coordinates of vector $\hat{c}$ and return a `numpy.array` with the coordinates of vector $\hat{t}$ $(-n_y, n_x)$ (see Fig.1).

```
@staticmethod
def normTvector(c_hat):
        return t_hat
```

For example, `CarbonFiller.normTvector([0.2, 1.0])` should return `[-1.0, 0.2]`.

1.5 `TVector`(8 points)

This is an **instance** method of the class that will find the coordinates of vector T which is perpendicular to vector C_h, by making use of the methods `normVector()` and `normTvector()` you have created (**Hint**: First you will need to find the coordinates of $\hat{t}$ and then, to multiply them by the length l, in order to obtain the coordinates of T). This method should take **one** positional argument:

1. `Ch` (`numpy.array of floats`): Coordinates of vector C_h.

and return a `numpy.array` of `floats` (`T.shape = (2,)`) with the coordinates of vector T.

```
def TVector(self, Ch):
        return T
```

For example, `CarbonFiller(2, 1, 1).TVector([4.33, -1.5])` should return `[0.327, 0.945]`.

1.6 `pq` (15points)

One way to generate atoms in your graphene sheet is to create atoms in a rectangle that spans beyond the edges of your desired region, and then decide which atoms to include. The purpose of this **static** method to find the min and max values of p and q that will describe a rectilinear sheet in which we can inscribe our rectangle. This method takes two positional arguments:

1. `Ch` (`numpy.array`): array holding the coordinates of vector C_h.
2. `T` (`numpy.array`): array holding the coordinates of vector T.

and returns two `numpy.arrays`, each one of `shape=(2,)`, that hold the min and max values of p and q (of type `int`), respectively (**Hint**: Use functions `math.ceil()` and `math.floor()`).

```
@staticmethod
def pq(Ch, T):
        return [p_min, p_max], [q_min, q_max]
```

For example, `p, v = CarbonFiller.pq([6.928, -3.0], [0.794, 1.836])` should return `p = [0, 9]` and `q=[-2, 2]`.

1.7 `grid_pq` (1 points)

Next, you will need to create the (p, q) grid of points; each point represents an atom i designated by a particular $p_i \in [p_{\min}, p_{\max}]$ and a particular $q_i \in [q_{\min}, q_{\max}]$. To this end, you will use the method `grid_pq` provided to you in the `misc.py` script. This function takes as input two `numpy.arrays`, holding the minimum and maximum values of p and q, respectively (outputs of the `pq()` method), and returns two `2-d nump arrays` holding the **integers** used identify each atom along the p- and $q-$direction, respectively.

```
def grid_pq( p,  q):
        return pg, qg
```

For example: `a, b = grid_pq( [0, 3], [-1, 1])`, gives `a =array([[0, 1, 2, 3], [0, 1, 2, 3], [0, 1, 2, 3], [0, 1, 2, 3]])` and `b =array([[-1, -1, -1, -1], [ 0, 0, 0, 0], [ 1, 1, 1, 1]])` where `a.shape=(3,4)` and `b.shape=(4, 3)`.

1.8 `coordinates` (20 points)

Since you have defined vectors that describe the graphene sheet, you can now determine the coordinates of any atom. This **static** method should take as input two `2-d numpy arrays` that hold the **integers** that identify each atom in the (p, q) grid (output of the `grid_pq()` method). It should return two `2-d numpy arrays` holding the x and y coordinates of each atom.

```
@staticmethod
def coordinates( pg, qg):
        return x, y
```

Again, you are required to round the list of coordinates to **three** decimal places. For example, `x, y = CarbonFiller.coordinates([[0, 1, 2], [0, 1, 2]], [[0, 0, 0], [1, 1, 1, ]])` should return `x=array([[0.   , 0.866, 1.732], [0.   , 0.866, 1.732]])`, `array([[ 0.   , -0.5, 0.  ], [ 1.   , 1.5, 1.  ]])` and `y=array([[ 0.   , -0.5, 0.  ], [ 1.   , 1.5, 1.  ]])`.

1.9 `distance()`(20 points)

Next you need to determine the distance of any atom along the mouth direction and along the length direction. This can be done by taking the dot product (an algebraic operation that takes two coordinate vectors, and returns a single number, i.e. if $\mathbf{a} = [a_1, a_2]$ and $\mathbf{b} = [b_1, b_2]$ then, $\mathbf{a} \cdot \mathbf{b} = a_1 b_1 + a_2 b_2$) of the atom position with the appropriate normal vector describing the mouth and length directions respectively. Thus, the distance along the tube axis can be defined as $t = -n_y \cdot x + n_x \cdot y$ while the distance along the mouth direction is $s = n_x \cdot x + n_y \cdot y$. This **static** method will take as input three positional arguments: the two `2-d numpy arrays` that hold the coordinates x and y of the atoms and the `numpy array` that holds the coordinates of the vector normal vector $\hat{c}$.

```
@staticmethod
def distance( x, y, c_hat):
        return s, t
```

It will return two `2-d numpy arrays` holding the values of s and t. You are required to round the coordinates to **three** decimal places. For example, `CarbonFiller.distance([[0.0, 0.866, 1.732], [0.0, 0.866, 1.732]],[[0.0, -0.5, 0.0], [1.0, 1.5, 1.0]], [0.64, 0.768])` should return `s=[[0.0, 0.17, 1.108], [0.768, 1.706, 1.876]]` and `t=[[0.0, -0.985, -1.33], [0.64, 0.295, -0.69]]`.

1.10 `include_atoms_gr` (1 point)

Finally, you will use the coordinates and distances in order to determine which atoms to include in your graphene sheet. This can be done by testing whether each atom is inside or outside the designated region. To this end, you will use the function `include_atoms_gr()` provided to you in the `misc.py` script. This method takes as input **six** positional arguments (in **strict order**) corresponding to: the x and y coordinates, the s and t distance, the length of vector C_h and the length l of the sheet, as measured in carbon-carbon bond lengths.

1. `x` (`2-d numpy arrays`): $x-$coordinates of the atoms.
2. `y` (`2-d numpy arrays`): $y-$coordinates of the atoms.
3. `s` (`2-d numpy arrays`): distance of atoms from the mouth of the tube ($x-$axis).
4. `t` (`2-d numpy arrays`): distance of atoms along the length direction ($y-$axis).
5. `arclen` (`float`): Length of vector C_h.
6. `t` (`float`): length of the sheet, as measured in carbon-carbon bond lengths.

```
def include_atoms_gr( x, y, s, t, arclen, l):
        return pos_gr
```

and returns the position of the atoms in the graphene sheet as a `numpy.array`.

1.11 `include_atoms_nt` (2 points)

The function `include_atoms_nt` provided to you in the `misc.py` script will be used in order to roll up the graphene sheet into a nanotube. In order to do this, we map the distance along the mouth direction to a cylinder (to do that, if d is the distance in the mouth direction and r is the radius of the tube then $x = r\cos(d/r)$ and $y = r\sin(d/r)$). This method takes as input **4** positional arguments (in **strict order**) that correspond to: the position of the atoms of the graphene sheet (output of the `include_atoms_gr()` method), the normal vector $\hat{c}$, the length of the vector C_h and the radius of the tube which you are asked to calculate).

1. `pos_gr` (`numpy.array`): x, y coordinates of the atoms of the graphene sheet.
2. `c_hat` (`numpy.array`): coordinates of the normal vector $\hat{c}$.
3. `tubrad` (`float`): radius of the tube.

```
def include_atoms_nt( pos_gr , c_hat , arclen , tubrad):
        return pos_nt
```

and returns the position of the atoms in the nanotube as a `numpy.array`.

1.12 Method `Graphene`(1 point)

In addition, this file contains the method `Graphene()`. This method **is not part** of the `CarbonFiller` class. The method will use the methods of the `CarbonFiller` class in order to construct the structure of the graphene sheet and the carbon nanotube.

```
def Graphene(n, m ,l):

        Cf = CarbonFiller(n, m, l)
        Ch = Cf.vector()
        T = Cf.TVector(Ch)

        p, q = Cf.pq(Ch, T)

        Pgrid, Qgrid = grid_pq(p, q)

        x, y = Cf.coordinates(Pgrid, Qgrid)

        c_hat, arclen = Cf.normVector(Ch)
        s, t = Cf.distance(x, y, c_hat)

        pos_gr = include_atoms_gr(x, y, s, t, arclen, l)

        return pos_gr
```

An example run would look like this:

```
pos_gr = Graphene(1, 0, 1)
print(pos_gr)
```

and the output should be:

```
[[0.    0.    ]
 [1.732 0.    ]
 [0.    1.    ]
 [1.732 1.    ]]
```

2 Methods in `plot_nano.py`

To visualize the graphene sheet, you are asked to create four more functions in `plot_nano.py`. The idea here is to identify carbons that are connected in the graphene sheet and plot red circles at those locations and connect them using a straight blue line.

2.1 `find_connectivity` (3 points)

The first function, `find_connectivity`, will be used to find the Euclidean distance squared of two points to decide the connectivity. If the distance is outside the upper and lower thresholds, this value will be set to zero.

The inputs to this function are:

- points: a two-dimensional (2D) numpy array that contains x and y coordinates of points in 2D space. Accordingly, this numpy array will have a dimension of number of points $\times$ 2.
- low_th: a scalar float quantity defines the lower accepted threshold values for the Euclidean distance squared value. Use 0.95.
- up_th: a scalar float quantity defines the upper accepted threshold values for the Euclidean distance squared value. Use 1.05.

The output to this function is:

- A two-dimensional numpy array. Elements in this array have value 0 if the Euclidean distance squared value the i-th and j-th points are less than or equal 0.95 or greater than or equal 1.05; otherwise, the values is calculated as follows: if $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$, then Euclidean distance squared is defined as $(x_2 - x_1)^2 + (y_2 - y_1)^2$.

An example run would look like this:

```
pp = np.array([[0.98741274, 0.45638062],
 [0.01421964, 0.90581148],
 [0.30552894, 0.03409079]])
print(find_connectivity(pp,0.5,1))
```

and the output should be:

```
[[0.         0.         0.64329422]
 [0.         0.         0.84475807]
 [0.64329422 0.84475807 0.        ]]
```

Function definition summary:

```
def find_connectivity(points,low_th,up_th)
    return DistMat
```

2.2 get_connectivity_ind (3 points)

The second function, get_connectivity_ind, searches the distance numpy array and returns the indexes of the elements that contain a Euclidean distance squared value greater than 0. In this function, we will call the `find_connectivity` function and then search for the indexes of the relevant connected points in the distance array.

The inputs to this function are:

- points: a two-dimensional (2D) numpy array that contains x and y coordinates of points in 2D space. Accordingly, this numpy array will have a dimension of number of points $\times$ 2.
- low_th: a scalar float quantity defines the lower accepted threshold values for the Euclidean distance squared value. Use 0.95
- up_th: a scalar float quantity defines the upper accepted threshold values for the Euclidean distance squared value. Use 1.05

The output of this function is:

- This function should return two numpy arrays, which are indexes of points where the distance squared is within the defined limit. the first numpy array contains the indexes for the first points and the second array contains the indexes for the second set of points.

An example run would look like this:

```
pp = np.array([[0.98741274, 0.45638062],
 [0.01421964, 0.90581148],
 [0.30552894, 0.03409079]])
xx,yy = get_connectivity_ind(pp,0.5,1)
print(xx,yy)
```

and the output should be:

```
[0 1] [2 2]
```

This indicates that the point with index 0 in variable *pp* with coordinates $(0.98741274, 0.45638062)$ is within the accepted distance squared range of point with index 2 in variable *pp* with coordinates $0.30552894, 0.03409079$. The same holds with point indexes 1 and 2.

Function definition summary:

```
def get_connectivity_ind(points,low_th,up_th)
    return index_points_limits
```

2.3 plot_carbon_carbon(3 points)

The third function, plot_carbon_carbon, will plot two small dots (carbons) at a specific location and connected by a line. The inputs of this functions are:

- p1 : numpy array coordinates of point 1
- p2 : numpy array coordinates of point 2

Running the following code should display:

```
p1 = np.array([0.98741274, 0.45638062])
p2 = np.array([0.30552894, 0.03409079])
plot_carbon_carbon(p1,p2)
```

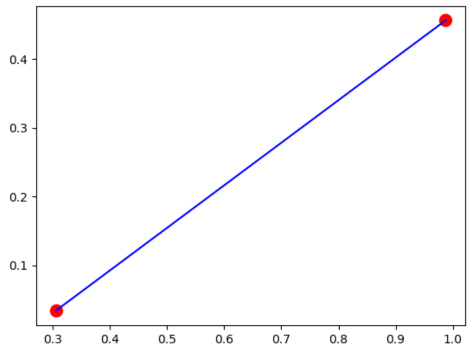

Figure 4

Function definition summary:

```
def plot_carbon_carbon(p1,p2)
```

2.4 plot_mesh (3 points)

The fourth function, plot_mesh, computes the actual mesh of the nanotube. This function will use plot_carbon_carbon to generate the figure with small dots, connecting every dot with a straight line.

The inputs are:

- xx : numpy array indexes for the first points
- yy : numpy array indexes for the second points
- points : numpy array x and y coordinates of carbons in 2D space

Function definition summary:

```
def plot_mesh(points,low_th,up_th)
```

Running the following code should display:

```
a = np.array([[0.    , 0.    ],
             [2.598, 1.5   ],
             [1.732, 1.    ],
             [0.866, 1.5   ],
             [0.    , 1.    ],
             [5.196, 3.    ],
             [4.33  , 2.5   ],
             [3.464, 3.    ],
             [2.598, 2.5   ]])
low_th = 0.95
up_th = 1.05
xx,yy = get_connectivity_ind(a,low_th,up_th)
plot_mesh(xx,yy,a)
```

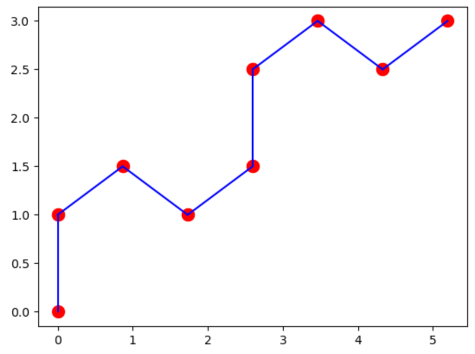

Figure 5

Test your code by comparing the final plot to the sample outputs in the file “Project D tests” as well as the tests of your own design. Make sure to include a PDF (`Tests.pdf`) showing your test results and observations.

Tip

When using the `plot` function from `plot_nano.py` script you might get the following message in Spyder: ”Figures now render in the Plots pane by default. To make them also appear inline in the Console, uncheck "Mute Inline Plotting" under the Plots pane options menu.". In order to display the plots, go to Spyder Preferences $\longrightarrow$ IPython console $\longrightarrow$ Graphics $\longrightarrow$ Graphics Backend $\longrightarrow$ Automatic (instead of Inline).

Grading

Your submission files should include: `solution.py` (containing class `CarbonFiller`), `plot_nano.py` (containing `find_connectivity.py, get_connectivity_ind.py, plot_carbon_carbon.py, plot_mesh.py`) and `Tests.pdf`.

- Make sure that the methods are **well-commented** for full credit. This means that they should contain comments explaining what the code does.
- Include a lot of comments in your code. You must write docstrings for all functions and classes.
- **We will deduct points** if file names are misspelled or if submitted late.
- Upload files to **GradeScope** by 11:59pm ET on 12/09/22.