Third Year Project - Dario De Stefano - final version

May 1, 2021

1 Third year project

1.1 Dario De Stefano

[1]:
```
import math
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt


from tqdm.notebook import tqdm

import dash
import dash_core_components as dcc
import dash_html_components as html
from dash.dependencies import Input, Output
import plotly.express as px
import plotly.graph_objects as go
from plotly.subplots import make_subplots
```

2 Protein folding using the HP model

In this project I am simulating protein folding using the HP mode. Amino acids are modelled as either hydrophobic or polar. Hydrophobic interactions are considered the main drive in the folding process (HH). The peptide bonds are displayed as lines, and amino acid as beads. The energy scheme used, lowers the total energy by 2 every time 2 H beads are neighbouring (but not interacting via peptide bonds).

The code runs many random rotations of portions of the protein to find which moves lower the energy. These moves are then accepted as the new starting point. In order to prevent reaching local minima, there is a condition, fulfilled with a certain % that allows moves with higher energy (unfolding moves).

This process is also referred to as sampling the energy space. In fact, by allowing quite a lot of unfolding moves and by running through a large number of iterations a large number of configurations and the relative energy are identified and compared (and potentially added to a record).

Enjoy!

Please restart kernel if there is any part of the code which is not functioning as it should be. This may due to variables pointing at the same object (especially because the 3D version is an extension of the 2D one).

3 2D Version

[2]:
```
"""This function generates a random walk that simulates the protein
as a string of beads which are either polar or hydrophobic"""

"""n is the dimension of the initial squared lattice from which the initial␣
 ↪coordinates will be chosen from,
length is the number of beads of the protein"""


def makeProtein(n, length):

    protein_coordinates=[[],[]]
    beads=[]

    x=np.random.randint(0,n-10)
    y=np.random.randint(0,n-10)

    protein_coordinates[0].append(x)     #easier to plot than␣
 ↪"protein_coordinates.append([x,y])"
    protein_coordinates[1].append(y)
    beads.append(np.random.randint(0,2))

    count=0

    while count <= length:


        x_increase=np.random.randint(0,2)
        y_increase=np.random.randint(0,2)



        if x_increase!=y_increase:      # This is to avoid consecutive beads on␣
 ↪a "diagonal" line.

            x=x + x_increase
            y=y + y_increase

            protein_coordinates[0].append(x)
            protein_coordinates[1].append(y)
            beads.append(np.random.randint(0,2))
```

```
            count+=1

        else:

            pass


    return protein_coordinates, beads
```

[3]:
```
"""This function performs a rigid rotation of one "leg" (segment) of the␣
↪protein
around a specific bead. It also prevents rotation that would cause an overlap
of different segments of the protein. Theta is the angle of rotation"""

def makeRandRotation(protein_coordinates,beads):
    counter=0
    while 1>0:

        theta=(np.pi/2)*np.random.randint(1,4)   #The angle is chosen randomly␣
↪and it can be either 90,180 or 270°

        counter+=1
        starting=np.random.randint(1,len(beads)-1)

        """I am adding a random flip to allow rotation of segments on both␣
↪sides of 'starting':
        """
        flip=np.random.randint(0,101)
        if flip>50:
            protein_coordinates[0]=np.flip(protein_coordinates[0])
            protein_coordinates[1]=np.flip(protein_coordinates[1])

        else:
            pass

        rotating_x=protein_coordinates[0][starting:]
        rotating_y=protein_coordinates[1][starting:]

        rotating_leg=np.array([rotating_x,rotating_y])

        rotating_leg[0]=rotating_leg[0]-protein_coordinates[0][starting]
        rotating_leg[1]=rotating_leg[1]-protein_coordinates[1][starting]

        """I am here using rotation matrices to change the coordinates of the␣
↪rotating portion of the protein.
        The portion has been translated to the origin (which is fixed at the␣
↪coordinates of "starting") and then vector/matrix
```

```
        multiplication is used to rotate the leg."""

        rotated_x= rotating_leg[0]*np.cos(theta) - rotating_leg[1]*np.
↪sin(theta) + protein_coordinates[0][starting]
        rotated_y= rotating_leg[0]*np.sin(theta) + rotating_leg[1]*np.
↪cos(theta) + protein_coordinates[1][starting]

        rotated_protein_x=np.concatenate([protein_coordinates[0][:
↪starting],rotated_x])
        rotated_protein_y=np.concatenate([protein_coordinates[1][:
↪starting],rotated_y])

        rotated_protein=np.array([rotated_protein_x,rotated_protein_y])

        '''To prevent rotations that may end up with overlapping segments of␣
↪the protein: '''


        for x in range(len(rotated_protein[0])):
            neighbouring_points= np.where(np.
↪sqrt((rotated_protein[0]-rotated_protein[0][x])**2 +␣
↪(rotated_protein[1]-rotated_protein[1][x])**2)<1 )

            if len(neighbouring_points[0])>1:

                break

            elif x==len(rotated_protein[0])-1:

                return rotated_protein, beads
        if counter>1000:
            print("loop")
```

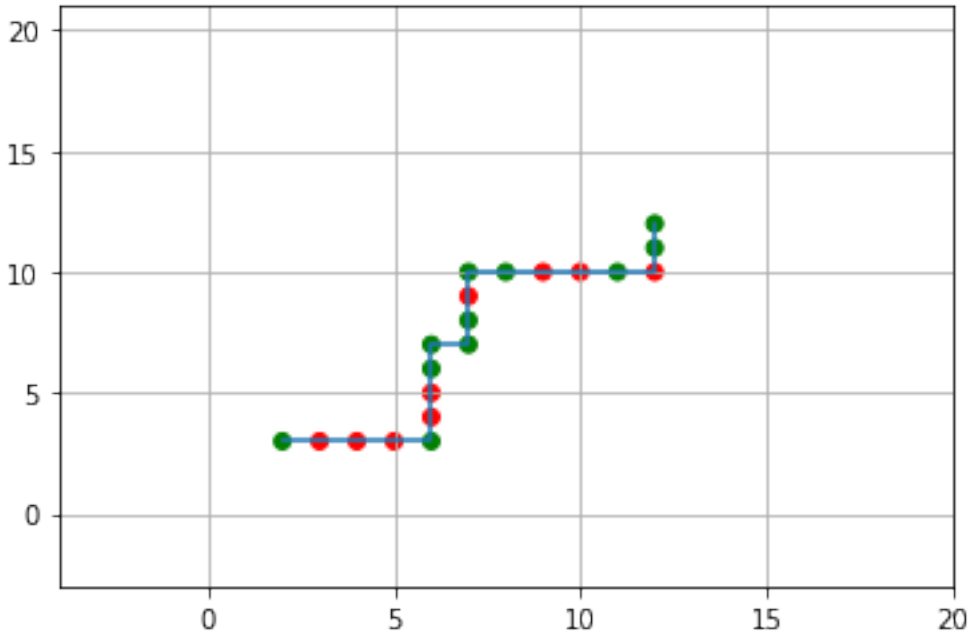
[4]:
```
"""This function is only for plotting purposes: """


def makeColours(beads):

    colours=[]

    for colour in beads:
        if colour==0:
            colours.append('red')
        else:
            colours.append('green')
```

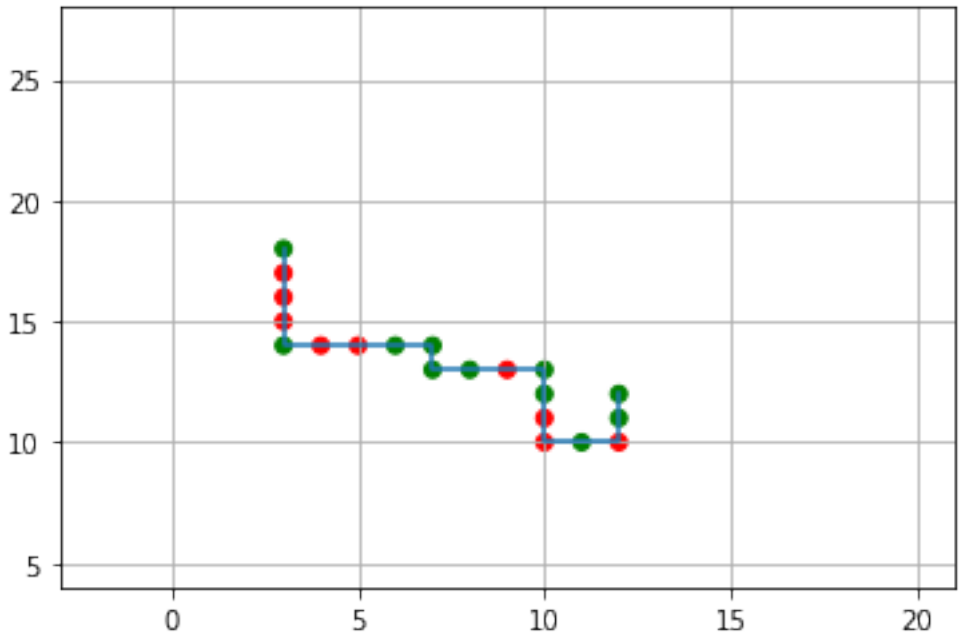
```
    return colours
```

[5]:
```
def plotProtein(x,y,colours,n,l):

    plt.grid(axis="both")
    plt.scatter(x,y,color=colours)
    plt.plot(x,y)
    plt.xlim(min(x)-l/3,min(x)+l)
    plt.ylim(min(y)-l/3,min(y)+l)
    plt.show()
```

[6]:
```
def plotProteinZoom(x,y,colours,n,l):

    plt.grid(axis="both")
    plt.scatter(x,y,color=colours)
    plt.plot(x,y)
    plt.xlim(min(x)-5,max(x)+5)
    plt.ylim(min(y)-5,max(y)+5)
    plt.show()
```

[7]:
```
"""
    This tab can be used to visualise a randomly generated protein and its␣
 ↪shape
    after ONE rigid rotation

    makeProtein(n, length) it takes two arguments, the size of the square␣
 ↪lattice and the lenght of the protein.

"""
n=20
l=18
protein=makeProtein(n,l)

rotated_protein, beads=makeRandRotation(protein[0],protein[1])
colours_1= makeColours(protein[1])
colours_2= makeColours(beads)
plotProtein(protein[0][0],protein[0][1],colours_1,n,l)
plotProtein(rotated_protein[0],rotated_protein[1],colours_2,n,l)
```





[8]:
```
def evaluateEnergy(rotated_protein, beads):

    total_energy=0
```

```
   for aminoacid in range(len(beads)):


       if beads[aminoacid]==0:


           """I am here using numpy.where to find all the amino acids around
           each hydrophobic amino acid. Namely, the read bead, corresponding␣
↪to value 0.
           Once the 4 (or less) neighbouring amino acids have been identified,
           the following line of code, will distinguish between those joined␣
↪by peptide bonds
           and the others."""

           neighbouring_points= np.where(np.
↪sqrt((rotated_protein[0]-rotated_protein[0][aminoacid])**2 +␣
↪(rotated_protein[1]-rotated_protein[1][aminoacid])**2)<1.01 )



           #print("I am checking aminoacids around the aminoacid number {}␣
↪which is {} (0 means H, 1 means P)".format(aminoacid,beads[aminoacid]))
           #print("the neighbouring points are: {}".
↪format(neighbouring_points))


           for neighbour in neighbouring_points[0]:
               """I don't want to count the amino acid itself for the energy␣
↪calculation:"""
               if neighbour==aminoacid:
                   pass

               else:
                   "consecutive beads are distanced  by ONE unit."
                   if (np.absolute(neighbour - aminoacid )!= 1 and␣
↪beads[neighbour]==0):
                       #print("of which, {} is not a peptide bond and it is {}␣
↪(0 means H, 1 means P)".format(neighbour,beads[neighbour]))
                       #print("therefore energy decreases by 1")
                       total_energy += protein[1][neighbour] -1
                   else:
                       pass
       else:
           pass
   return total_energy
```

```
evaluateEnergy(rotated_protein, beads)
```

[8]: 0

[9]:
```
def energyPlot(energy, x):

    y_seq=pd.Series(energy)
    x_seq=pd.Series(x)
    df=pd.DataFrame()
    df["x"]=x_seq
    df["y"]=y_seq
    fig = px.line(df, x="x", y="y", title='Energy of local minima found')
    fig.show()
```

[10]:
```
"""Initial configuration and first rotation  !!  length protein is l+2 !!   """

n=50
l=20
protein=makeProtein(n,l)

interaction= makeColours(protein[1])


total_energy_in=evaluateEnergy(np.array(protein[0]), protein[1])
print("Initial energy is: {}".format(total_energy_in))
rotated_protein, beads=makeRandRotation(protein[0],protein[1])
total_energy=evaluateEnergy(rotated_protein, beads)
print("Energy after the first rigid rotation is: {}".format(total_energy))

"""Cell is separated from the next one to allow a different number of␣
 ↪iterations on the same protein"""
```

```
Initial energy is: 0
Energy after the first rigid rotation is: -2
```

[10]: 'Cell is separated from the next one to allow a different number of iterations on the same protein'

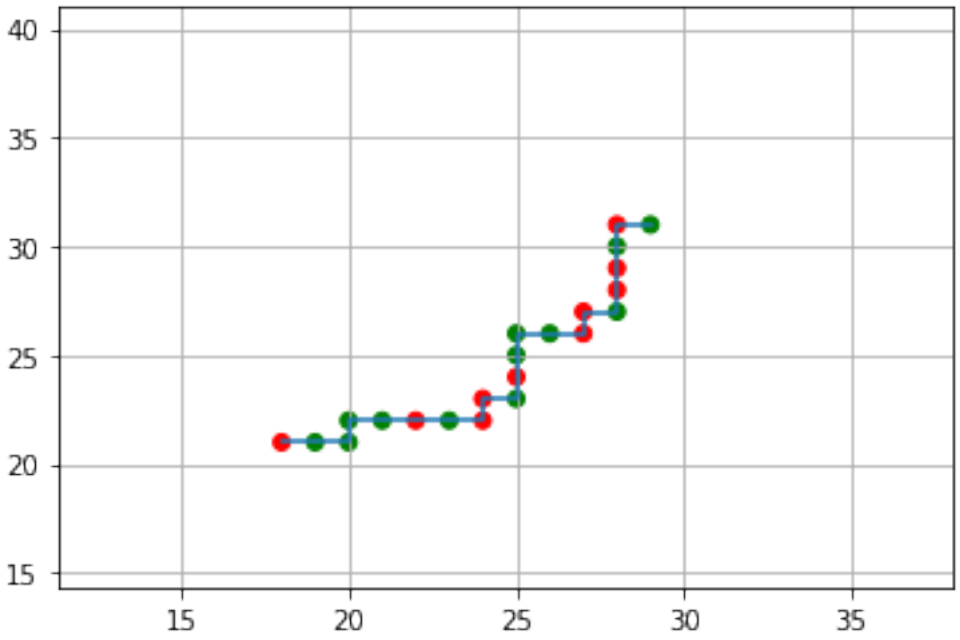
4 MAIN BODY:

[11]:
```
m=0

print("The initial configuration is: ")
plotProtein(protein[0][0],protein[0][1],interaction,n,l)
print("Initial energy is: {}".format(total_energy_in))
```

```
minima={}
minima_beads={}
place_minimum=[]
minima[total_energy]=rotated_protein
minima_beads[total_energy]=beads


"""In the following lines of code, a new configuration is proposed in each␣
↪iteration.
If the new configuration lowers the energy, OR if R<W is satisfied, the more is␣
↪accepted, otherwise it is rejected."""


iterations=100000
while m<iterations:
    for i in tqdm(range(iterations)):
        m+=1

        rotated_protein_new, beads_new=makeRandRotation(rotated_protein,beads)
        total_energy_new=evaluateEnergy(rotated_protein_new,  beads_new)

        """
        The following condition serves the purpose of allowing some partial␣
↪unfolding of the protein.
        This is to prevent the code from getting stuck into a local minimum.

        This can also be used to simulate different temperature in real world.␣
↪In fact, at higher temperature,
        folding does not happen.

        In this simulation if condition R<W is made more likely to happen,e.g.␣
↪by changing how W is defined
        (by dividing by T), then the code will and unfold the protein  without␣
↪reaching a more stable configuration.
        However, please note that is a simplified model and there is no way to␣
↪check whether a minimum is a global one.

        """

        W=np.exp(-(total_energy_new-total_energy))
        R=np.random.randint(0,1000)/1000

        if total_energy_new-total_energy <0 :
            rotated_protein=rotated_protein_new
```

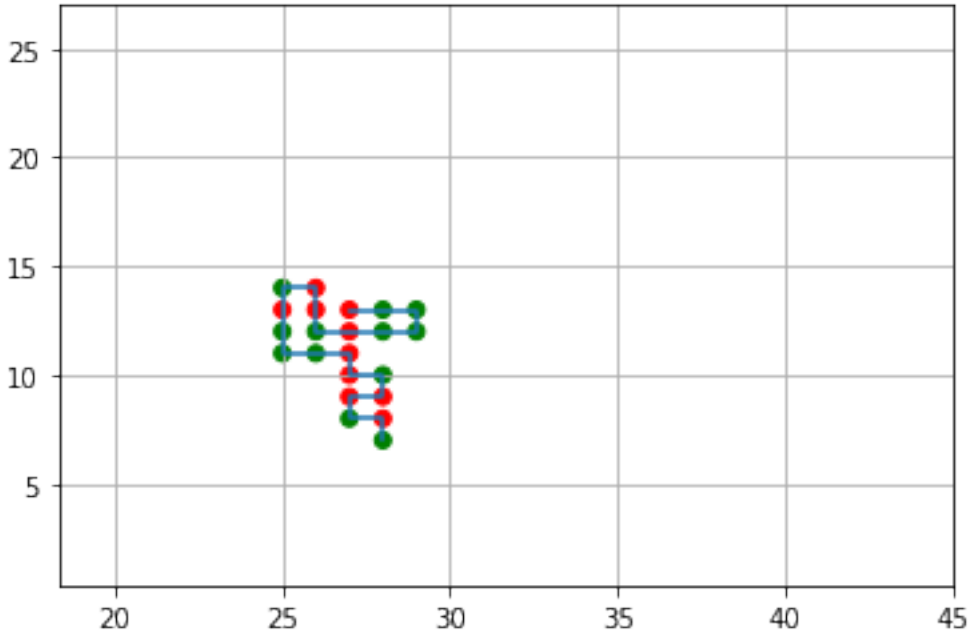
```
            total_energy=total_energy_new
            place_minimum.append(m)

            minima[total_energy]=rotated_protein
            minima_beads[total_energy]=beads_new


        elif R<W:
            rotated_protein=rotated_protein_new

        else:
            pass


print("The lowest minimum is: ")
folded_protein =minima[min(minima)]
interaction= makeColours(minima_beads[min(minima)])
plotProtein(folded_protein[0],folded_protein[1],interaction,n,l)
plotProteinZoom(folded_protein[0],folded_protein[1],interaction,n,l)
print("Energy is:{} ".format(min(minima)))
print("number of iterations:{}".format(m))
```

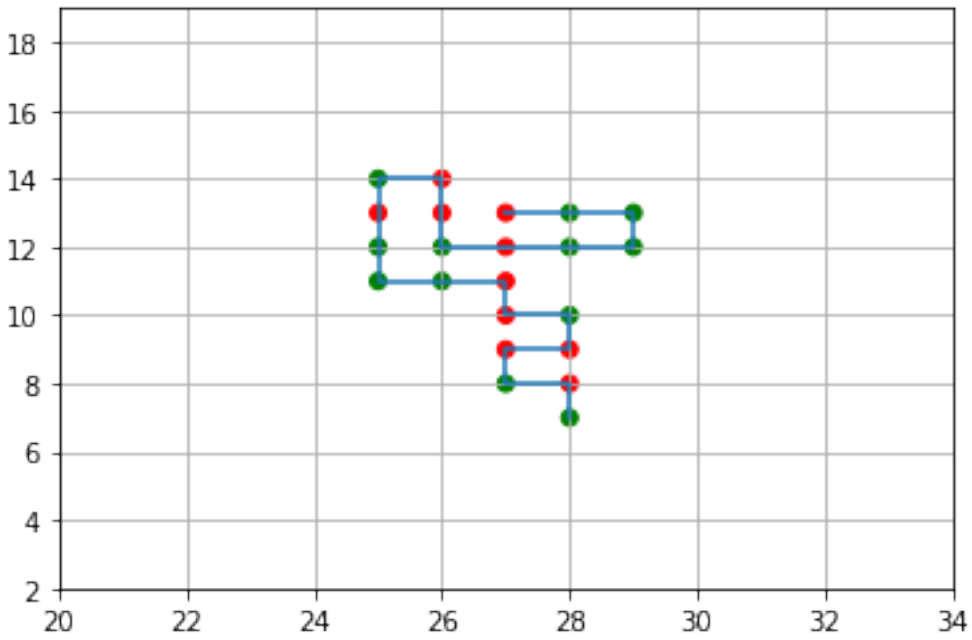
```
The initial configuration is:
```




```
Initial energy is: 0
HBox(children=(FloatProgress(value=0.0, max=100000.0), HTML(value='')))
```

The lowest minimum is:





Energy is:-12

```
number of iterations:100000
```

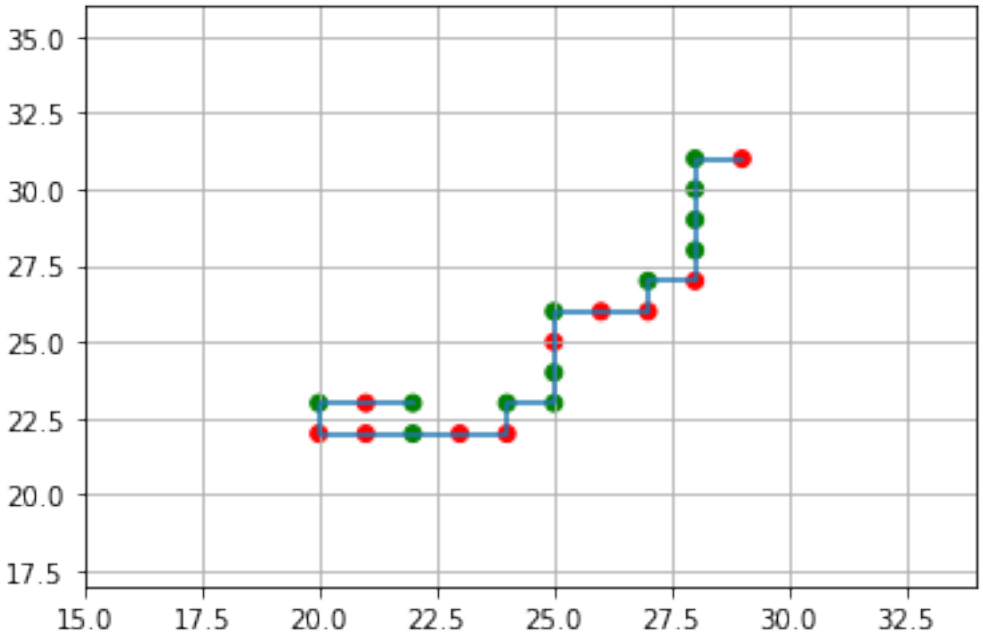
[12]:
```
#evaluateEnergy(folded_protein,  beads_new)
print(list(minima.keys()))

energyPlot(list(minima.keys()), place_minimum)
```

```
[-2, -4, -6, -8, -10, -12]
```

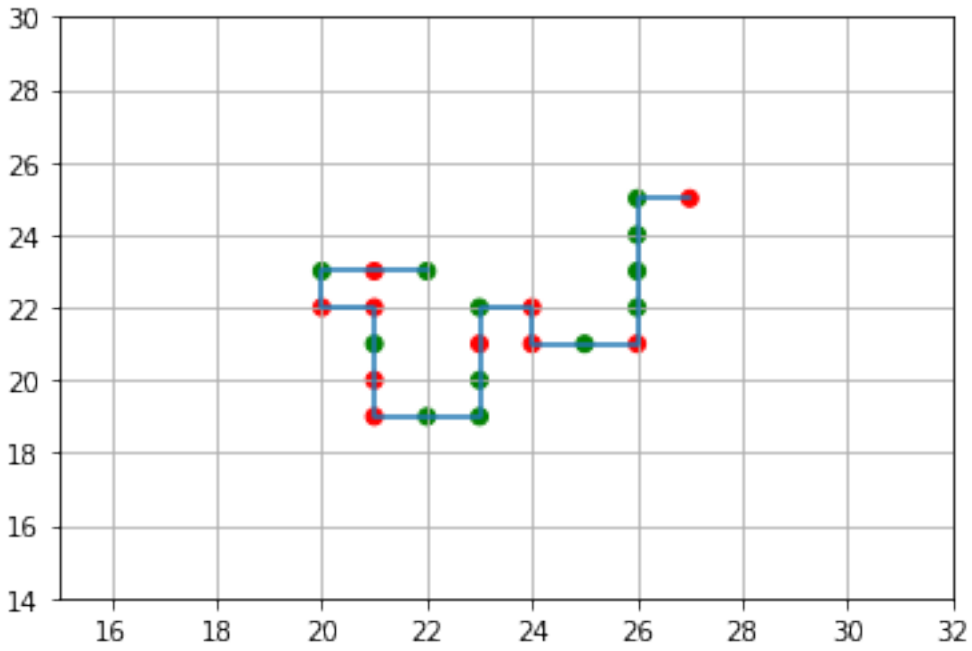
4.1 The following are all the minima that have been found:

[13]:
```
for minimum in minima:
    folded_protein =minima[minimum]
    beads_new= minima_beads[minimum]
    interaction= makeColours(beads_new)
    print("Energy is: ", minimum)
    plotProteinZoom(folded_protein[0],folded_protein[1],interaction,n,l)
```

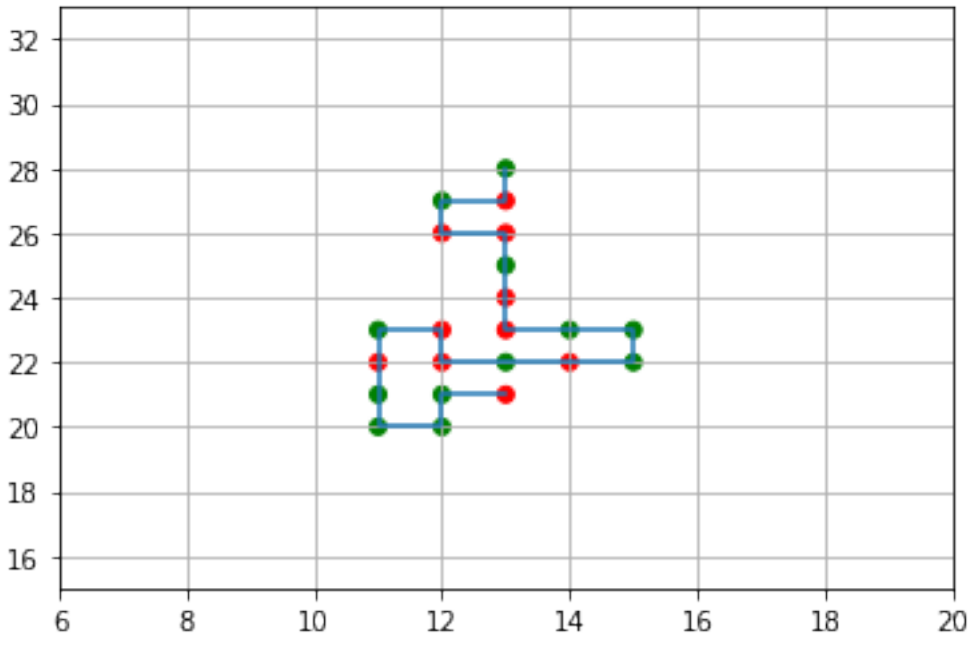
```
Energy is:  -2
```



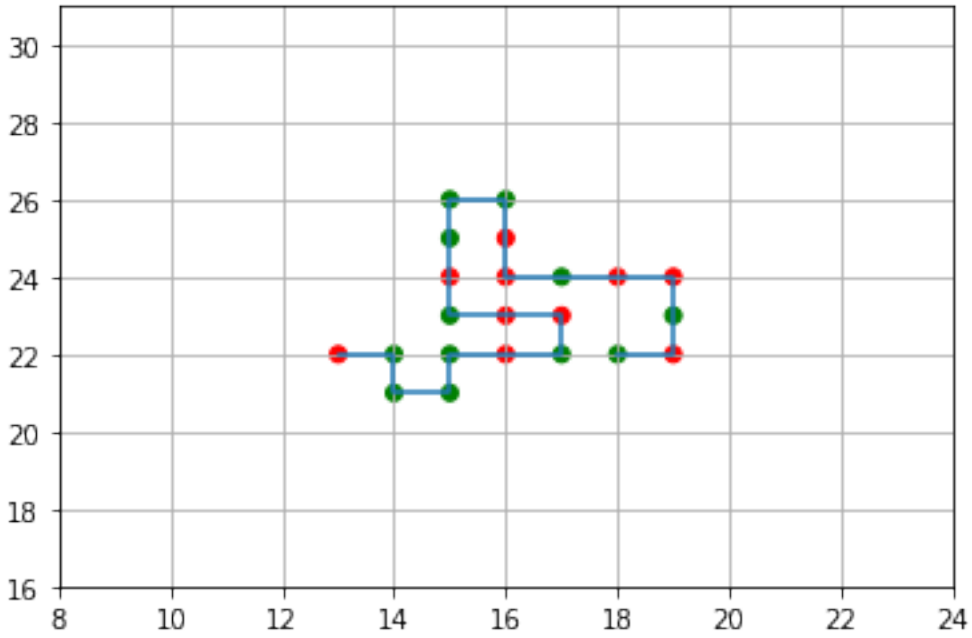

```
Energy is:  -4
```



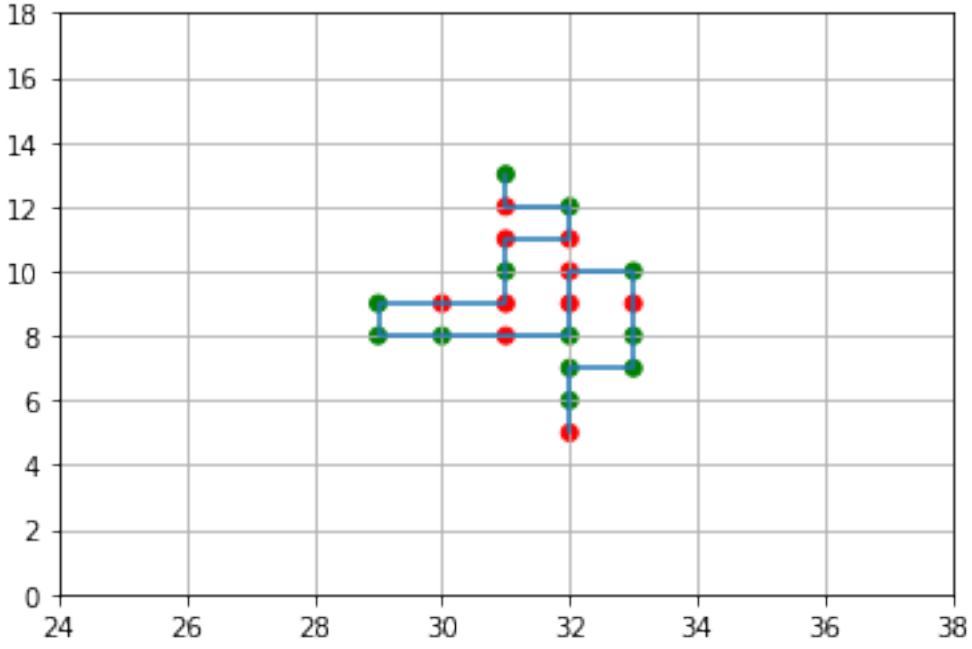

```
Energy is:  -6
```



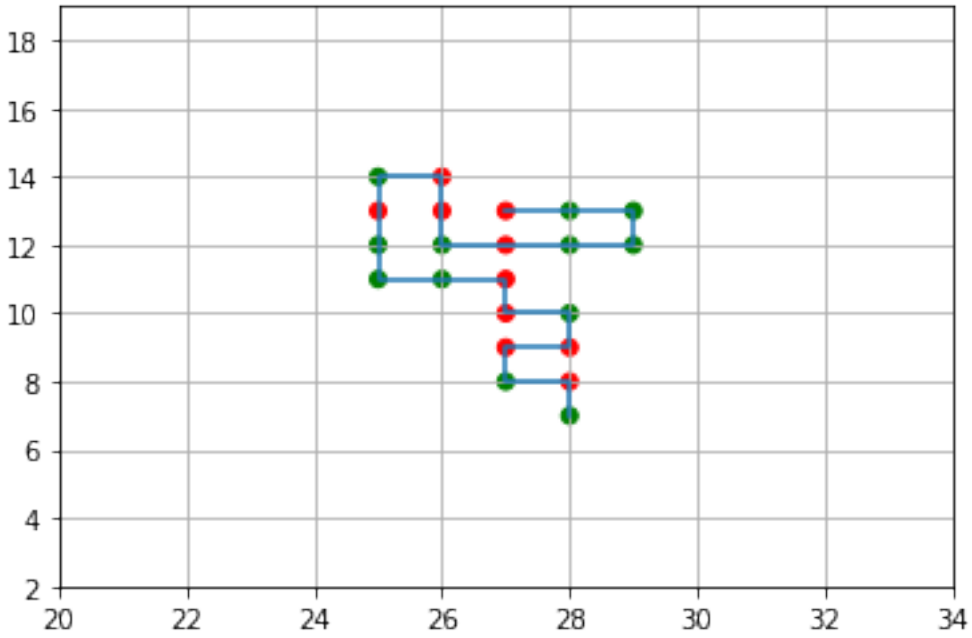

```
Energy is:  -8
```




```
Energy is:  -10
```




```
Energy is:  -12
```



5 3D VERSION:

[1]:
```
import math
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt


from tqdm.notebook import tqdm

import dash
import dash_core_components as dcc
import dash_html_components as html
from dash.dependencies import Input, Output
import plotly.express as px
import plotly.graph_objects as go
from plotly.subplots import make_subplots
```

[2]:
```
"""This function generates a random walk that simulates the protein
as a string of beads which are either polar or hydrophobic"""

"""n is the dimension of the squared lattice, length is the number of beads of␣
 ↪the protein"""

"""an optional argument has been added on the latest version of this code,
```

```
please insert beads with same number of elements as "length" """

def makeProtein3D(n, length, beads=[]):
    check=0
    if beads==[]:
        check=1

    protein_coordinates=[[],[],[]]


    x=np.random.randint(0,n)
    y=np.random.randint(0,n)
    z=np.random.randint(0,n)


    protein_coordinates[0].append(x)     #easier to plot than␣
↪"protein_coordinates.append([x,y])"
    protein_coordinates[1].append(y)
    protein_coordinates[2].append(z)
    if check==1:
        beads.append(np.random.randint(0,2))
    else:
        pass

    count=0

    while count <= length-2:


        x_increase=np.random.randint(0,2)
        y_increase=np.random.randint(0,2)
        z_increase=np.random.randint(0,2)


        if x_increase+y_increase+z_increase>1 or␣
↪x_increase==y_increase==z_increase==0:

            pass

        else:
            #print(x_increase,y_increase,z_increase)

            x=x + x_increase
            y=y + y_increase
            z=z + z_increase

            protein_coordinates[0].append(x)
```

```
            protein_coordinates[1].append(y)
            protein_coordinates[2].append(z)
            if check==1:
                beads.append(np.random.randint(0,2))
            else:
                pass

            count+=1


    return protein_coordinates, beads
```

[3]:
```
"""This function performs a rigid rotation of one "leg" (segment) of the protein
around a specific bead. It also prevents rotation that would cause an overlap
of different segments of the protein. Theta is the angle of rotation"""

def makeRandRotation3D(protein_coordinates,beads):

    while 1>0:

        starting=np.random.randint(1,len(beads)-1)
        theta=(np.pi/2)*np.random.randint(1,4)
        phi=(np.pi/2)*np.random.randint(1,4)


        """I am adding a random flip to allow rotation of segments on both sides of 'starting':   """
        flip=np.random.randint(0,101)
        if flip>50:
            protein_coordinates[0]=np.flip(protein_coordinates[0])
            protein_coordinates[1]=np.flip(protein_coordinates[1])
            protein_coordinates[2]=np.flip(protein_coordinates[2])

        else:
            pass

        rotating_x=protein_coordinates[0][starting:]
        rotating_y=protein_coordinates[1][starting:]
        rotating_z=protein_coordinates[2][starting:]

        rotating_leg=np.array([rotating_x,rotating_y,rotating_z])

        rotating_leg[0]=rotating_leg[0]-protein_coordinates[0][starting]
        rotating_leg[1]=rotating_leg[1]-protein_coordinates[1][starting]
        rotating_leg[2]=rotating_leg[2]-protein_coordinates[2][starting]
```

```
        rotated_x= rotating_leg[0]*np.cos(phi) - rotating_leg[1]*np.sin(phi) +␣
↪protein_coordinates[0][starting]
        rotated_y= rotating_leg[0]*np.cos(theta)*np.sin(phi) +␣
↪rotating_leg[1]*np.cos(theta)*np.cos(phi)-rotating_leg[2]*np.sin(theta)  +␣
↪protein_coordinates[1][starting]
        rotated_z= rotating_leg[0]*np.sin(theta)*np.sin(phi) +␣
↪rotating_leg[1]*np.sin(theta)*np.cos(phi)+rotating_leg[2]*np.cos(theta)  +␣
↪protein_coordinates[2][starting]

        rotated_protein_x=np.concatenate([protein_coordinates[0][:
↪starting],rotated_x])
        rotated_protein_y=np.concatenate([protein_coordinates[1][:
↪starting],rotated_y])
        rotated_protein_z=np.concatenate([protein_coordinates[2][:
↪starting],rotated_z])

        rotated_protein=np.
↪array([rotated_protein_x,rotated_protein_y,rotated_protein_z])

        '''To prevent rotations that may end up with overlapping segments of␣
↪the protein: '''


        for x in range(len(rotated_protein[0])):
            neighbouring_points= np.where(np.
↪sqrt((rotated_protein[0]-rotated_protein[0][x])**2 +␣
↪(rotated_protein[1]-rotated_protein[1][x])**2 +␣
↪(rotated_protein[2]-rotated_protein[2][x])**2)<1 )

            if len(neighbouring_points[0])>1:

                break

            elif x==len(rotated_protein[0])-1:
                #print("rotation around point number:{}, theta={} pi/2 around x␣
↪and phi={} pi/2 around z".format(starting,theta/(np.pi/2),phi/(np.pi/2)))
                return rotated_protein, beads
```

[4]:
```
"""This function is only for plotting purposes: """


def makeColours3D(beads):

    colours=[]
```

```
    for colour in beads:
        if colour==0:
            colours.append('red')
        else:
            colours.append('green')

    return colours
```

[5]:
```
def plotProtein3D(x,y,z,colours,n,l):

    fig = plt.figure()
    ax = plt.axes(projection ='3d')
    ax.scatter(x,y,z,color=colours)
    ax.plot(x,y,z,color="blue")
    ax.set_xlim(min(x)- l/2, min(x)+ l/2)
    ax.set_ylim(min(y)- l/2, min(y)+ l/2)
    ax.set_zlim(min(z)- l/2, min(z)+ l/2)


    plt.show()
```

[6]:
```
def plotProteinZoom3D(x,y,z,colours,n,l):



    fig = plt.figure()
    ax = plt.axes(projection ='3d')
    ax.scatter(x,y,z,color=colours)
    ax.plot(x,y,z,color="blue")

    ax.set_xlim(min(x), max(x))
    ax.set_ylim(min(y), max(y))
    ax.set_zlim(min(z), max(z))
    ax.set_autoscale_on(False)
    plt.show()
```

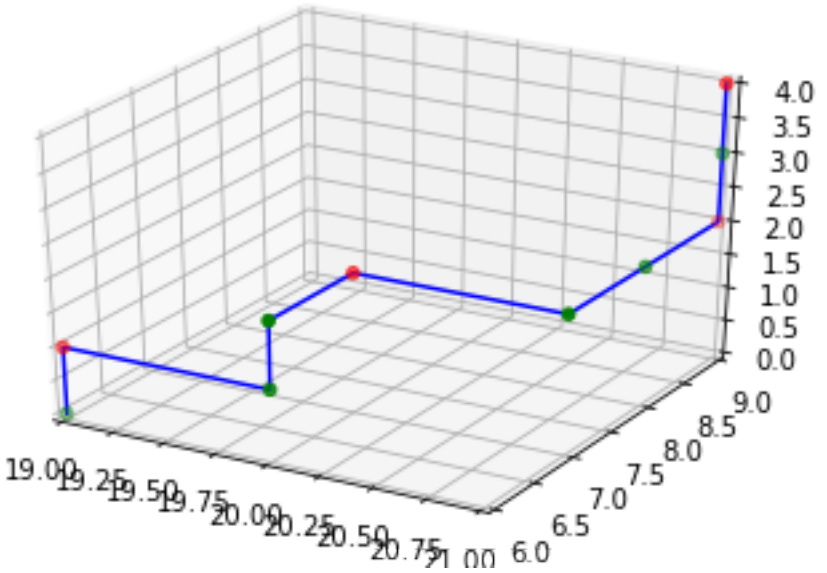
[50]:
```
def evaluateEnergy3D(rotated_protein, beads):

    total_energy=0

    for aminoacid in range(len(beads)):

        if beads[aminoacid]==0:
```

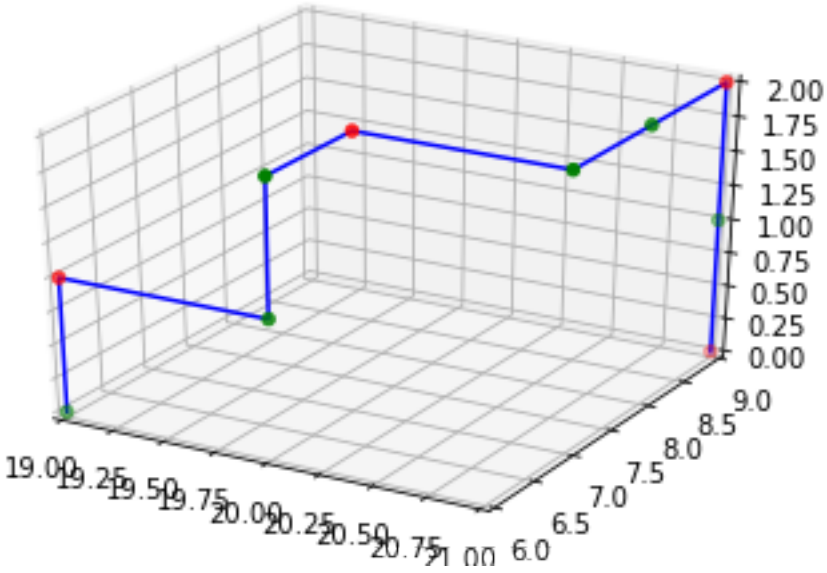
```
            neighbouring_points= np.where(np.
↪sqrt((rotated_protein[0]-rotated_protein[0][aminoacid])**2 +␣
↪(rotated_protein[1]-rotated_protein[1][aminoacid])**2 +␣
↪(rotated_protein[2]-rotated_protein[2][aminoacid])**2)<1.01 )



            for neighbour in neighbouring_points[0]:
                if neighbour==aminoacid:
                    pass
                else:
                    if (np.absolute(neighbour - aminoacid )!= 1 and␣
↪beads[neighbour]==0) :
                        #
                        total_energy +=beads[neighbour] -1
                    else:
                        #print("peptide bond")
                        pass

        else:
            pass
    return total_energy

#evaluateEnergy3D(rotated_protein_test[0], rotated_protein_test[1])
```

[51]:
```
def energyPlot(energy, x):

    y_seq=pd.Series(energy)
    x_seq=pd.Series(x)
    df=pd.DataFrame()
    df["x"]=x_seq
    df["y"]=y_seq
    fig = px.line(df, x="x", y="y", title='Energy of local minima found.'+ "␣
↪Iterations: "+str(iterations)+"\n Sequence: "+sqnc)
    fig.show()
```

[52]:
```
"""
    This tab can be used to visualise a randomly generated protein and its␣
↪shape
    after ONE rigid rotation

"""

"""protein=makeProtein3D(n,length)"""
n_test=20
l_test=10
protein_test=makeProtein3D(n_test,l_test)
```

```
interaction_test=makeColours3D(protein_test[1])
plotProteinZoom3D(protein_test[0][0],protein_test[0][1],protein_test[0][2],interaction_test,n_
rotated_protein_test=makeRandRotation3D(protein_test[0],protein_test[1])
plotProteinZoom3D(rotated_protein_test[0][0],rotated_protein_test[0][1],rotated_protein_test[0
```

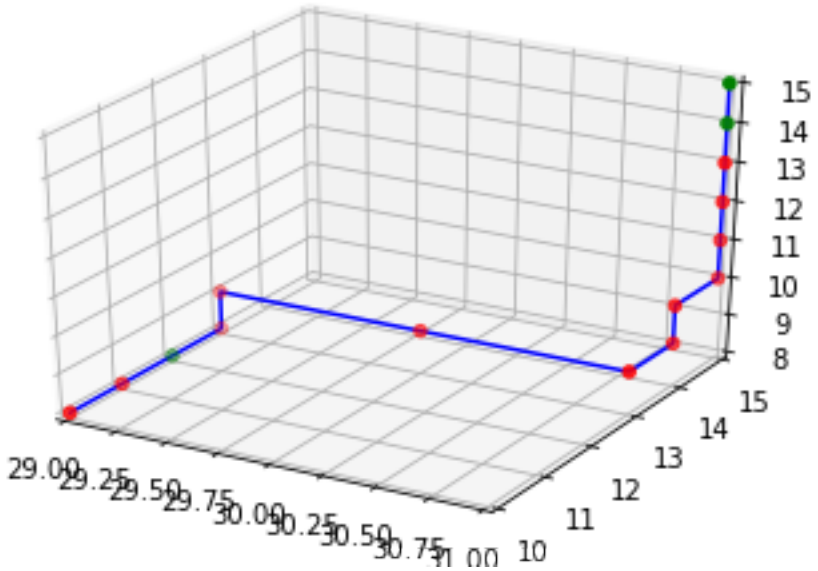




[59]:
```
"""Initial configuration and first rotation"""
```

```
n=50
l=15
protein= makeProtein3D(n,l,[])

print(len(protein[1]), len(protein[0][0]))

'''The follwing is for plotting purposes only: '''
interaction=makeColours3D(protein[1])

print("The initial configuration is: ")
plotProteinZoom3D(protein[0][0],protein[0][1],protein[0][2],interaction,n,l)
print("Initial energy is: {}".format(total_energy_in))



total_energy_in=evaluateEnergy3D(np.array(protein[0]), protein[1])
print("Initial energy is: {}".format(total_energy_in))
rotated_protein, beads=makeRandRotation3D(protein[0],protein[1])
total_energy=evaluateEnergy3D(rotated_protein, protein[1])
print("Energy after the first rigid rotation is: {}".format(total_energy))

"""Cell is separated from the next one to allow a different number of␣
 ↪iterations on the same protein"""
```

```
15 15
The initial configuration is:
```



```
Initial energy is: 0
Initial energy is: 0
Energy after the first rigid rotation is: 0
```

[59]:
```
'Cell is separated from the next one to allow a different number of iterations
on the same protein'
```

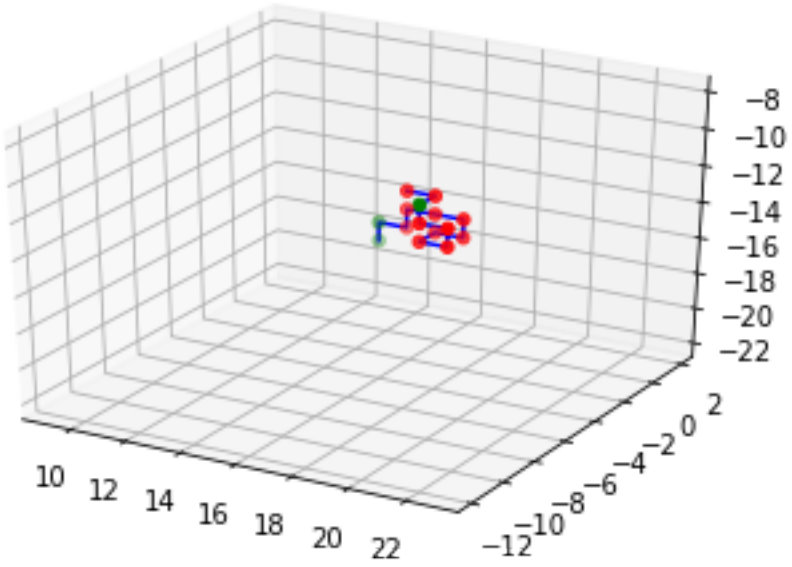
5.1 MAIN BODY STABLE 3D VERSION

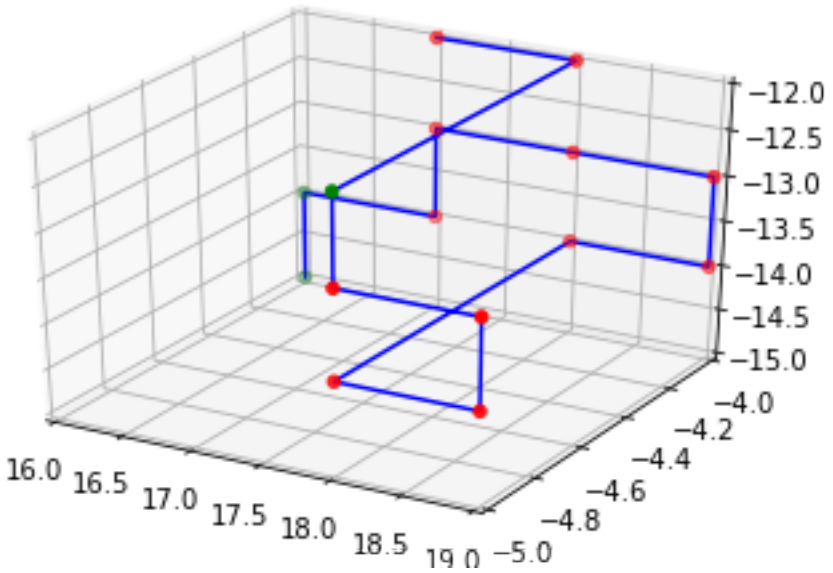
[60]:
```
m=1

minima={}
minima_beads={}
energy_record={}
minima[total_energy]=rotated_protein

iterations=10000
while m<iterations:
    for i in tqdm(range(iterations)):
        m+=1

        rotated_protein_new,␣
 ↪beads=makeRandRotation3D(rotated_protein,protein[1])
        total_energy_new=evaluateEnergy3D(rotated_protein_new, beads)
        energy_record[m]=total_energy_new
        W=np.exp(-(total_energy_new-total_energy))

        R=np.random.randint(0,1000)/1000

        if total_energy_new-total_energy <0 :
            rotated_protein=rotated_protein_new
            total_energy=total_energy_new

            minima[total_energy]=rotated_protein
            minima_beads[total_energy]=beads

        elif R<W:
            rotated_protein=rotated_protein_new
            total_energy=total_energy_new
        else:
            pass


print("The lowest minimum is: ")
folded_protein=minima[min(minima)]
interaction=makeColours3D(minima_beads[min(minima)])
```

```
plotProtein3D(folded_protein[0],folded_protein[1],folded_protein[2],interaction,n,l)
plotProteinZoom3D(folded_protein[0],folded_protein[1],folded_protein[2],interaction,n,l)
print("Energy is:{} ".format(min(minima)))
print("number of iterations:{}".format(m))
```

```
HBox(children=(FloatProgress(value=0.0, max=10000.0), HTML(value='')))
```

The lowest minimum is:






```
Energy is:-16
number of iterations:10001
```

6 Plotly for 3D Plots

[61]:
```
sequence_x= pd.Series(protein[0][0])
sequence_y= pd.Series(protein[0][1])
sequence_z= pd.Series(protein[0][2])

df1= pd.DataFrame()

df1["x"]= sequence_x
df1["y"]= sequence_y
df1["z"]= sequence_z


sequence_x= pd.Series(folded_protein[0])
sequence_y= pd.Series(folded_protein[1])
sequence_z= pd.Series(folded_protein[2])

df= pd.DataFrame()

df["x"]= sequence_x
df["y"]= sequence_y
df["z"]= sequence_z
```

6.1 BEFORE FOLDING

[62]:
```
fig = px.line_3d(df1, x='x', y='y', z='z',)
scatter=go.Scatter3d(x=df1['x'], y=df1['y'], z=df1['z'] ,␣
 ↪mode='markers',marker=dict(size=10,color=interaction))
fig = fig.add_trace(scatter)

fig.show()
```

6.2 AFTER FOLDING

[ ]:
```

```

[63]:
```
fig1 = go.Scatter3d(x=df['x'], y=df['y'], z=df['z'] ,␣
 ↪mode='markers',marker=dict(size=15,color=interaction))



fig = px.line_3d(df, x='x', y='y', z='z',)
fig.add_trace(fig1 )
fig.show()
print("energy is: ", min(minima))
```

```
energy is:  -16
```

7 This is the end of the development of the code. In the following cells I am playing with proteins with a specific -and not randomly selected- sequece (primary structure). You can also display a 3D "folding sequence and other metrics are measured and plotted in the last cells. Have fun!

8 REAL-WORLD PROTEIN TEST !! *Testing/playing around* !!

Charged (side chains often form salt bridges): - Arginine - Arg - R - Lysine - Lys - K - Aspartic acid - Asp - D - Glutamic acid - Glu - E

Polar (form hydrogen bonds as proton donors or acceptors): - Glutamine - Gln - Q - Asparagine - Asn - N - Histidine - His - H - Serine - Ser - S - Threonine - Thr - T - Tyrosine - Tyr - Y - Cysteine - Cys - C

Amphipathic (often found at the surface of proteins or lipid membranes, sometimes also classified as polar. I will classify as hydrophobic): - Tryptophan - Trp - W - Tyrosine - Tyr - Y - Methionine - Met - M (may function as a ligand to metal ions)

Hydrophobic (normally buried inside the protein core): - Alanine - Ala - A - Isoleucine - Ile - I - Leucine - Leu - L - Methionine - Met - M - Phenylalanine - Phe - F - Valine - Val - V - Proline - Pro - P - Glycine - Gly - G

Source: https://proteinstructures.com/Structure/Structure/amino-acids.html

[40]:
```
def turnIntoHP(sequence_string):
    hydrophobic={"A", "I","L","M","F","V","P","G", "W","Y"}
    sequence=[]

    for letter in sequence_string:
        if letter in hydrophobic:
            sequence.append(0)
        else:
            sequence.append(1)
    return sequence
```

9 Alpha Helix:

[34]:
```
Alpha_helix_str="MAEKMAEKMAEKMAE"
Alpha_helix=turnIntoHP(Alpha_helix_str)

print(Alpha_helix)
print(len(Alpha_helix))
```

```
[0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1]
15
```

[35]:
```
"""Initial configuration and first rotation"""

n=50
l=15
protein=makeProtein3D(n,l,Alpha_helix)



'''The follwing is for plotting purposes only: '''
interaction=makeColours3D(protein[1])

total_energy_in=evaluateEnergy3D(np.array(protein[0]), protein[1])
print("Initial energy is: {}".format(total_energy_in))
rotated_protein, beads=makeRandRotation3D(protein[0],protein[1])
total_energy=evaluateEnergy3D(rotated_protein, protein[1])
print("Energy after the first rigid rotation is: {}".format(total_energy))




"""Cell is separated from the next one to allow a different number of␣
 ↪iterations on the same protein"""
```

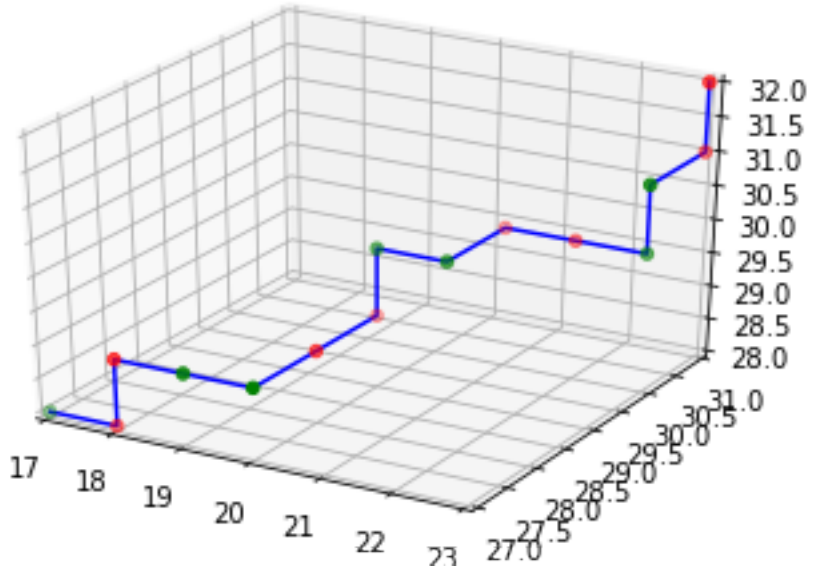
```
Initial energy is: 0
Energy after the first rigid rotation is: 0
```

[35]:
```
'Cell is separated from the next one to allow a different number of iterations
on the same protein'
```

[36]:
```
m=1

print("The initial configuration is: ")
plotProteinZoom3D(protein[0][0],protein[0][1],protein[0][2],interaction,n,l)
print("Initial energy is: {}".format(total_energy_in))



minima={}
minima_beads={}
energy_record={}
minima[total_energy]=rotated_protein

iterations=1000000
while m<iterations:
    for i in tqdm(range(iterations)):
        m+=1

        rotated_protein_new,␣
 ↪beads=makeRandRotation3D(rotated_protein,protein[1])
        total_energy_new=evaluateEnergy3D(rotated_protein_new, beads)
        energy_record[m]=total_energy_new
        W=np.exp(-(total_energy_new-total_energy))
        R=np.random.randint(0,1000)/1000

        if total_energy_new-total_energy <0 :
            rotated_protein=rotated_protein_new
            total_energy=total_energy_new

            minima[total_energy]=rotated_protein
            minima_beads[total_energy]=beads

        elif R<W:
            rotated_protein=rotated_protein_new
            total_energy=total_energy_new
        else:
            pass


print("The lowest minimum is: ")
folded_protein=minima[min(minima)]
```

```
interaction=makeColours3D(minima_beads[min(minima)])

plotProtein3D(folded_protein[0],folded_protein[1],folded_protein[2],interaction,n,l)
plotProteinZoom3D(folded_protein[0],folded_protein[1],folded_protein[2],interaction,n,l)
print("Energy is:{} ".format(min(minima)))
print("number of iterations:{}".format(m))
```

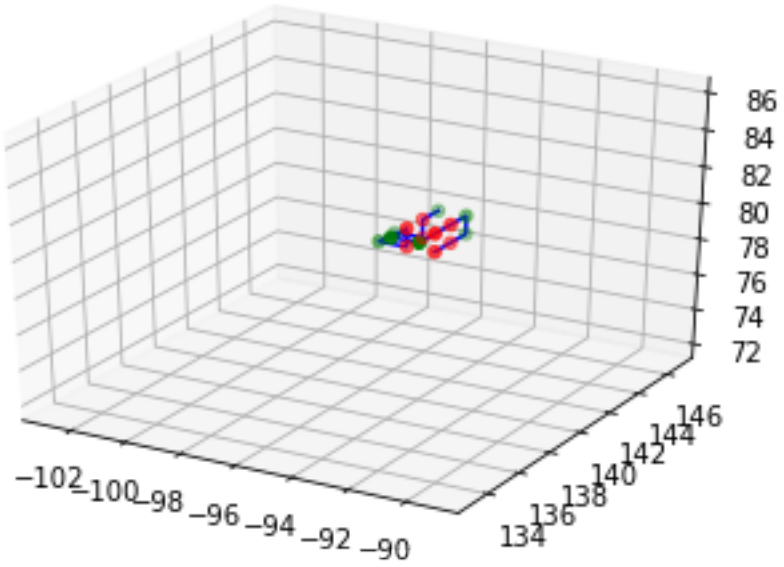
The initial configuration is:

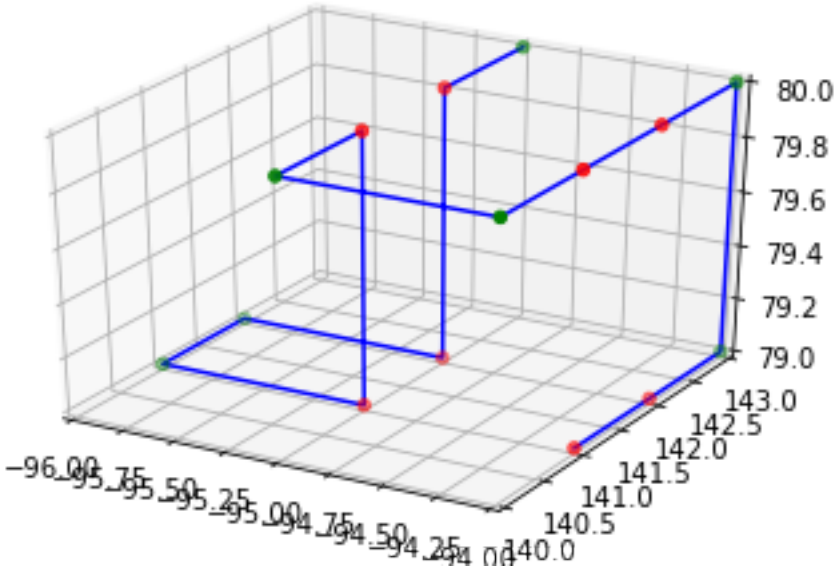


Initial energy is: 0

HBox(children=(FloatProgress(value=0.0, max=100000.0), HTML(value='')))

The lowest minimum is:






```
Energy is:-16
number of iterations:100001
```

[42]:
```
def createHPstring(lis):
    string=""
    for el in lis:
        if el==0:
```

```
            string+="H"
        else:
            string+="P"
    return string

sqnc= createHPstring(beads)


sequence_x= pd.Series(protein[0][0])
sequence_y= pd.Series(protein[0][1])
sequence_z= pd.Series(protein[0][2])

df1= pd.DataFrame()

df1["x"]= sequence_x
df1["y"]= sequence_y
df1["z"]= sequence_z


sequence_x= pd.Series(folded_protein[0])
sequence_y= pd.Series(folded_protein[1])
sequence_z= pd.Series(folded_protein[2])

df= pd.DataFrame()

df["x"]= sequence_x
df["y"]= sequence_y
df["z"]= sequence_z

fig = px.line_3d(df1, x='x', y='y', z='z',)
scatter=go.Scatter3d(x=df1['x'], y=df1['y'], z=df1['z'] ,␣
 ↪mode='markers',marker=dict(size=10,color=interaction))
fig = fig.add_trace(scatter)

fig.show()

fig1 = go.Scatter3d(x=df['x'], y=df['y'], z=df['z'] ,␣
 ↪mode='markers',marker=dict(size=15,color=interaction), )


fig = px.line_3d(df, x='x', y='y', z='z',height=600,title=" Iterations:␣
 ↪"+str(iterations)+"\n Sequence: "+sqnc+" Energy: "+str(min(minima)))
fig.add_trace(fig1 )
fig.update_traces(line=dict(width=8, color=interaction,))
fig.show()

energyPlot(list(energy_record.values()), list(energy_record.keys()))
```

```
energyPlot(list(minima.keys()), range(len(minima.keys())))
```

[44]:
```
for protein in minima:

    sequence_x= pd.Series(minima[protein][0])
    sequence_y= pd.Series(minima[protein][1])
    sequence_z= pd.Series(minima[protein][2])

    df= pd.DataFrame()

    df["x"]= sequence_x
    df["y"]= sequence_y
    df["z"]= sequence_z

    fig1 = go.Scatter3d(x=df['x'], y=df['y'], z=df['z'] ,␣
↪mode='markers',marker=dict(size=10,color=interaction), )


    fig = px.line_3d(df, x='x', y='y', z='z',height=600,title="Energy: "+␣
↪str(protein) +", Iterations: "+str(iterations)+"\n Sequence: "+sqnc)
    fig.add_trace(fig1 )
    fig.update_traces(line=dict(width=5, color=interaction,))
    fig.show()

energyPlot(list(energy_record.values()), list(energy_record.keys()))
energyPlot(list(minima.keys()), range(len(minima.keys())))
```

Here I am studying how different W affects the simulation, not that for simplicity I have used T instead of k*T, with k being the boltzmann constant

the first round of simulations is different proteins with increasing length. I have plotted the number of iterations required to reach the lowest minimum

the second round of simulations tries different values of W (t) to see whether there is an optimal value to fold the (same!) protein quicker than other values of t.

[123]:
```
def fold(l, iterations, T):
    n=l+20
    m=1

    interaction=makeColours3D(protein[1])

    total_energy_in=evaluateEnergy3D(np.array(protein[0]), protein[1])

    rotated_protein, beads=makeRandRotation3D(protein[0],protein[1])
    total_energy=evaluateEnergy3D(rotated_protein, protein[1])
```

```
    m_record={0:0}


    while m<iterations:
        for i in tqdm(range(iterations)):
            m+=1

            rotated_protein_new,␣
↪beads=makeRandRotation3D(rotated_protein,protein[1])
            total_energy_new=evaluateEnergy3D(rotated_protein_new, beads)

            W=np.exp(-(total_energy_new-total_energy)/T)

            R=np.random.randint(0,1000)/1000

            if total_energy_new-total_energy <0 :
                rotated_protein=rotated_protein_new
                total_energy=total_energy_new
                m_record[total_energy]=m

            elif R<W:
                rotated_protein=rotated_protein_new
                total_energy=total_energy_new

            else:
                pass


    return m_record[min(m_record)], min(m_record)
```

[117]:
```
metrics={"length":[],"iterations required":[]}
for l in range(5,35,5):
    protein=makeProtein3D(l+20 ,l,[])
    metrics["length"].append(l)
    metrics["iterations required"].append(fold(l, 10000,1)[0])
```

```
HBox(children=(FloatProgress(value=0.0, max=10000.0), HTML(value='')))


HBox(children=(FloatProgress(value=0.0, max=10000.0), HTML(value='')))


HBox(children=(FloatProgress(value=0.0, max=10000.0), HTML(value='')))
```

```
HBox(children=(FloatProgress(value=0.0, max=10000.0), HTML(value='')))
```

```
HBox(children=(FloatProgress(value=0.0, max=10000.0), HTML(value='')))
```

```
HBox(children=(FloatProgress(value=0.0, max=10000.0), HTML(value='')))
```

Note from the previous cell that the IT/s as a function of the increasing length of the simulated protein are displayed!! :-)

NOTE: 10k might not be enough at that "temperature", in fact many local minima have been found close to the max iteration

[118]:
```
print(metrics)

metrics_df= pd.DataFrame(metrics)
metrics_df.head()
fig = px.bar(metrics, y='iterations required', x= 'length')
fig.show()
```

```
{'length': [5, 10, 15, 20, 25, 30], 'iterations required': [0, 9529, 9762, 4970,
8252, 6019]}
```

[ ]:
```
"""In this cell I will plot the number of moves required to find the lowest
minimum, when the unfolding move is more or less likely.
The SAME protein will be used to simulate at different "temperature", namely␣
 ↪different prob. for the unfolding move"""

protein=makeProtein3D(25,15,[])
metrics_new={"temp":[],"iterations required":[], "energy reached":[]}
l=15
print(l)
for t in range(1,30,3):
    t=t/10
    metrics_new["temp"].append(t)
    metrics_new["iterations required"].append(fold(l, 10000,t)[0])
    metrics_new["energy reached"].append(fold(l, 10000,t)[1])
```

[129]:
```
print(metrics_new)
metrics_df_new= pd.DataFrame(metrics_new)
metrics_df_new.head()
```

```
fig = px.bar(metrics_new, y='iterations required', x= 'temp')
fig2 = px.line(metrics_new, y='energy reached', x= 'temp')
fig.show()
fig2.show()
```

```
{'temp': [0.1, 0.4, 0.7, 1.0, 1.3, 1.6, 1.9, 2.2, 2.5, 2.8], 'iterations
required': [9433, 9501, 9972, 8525, 9830, 4903, 5323, 7367, 9671, 4598], 'energy
reached': [-8, -8, -8, -8, -8, -8, -8, -8, -6, -8]}
```

It seems like t=1.5 allows the fastest folding with the lowest energy reached. At t=2.5 folding becomes less effective because too many unfolding moves are allowed. More study is required, with higher number of iterations.

In the next cell I will test it on a larger protein. Apologies for not creating a function to handle the folding process and plotting. I will do it asap

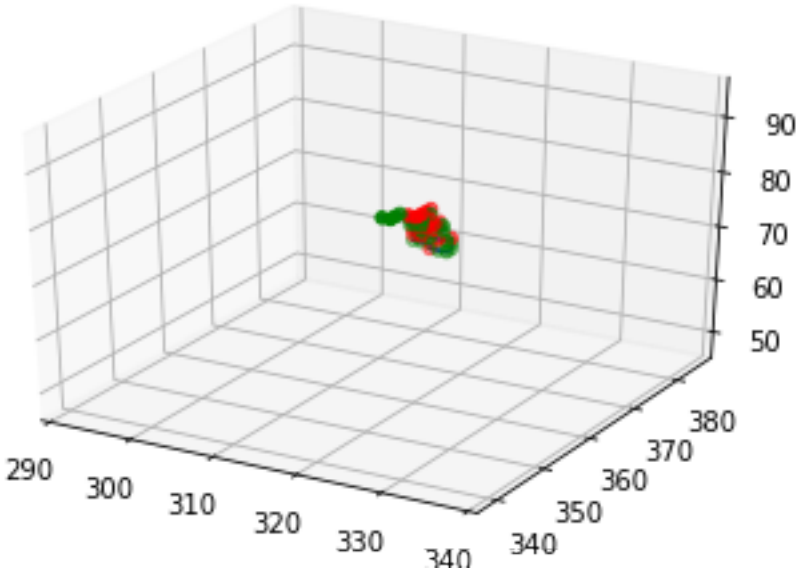
[137]:
```
"""Initial configuration and first rotation"""

n=70
l=50
protein=makeProtein3D(n,l,[])



'''The follwing is for plotting purposes only: '''
interaction=makeColours3D(protein[1])

total_energy_in=evaluateEnergy3D(np.array(protein[0]), protein[1])
print("Initial energy is: {}".format(total_energy_in))
rotated_protein, beads=makeRandRotation3D(protein[0],protein[1])
total_energy=evaluateEnergy3D(rotated_protein, protein[1])
print("Energy after the first rigid rotation is: {}".format(total_energy))




"""Cell is separated from the next one to allow a different number of␣
 ↪iterations on the same protein"""


m=1

minima={}
minima_beads={}
energy_record={}
minima[total_energy]=rotated_protein

iterations=50000
```

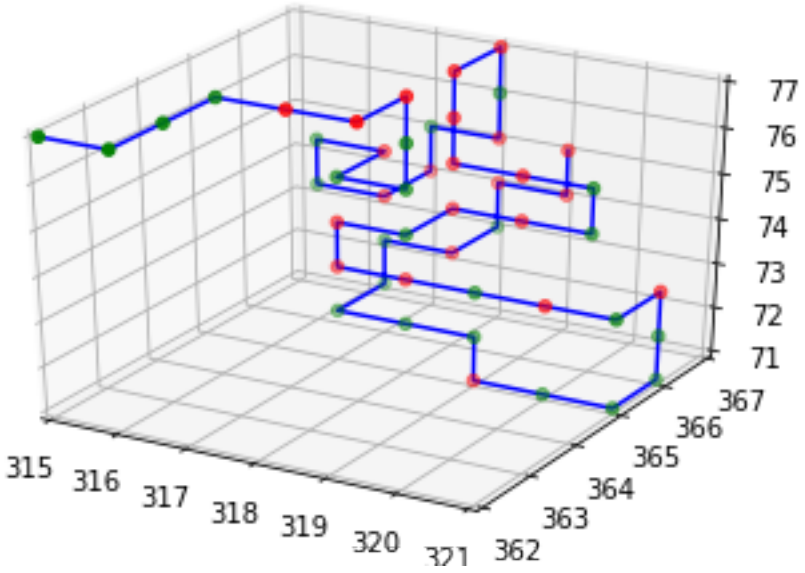
```
while m<iterations:
    for i in tqdm(range(iterations)):
        m+=1

        rotated_protein_new,␣
 ↪beads=makeRandRotation3D(rotated_protein,protein[1])
        total_energy_new=evaluateEnergy3D(rotated_protein_new, beads)
        energy_record[m]=total_energy_new
        W=np.exp(-(total_energy_new-total_energy)/1.5)

        R=np.random.randint(0,1000)/1000

        if total_energy_new-total_energy <0 :
            rotated_protein=rotated_protein_new
            total_energy=total_energy_new

            minima[total_energy]=rotated_protein
            minima_beads[total_energy]=beads

        elif R<W:
            rotated_protein=rotated_protein_new
            total_energy=total_energy_new
        else:
            pass


print("The lowest minimum is: ")
folded_protein=minima[min(minima)]
interaction=makeColours3D(minima_beads[min(minima)])

plotProtein3D(folded_protein[0],folded_protein[1],folded_protein[2],interaction,n,l)
plotProteinZoom3D(folded_protein[0],folded_protein[1],folded_protein[2],interaction,n,l)
print("Energy is:{} ".format(min(minima)))
print("number of iterations:{}".format(m))
```

```
Initial energy is: 0
Energy after the first rigid rotation is: 0
```

```
HBox(children=(FloatProgress(value=0.0, max=50000.0), HTML(value='')))
```

```
The lowest minimum is:
```






```
Energy is:-32
number of iterations:50001
```

[138]:
```
sequence_x= pd.Series(folded_protein[0])
sequence_y= pd.Series(folded_protein[1])
sequence_z= pd.Series(folded_protein[2])
```

```
df= pd.DataFrame()

df["x"]= sequence_x
df["y"]= sequence_y
df["z"]= sequence_z

fig = px.line_3d(df, x='x', y='y', z='z',)
scatter=go.Scatter3d(x=df['x'], y=df['y'], z=df['z'] ,␣
 ↪mode='markers',marker=dict(size=10,color=interaction))
fig = fig.add_trace(scatter)

fig.show()

fig1 = go.Scatter3d(x=df['x'], y=df['y'], z=df['z'] ,␣
 ↪mode='markers',marker=dict(size=15,color=interaction), )
```