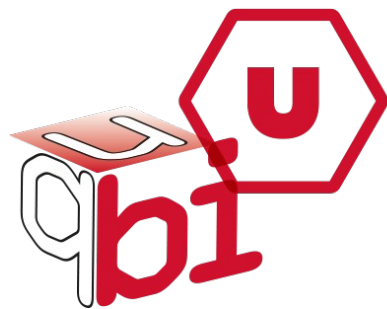


Internship Report Part II - Paweł Batóg

Research group - Description



QuBi

Quantitative Biolmaging

The Quantitative Biolmaging (QuBi) lab was formed in 2017 by Dr. Carlo Manzo (Ramón y Cajal Fellow) at the Faculty of Science and Technology of the UVic-UCC. It has been recognized as an emerging research group by the Government of Catalonia (2017SGR940)

The research group in Quantitative Biolmaging is focused to study molecular mechanisms underlying complex biological processes through advanced microscopy and image analysis techniques. The group aims at applying these technologies to characterize at the molecular level cellular signaling processes in health and disease, with the final objective of providing guidelines for the development of new approaches for nanodiagnostics and personalized medicine.

The group activity is carried out along three main lines of research, focused to the development of:

- i) localization microscopy techniques, including single particle tracking and super-resolution imaging
- ii) quantitative methods for the analysis and the treatment of super-resolution images and single-molecule data
- iii) stochastic approaches for the modeling of protein motion and organization

The research has a highly interdisciplinary character, bridging elements of physics, engineering and data science to tackle problems in biomedicine and cell biology.

Documentation

The documentation contains information about the Python programming language version, the version of the external library installer - PIP and all libraries that are not in the standard Python library. In addition, the documentation will contain all information about the functions that appear in the program and the overall structure of the application. As it is written in the first report, the program is divided into four types of functions. They will also be described in this way here.

Required libraries

List of all libraries

```
from PyQt5 import QtCore, QtGui, QtWidgets
import numpy as np
import scipy as scipy
import matplotlib.pyplot as plt
import pylab
from itertools import chain
import random
from scipy.spatial.distance import pdist, squareform
from scipy.special import erf
from PIL import Image
import os
import glob
import copy
import shutil
```

Each external library was installed using **PIP 19.0.3** working in environment **Python 3.7**.

Name	Version
PyQt5	5.11.3
PyQt5 - tools	5.11.3.1.4
Numpy	1.16.2
Scipy	1.2.1
Matplotlib	3.0.3
Pillow	6.0.0

If the library is not included in the table above, it means that it should be installed automatically when configuring the Python 3.7 environment and no additional installation is needed.

Purpose of required libraries

PyQt5 – The most important library which is used for creating the interface of application.

PyQt5 - tools – This library is not used directly in the application but allows to use Designer application. In particular, it allows to read .ui files.

Numpy / Scipy -- Package for scientific computing with Python. Allows to perform complicated calculations easily.

Matplotlib – Package that is used in creating and visualizing mathematical graphs.

Itertools – Library that offers previously prepared functions for solving more complicated problems.

Random – A package introducing the concept of randomness in the application.

Pillow – A library that allows to perform efficient activities in the pictures - creating, adding and conversions.

Os / Shutil / Glob– Libraries that allow to work on the files located on the hard drive. For example creating folders and deleting them.

Copy - The library used when creating an identical independent copy of one of the variables in the program.

Global variables

These are variables that will work over the entire application. This means that they are not limited to a specific function but for all functions.

```
random.seed(1209)

N_mols = 100          # number of molecules
frames = 20           # number of generated frames
points = 30           # number of visible points

pix_size = 110        # pixel size
pixels=32             # number of pixels
stdev = np.true_divide(25,pix_size) # super_res microscope localiz precision
min_d = np.true_divide(5,pix_size)  # molecular diameter in pixels

ren_fact=5            # image rendering factor
```

Random seed - Function that provide application the same random probability even in different opening of the program.

N_mols, frames, points - The only variables that are manipulated by user through the user interface. Contain information about number of generated molecules, number of frames that program is going to generate and number of visible points on one frame.

Pix_size, pixels, stdev, min_d, ren_fact - Variables used in creating frame/image process. Correcting these parameters allows to change the resolution of the generated frame. Overdoing it with high resolution causes a significant slowdown in the frame generation algorithm.

Interface functions

Setup_UI function is used to create the graphical site of application. Each code line describes the dimensions, content and layout of each widget separately.

As long as there is no need to change the graphical user interface, there is no need to intervene in this part of the

```
def setupUI(self, ImageWindow):
    ImageWindow.setObjectName("ImageWindow")
    ImageWindow.resize(1286, 736)
    self.gridLayout_9 = QtWidgets.QGridLayout(ImageWindow)
    self.gridLayout_9.setObjectName("gridLayout_9")
    self.gridLayout_2 = QtWidgets.QGridLayout()
    self.gridLayout_2.setObjectName("gridLayout_2")
    self.gridLayout = QtWidgets.QGridLayout()
    self.gridLayout.setObjectName("gridLayout")
    self.label_6 = QtWidgets.QLabel(ImageWindow)
    self.label_6.setAlignment(QtCore.Qt.AlignCenter)
    self.label_6.setObjectName("label_6")
    self.gridLayout.addWidget(self.label_6, 2, 1, 1, 1)
    self.label_7 = QtWidgets.QLabel(ImageWindow)
    self.label_7.setObjectName("label_7")
    sizePolicy = QtWidgets.QSizePolicy(QtWidgets.QSizePolicy.Expanding, QtWidgets.QSizePolicy.Expanding)
    sizePolicy.setHorizontalStretch(50)
    sizePolicy.setVerticalStretch(50)
    self.gridLayout_2.addWidget(self.label_7, sizePolicy().hasHeightForWidth())
    self.label_7.setSizePolicy(sizePolicy)
    self.label_7.setMinimumSize(QtCore.QSize(200, 200))
    self.label_7.setMaximumSize(QtCore.QSize(400, 400))
    self.label_7.setSizeIncrement(QtCore.QSize(0, 0))
    self.label_7.setText("")
    self.label_7.setObjectName("label_7")
    self.gridLayout.addWidget(self.label_7, 4, 1, 1, 1)
    spacerItem = QtWidgets.QSpacerItem(20, 40, QtWidgets.QSizePolicy.Minimum, QtWidgets.QSizePolicy.Expanding)
    self.gridLayout.addItem(spacerItem, 5, 1, 1, 1)
    spacerItem1 = QtWidgets.QSpacerItem(40, 20, QtWidgets.QSizePolicy.Expanding, QtWidgets.QSizePolicy.Minimum)
    self.gridLayout.addItem(spacerItem1, 4, 2, 1, 1)
    spacerItem2 = QtWidgets.QSpacerItem(20, 40, QtWidgets.QSizePolicy.Minimum, QtWidgets.QSizePolicy.Expanding)
    self.gridLayout.addItem(spacerItem2, 5, 1, 1, 1)
    spacerItem3 = QtWidgets.QSpacerItem(40, 20, QtWidgets.QSizePolicy.Expanding, QtWidgets.QSizePolicy.Minimum)
    self.gridLayout.addItem(spacerItem3, 4, 0, 1, 1)
    self.gridLayout_2.addLayout(self.gridLayout, 0, 0, 1, 1)
    self.gridLayout_3 = QtWidgets.QGridLayout()
    self.gridLayout_3.setObjectName("gridLayout_3")
    spacerItem4 = QtWidgets.QSpacerItem(40, 20, QtWidgets.QSizePolicy.Expanding, QtWidgets.QSizePolicy.Minimum)
    self.gridLayout_3.addItem(spacerItem4, 2, 2, 1, 1)
    self.label_9 = QtWidgets.QLabel(ImageWindow)
    sizePolicy = QtWidgets.QSizePolicy(QtWidgets.QSizePolicy.Expanding, QtWidgets.QSizePolicy.Expanding)
    sizePolicy.setHorizontalStretch(50)
    sizePolicy.setVerticalStretch(50)
    self.gridLayout_3.addWidget(self.label_9, sizePolicy().hasHeightForWidth())
    self.label_9.setSizePolicy(sizePolicy)
    self.label_9.setMinimumSize(QtCore.QSize(200, 200))
    self.label_9.setMaximumSize(QtCore.QSize(400, 400))
    self.label_9.setText("")
    self.label_9.setObjectName("label_9")
    self.gridLayout_3.addWidget(self.label_9, 2, 1, 1, 1)
    self.label_8 = QtWidgets.QLabel(ImageWindow)
    self.label_8.setAlignment(QtCore.Qt.AlignCenter)
    self.label_8.setObjectName("label_8")
    self.gridLayout_3.addWidget(self.label_8, 0, 1, 1, 1)
    spacerItem5 = QtWidgets.QSpacerItem(20, 40, QtWidgets.QSizePolicy.Minimum, QtWidgets.QSizePolicy.Expanding)
    self.gridLayout_3.addItem(spacerItem5, 5, 1, 1, 1)
    spacerItem6 = QtWidgets.QSpacerItem(20, 40, QtWidgets.QSizePolicy.Minimum, QtWidgets.QSizePolicy.Expanding)
    self.gridLayout_3.addItem(spacerItem6, 1, 1, 1, 1)
    spacerItem7 = QtWidgets.QSpacerItem(40, 20, QtWidgets.QSizePolicy.Expanding, QtWidgets.QSizePolicy.Minimum)
    self.gridLayout_3.addItem(spacerItem7, 2, 0, 1, 1)
    self.gridLayout_2.addLayout(self.gridLayout_3, 0, 1, 1, 1)
    self.gridLayout_4 = QtWidgets.QGridLayout()
    self.gridLayout_4.setObjectName("gridLayout_4")
    spacerItem8 = QtWidgets.QSpacerItem(40, 20, QtWidgets.QSizePolicy.Expanding, QtWidgets.QSizePolicy.Minimum)
    self.gridLayout_4.addItem(spacerItem8, 2, 2, 1, 1)
    self.label_10 = QtWidgets.QLabel(ImageWindow)
    sizePolicy = QtWidgets.QSizePolicy(QtWidgets.QSizePolicy.Expanding, QtWidgets.QSizePolicy.Expanding)
```

code. If there is a need to change the graphical interface site of the application, there are two ways to change it.

The first one is to manually change the code, the second to read the .ui file to the Designer application and then generate a new .py file. Its disadvantage is that it requires manual rewriting of all the functions of the algorithm after conversion of the .ui file.

In the case of a satisfactory user interface, the only place where there is need to change the code is where programmer wants to assign

```
# ON PUSH BUTTON/SLIDER EVENTS
self.pushButton_4.clicked.connect(self.CreateImages)           # OnPlayButton event
self.horizontalSlider.valueChanged.connect(self.LoadImages)    # OnChangeValueOfSlider event
self.pushButton.clicked.connect(self.NextFrame)               # OnNextButton event
self.pushButton_2.clicked.connect(self.PreviousFrame)          # OnPreviousButton event
```

the action to the buttons.

In the code above, each line corresponds to a different widget. The first – pushButton4 – refers to “Play” button and activate CreateImages function. The second horizontalSlider describe the action that has to be taken when the value of slider is changed. The third and the forth are “Next” and “Previous” buttons which refers to NextFrame and PreviousFrame functions. In each case, the function names to which the widgets refer are stored inside the final parentheses.

The last function which cooperates in creating a graphical interface is the **Retranslate_UI** function.

```
#CHANGING DEFAULT TEXT IN LABELS - START
def retranslateUi(self, ImageWindow):
    _translate = QtCore.QCoreApplication.translate
    ImageWindow.setWindowTitle(_translate("ImageWindow", "Form"))
    self.label_6.setText(_translate("ImageWindow", "Super Resolution Image"))
    self.label_8.setText(_translate("ImageWindow", "Super Resolution Sum Image"))
    self.label_11.setText(_translate("ImageWindow", "Conventional Image"))
    self.pushButton_2.setText(_translate("ImageWindow", "Previous"))
    self.pushButton.setText(_translate("ImageWindow", "Next"))
    self.label_12.setText(_translate("ImageWindow", "Plot Image"))
    self.label_15.setText(_translate("ImageWindow", "Points"))
    self.label_17.setText(_translate("ImageWindow", "Final Plot Image"))
    self.label_18.setText(_translate("ImageWindow", "Frame: "))
    self.label_19.setText(_translate("ImageWindow", "--"))
    self.label_30.setText(_translate("ImageWindow", "CONSOLE: "))
    self.label_20.setText(_translate("ImageWindow", "Number of molecules"))
    self.lineEdit.setInputMask(_translate("ImageWindow", "9999"))
    self.lineEdit_3.setInputMask(_translate("ImageWindow", "9999"))
    self.lineEdit_8.setInputMask(_translate("ImageWindow", "9999"))
    self.label_23.setText(_translate("ImageWindow", "Points/Frame"))
    self.label_22.setText(_translate("ImageWindow", "Frames"))
    self.label.setText(_translate("ImageWindow", "Options"))
    self.pushButton_4.setText(_translate("ImageWindow", "Play"))
    self.lineEdit_3.setText(_translate("ImageWindow", "20"))
    self.lineEdit_8.setText(_translate("ImageWindow", "30"))
    self.lineEdit.setText(_translate("ImageWindow", "100"))
    print('RetranslateUI - DONE')
    self.CreateFolders()
#CHANGING DEFAULT TEXT IN LABELS - END
```

This function determines the text content of individual widgets as well as the limitations of input data.

All widgets are described at the very beginning of the .py file. Description contains information and its ID number about each specific and important from the programming point of view widget.

```
# The individual elements are responsible for:
# label_16, _14, _13, _7, _9, _10 - Showing images,
# label_19 - Informations about frame,
# label_30 - Informations about wrong/good Input Data
# line_edit(0), _3, _8 - INPUT for N_mols, frames, points
# pushButton_4 - Generate frames
# pushButton(0), _2 - Next/Previous frame
```

Functions called directly by users

Functions triggered by the user in a direct way, e.g. pressing a button or moving the slider. None of these functions generate frames. On the other hand, these functions can be used to load pictures or call functions that generate frames.

Create_Images

```
def CreateImages(self):
    print('Function')
    Mol_N = int(self.lineEdit.text())
    print('N_mols: ', Mol_N)
    Frame_N = int(self.lineEdit_3.text())
    print('Frames: ', Frame_N)
    Points_N = int(self.lineEdit_8.text())
    print('Points: ', Points_N)

    if Mol_N >= Points_N and Frame_N >= 10 and Points_N != 0:
        [xn, yn, zn] = self.generate_molecules(pixels, Mol_N, min_d)
        print('Points - DONE.')
        self.frame_simulation(pixels, xn, yn, stdev, Frame_N, Points_N)
        print('Images - DONE.')
        self.Refresh()
        self.horizontalSlider.setMaximum(Frame_N-1)
        self.label_30.setText("CONSOLE: OK")
    elif Mol_N < Points_N:
        self.label_30.setText("CONSOLE: N_mols to low")
    elif Frame_N < 10:
        self.label_30.setText("CONSOLE: Frames to low (min = 10)")
    elif Points_N == 0:
        self.label_30.setText("CONSOLE: Points to low (min = 1)")

# After Clicking Play Button Function
# Read Input Mol_N
# Read Input Frame_N
# Read Input Points_N
# Conditions to enter the function
# Calling generate molecules function
# Calling generate frames function - passing points generated above
# Refresh images function
# Wrong parameters
# Wrong parameters
# Wrong parameters
```

Create_Images is one of the most important function that is activated when the “Play” button is pressed. This function read the user input data such as number of molecules, how many frames user wants to generate and how many points should appear on one frame.

Furthermore this function checks if all of the required conditions are met. If the conditions have been met, the function is going to call functions

that generate molecules and frames. If no, the function will return one of the “Wrong parameters” messages.

Next_Frame/Previous_Frame

```
def NextFrame(self):
    frame = self.horizontalSlider.value()
    frame = frame + 1
    self.horizontalSlider.setValue(frame)

def PreviousFrame(self):
    frame = self.horizontalSlider.value()
    frame = frame - 1
    self.horizontalSlider.setValue(frame)
```

Next frame function
Takes value from the slider
Increment the value from slider
Assigns the value to slider

Previous frame function
Takes value from the slider
Decrement the value from slider
Assigns the value to slider

Next_Frame and **Previous_Frame** functions describe the action the program will execute after pressing the buttons “Next” and “Previous”. In this case, calling these functions will result in assigning one or more values to the slider, respectively. It is not required to call the frame change function because the program calls the function **Load_Images** in case when value of the slider is changed.

Load_Images

```
def LoadImages(self):
    frame = self.horizontalSlider.value()
    frame = frame+1

    filename = 'image' + str(frame)
    script_dir = os.path.dirname(__file__)
    results_dir = os.path.join(script_dir, 'Plot/')
    results_dir2 = os.path.join(script_dir, 'Image/')
    results_dir3 = os.path.join(script_dir, 'SumImage/')

    self.label_19.setText(str(frame))

    pixmap = QtGui.QPixmap('Plot\FinalImage.png')
    pixmap = pixmap.scaled(self.label_16.width(), self.label_16.height())
    self.label_16.setPixmap(pixmap)
    self.label_16.repaint()
    self.label_16.parentWidget().repaint()
```

Takes value from the slider
Increment value from slider (Slider counts from 0 but we want 1 frame)

Create variable that contain name of the image file
Create paths to different folders with images
#

Set number of appearing frame in the output widget

Read the image from the folder
Scale the size of the image to the size of the label
Set new image in the label
Refresh the image in the label (in case of reading two different images with the same name
Does the same but for the parent of the label widget
Rest of the functions work the same as above

This function initially reads the number of the frame which will then be loaded into individual labels. Then the function finds paths leading to

individual images. In the meantime, the function sets the current frame number on the output.

In addition, when calling this function during loading the image, the function sets its width and height to the dimensions of label. This means that when user changes the dimensions of the application window, all images will always be matched to it.

Functions called by interface functions

In this part there is a function that generates the frames. In addition, there are functions that create folders when user run the software and function that provide additional refreshing images when **Load_Images** function is called.

Frame_Simulation

```
def frame_simulation(self,pixels,xn,yn,stdev, howmanyframes, howmanypoints):

    # Create path for generating images
    script_dir = os.path.dirname(__file__)
    results_dir = os.path.join(script_dir, 'Plot/')
    results_dir2 = os.path.join(script_dir, 'Image/')
    results_dir3 = os.path.join(script_dir, 'SumImage/')

    # Delete files in the folders (delete all files even not images)
    files = glob.glob('Plot/*')
    for f in files:
        os.remove(f)
    files = glob.glob('Image/*')
    for f in files:
        os.remove(f)
    files = glob.glob('SumImage/*')
    for f in files:
        os.remove(f)
    print('Dir_Files - DONE')

    points = 0
    n=xn.shape[-1]
    x=[]
    y=[]

    # Bool whether the Points image was rendered or not

    localization=np.random.geometric(p=0.2, size=n)

    # Generate random localizations of points

    for i in range(0,n):
        Xij = np.random.normal(loc=0,scale=stdev, size=int(localization[i]))
        Yij = np.random.normal(loc=0,scale=stdev, size=int(localization[i]))
        x.append(xn[i] + Xij)
        y.append(yn[i] + Yij)

    # Assign X
    # Assign Y
    # Append on x and y lists

    x_unchain= np.asarray(list(chain.from_iterable(x)))
    y_unchain= np.asarray(list(chain.from_iterable(y)))

    plot_memory_xn = 0
    plot_memory_yn = 0
    plot_memory = []

    # These functions stores points and coordinates
    # where they appeared (used in generating FinalImage)

    SUM_IM = 0

    # SUM_IMAGE Variable
```

This function is modified function that was provided by Profesor Carlo Manzo. Function consists of two parts. The first is preparation for image rendering. The second is to generate image. The second begins when the first loop is created which does not appear on the image above.

The **Frame_Simulation** function creates the path to the folders in which the generated images will be stored. Then it removes the entire contents of the folder in order to free the space for new images (NOTE - The function deletes all files without exception).

The variables responsible for storage of all the points of occurrence of molecules are generated in this function. Points of occurrence are generated in the same function according to a random pattern.

```

for image in range(0,howmanyframes):                                # This loop will be executed N_Frames times

    [newxn,newyn] = self.random_movement(xn,yn,n)                    # Generate random movement in exact frame
    xy = random.sample(range(newxn.shape[0]), howmanypoints)        # Take randomly N_Points from generated random movement points

    # Save points to memory
    plot_memory = np.append(plot_memory,xy)
    plot_memory_xn = np.append(plot_memory_xn, newxn)
    plot_memory_yn = np.append(plot_memory_yn, newyn)

    if points == 0:
        # Generate POINTS_IMAGE
        fig = plt.figure(1, figsize=(5.5,5.5))
        plt.xlim((0,pixels))
        plt.ylim((0,pixels))
        plt.scatter(xn,yn,c='b',marker='+')
        plt.axis('off')
        plt.gca().invert_yaxis()
        plt.savefig(results_dir + 'pointsImage', bbox_inches='tight')
        points = 1
        plt.close()

        # Generate PLOT_IMAGE
        fig = plt.figure(1, figsize=(5.5,5.5))
        plt.scatter(newxn[xy],newyn[xy],c='r',marker='.')
        plt.xlim((0,pixels))
        plt.ylim((0,pixels))
        plt.axis('off')
        plt.gca().invert_yaxis()
        filename = 'image' + str(image+1)
        plt.savefig(results_dir + filename, bbox_inches='tight')
        plt.close()

        # Generate IMAGE
        IM = self.map_into_image(newxn[xy],newyn[xy],pixels,ren_fact, stdev)
        im = Image.fromarray(IM)
        im.save(results_dir2 + filename + ".TIFF")

        # Generate SUMIMAGE
        SUM_IM = np.float64(IM) + np.float64(SUM_IM)
        SUM_IM = SUM_IM / SUM_IM.max()
        SUM_IM = np.uint16(SUM_IM*(2**16-1))

        im = Image.fromarray(SUM_IM)
        im.save(results_dir3 + filename + ".TIFF")
        print('Created Images: ',image+1)

    # Generate FINALIMAGE
    fig = plt.figure(1, figsize=(5.5,5.5))
    plt.scatter(plot_memory_xn,plot_memory_yn,c='r',marker='.')
    plt.scatter(xn,yn,c='b',marker='+')
    plt.axis('equal')
    plt.xlim((0,pixels))
    plt.ylim((0,pixels))
    plt.gca().invert_yaxis()
    plt.axis('off')
    filename = 'zFinalImage'
    plt.savefig(results_dir + filename, bbox_inches='tight')
    plt.close()

    # Generate CONVING
    filename = 'convimg'
    WT = self.map_into_image(xn,yn,pixels,1, 240/pix_size)
    wt = Image.fromarray(WT)
    wt.save(results_dir3 + filename + ".TIFF")

self.horizontalSlider.setValue(0)                                # Set slider value to 0
return [x_unchain,y_unchain]

```

In the second part of the function, they are sequentially looped PLOT_IMAGE, IMAGE and SUM_IMAGE. These are only images that are generated in each loop. This is due to the fact that for each subsequent loop they contain a different set of points. Also in the form of a list all

points that have occurred during the operation of all loops are also saved in the loop. These points will be used when creating FINAL_IMAGE image. In the first loop there is created POINTS_IMAGE image which can be generated only once because function do not change the mother position of molecules. The last generated image is CONVIMG which need only information about mother position of points. The last line assign 0 to slider value.

Create_Folders

```
def CreateFolders(self):  
    if os.path.exists("./Image"):          # check if path exist, if yes  
        shutil.rmtree("./Image")          # remove content  
    if os.path.exists("./Plot"):           # repeat it for every path  
        shutil.rmtree("./Plot")  
    if os.path.exists("./SumImage"):       # repeat it for every path  
        shutil.rmtree("./SumImage")  
  
    os.mkdir("./Image")                   # Creat folders  
    os.mkdir("./Plot")  
    os.mkdir("./SumImage")
```

Create_Folders function is a simple function that is called for the first time when the application is started. Checks if there are folders in which images will be saved in the future. If yes - the function deletes the folders along with the content. If no it does nothing. Then creates empty folders again.

Refresh

```
def Refresh(self):  
    app.processEvents()                  # Check Events in the application  
    for image in range(0,20):           # Loop start  
        self.horizontalSlider.setValue(image)  # Change the value of slider  
    self.horizontalSlider.setValue(0)    # Value of slider = 0  
    self.LoadImages()                   # Call Load_Images functions
```

Refresh function is called whenever the user decides to generate the frames again. This function makes the application load the same photos repeatedly. The existence of this function is caused by the fact that PyQt5 inherently stores information about currently loaded images - this means that when user wants to load new images with the same name, the program may decide to show the same pictures. This is because they have the same name. This is a little tricky but working way to avoid this uncomfortable property.

Calculation functions

These functions are used to perform all calculations. In particular, they create mother points, generate random occurrence of molecules, generate images and rewrite to common form images.

Random_Movement

```
def random_movement(self,xn,yn,n):
    newxn = copy.copy(xn)
    newyn = copy.copy(yn)
    for point in range(0,n):
        newxn[point] = newxn[point] + np.random.normal(loc=0,scale=stdev, size=1)
        newyn[point] = newyn[point] + np.random.normal(loc=0,scale=stdev, size=1)
    return [newxn,newyn]
```

Generate exact copy of mother points
We do this to avoid modification of mother points
For number of molecules:
Generate random x coordinate
Generate random y coordinate
Return new coordinates

This function is called in **Frame_Simulation** function. It is used to generate random occurrence of the points. **Frame_Simulation** call this function N_Frames times. The function itself creates an identical copy of the mother points and then assigns a new occurrence of the point in its environment. Function returns generated points.

Generate_Molecules

```
def generate_molecules(self,pixels,n,min_d):
    xn=np.random.uniform(0,(pixels-1),n)
    yn=np.random.uniform(0,(pixels-1),n)
    z=np.stack((xn,yn), axis=-1)
    D = squareform(pdist(z)) + 1000*min_d*np.identity(n)

    while D.min()<min_d:
        ind1, ind2 = np.where(D < min_d)
        unique = (ind1 < ind2)
        ind1 = ind1[unique]

        xn[ind1]=xn[ind1]+np.random.normal(0,min_d,ind1.shape[-1])
        yn[ind1]=yn[ind1]+np.random.normal(0,min_d,ind1.shape[-1])
        z=np.stack((xn,yn), axis=-1)
        D = squareform(pdist(z)) + 1000*min_d*np.identity(n)

    return [xn,yn]
```

Generate Molecules function - called in CreateImages
Draw samples from a uniform distribution

A function provided by Professor Carlo Manzo. It is used to generate mother points, which then are used to generate frames. This function is only called in the **Create_Images** function. Then the points are passed to **Frame_Simulation** function.

MolKernel

```
def MolKernel(self,x_inx,y_inx,x_pos,y_pos,stdev):  
    [xx,yy] = np.meshgrid(x_inx,y_inx)  
    img_kernel = (0.5*(erf(np.true_divide((xx-x_pos+0.5),(np.sqrt(2)*stdev)))-erf(np.true_divide((xx-x_pos-0.5),  
    (np.sqrt(2)*stdev)))))*(0.5*(erf(np.true_divide((yy-y_pos+0.5),(np.sqrt(2)*stdev)))-erf(np.true_divide((yy-y_pos-0.5),(np.sqrt(2)*stdev)))))  
    img_kernel=img_kernel/img_kernel.sum()  
    return img_kernel
```

A function provided by Professor Carlo Manzo. **MolKernel** is strongly associated with the process of mapping the image. Normalizes the image mapping so that the final effect is devoid of artifacts. Used in **Map_Into_Image** function.

Map_Into_Image

```
def map_into_image(self,x,y,pixels, fact, st):  
    M=np.zeros((pixels*fact,pixels*fact))  
    xx=np.linspace(0,fact*pixels-1,fact*pixels)  
    yy=np.linspace(0,fact*pixels-1,fact*pixels)  
    for i in range(0,x.shape[-1]):  
        img=self.MolKernel(xx,yy,fact*x[i],fact*y[i],fact*st)  
        M=M+img  
    M=M/M.max()  
    return np.asarray(np.uint16(M*(2**16-1)))
```

A function provided by Professor Carlo Manzo. It maps the array of information stored in the variable to the image form – recognized by Pillow library. Appears in **Frame_Simulation** function.

Beginning and the end of the application

Despite many lines of code, calling an application comes down to a few lines. In these lines there is a condition to correctly call the application, calling the application, removal of folders when closing the application and the process of closing the application.

```
if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    ImageWindow = QtWidgets.QWidget()
    ui = Ui_ImageWindow()
    ui.setupUi(ImageWindow)
    ImageWindow.show()
    print('Window Called - DONE')
    ret = app.exec_()

    # Delete Created Dictionaries
    shutil.rmtree("./Image")
    shutil.rmtree("./Plot")
    shutil.rmtree("./SumImage")

    sys.exit(ret)
```

Remove content of these folders

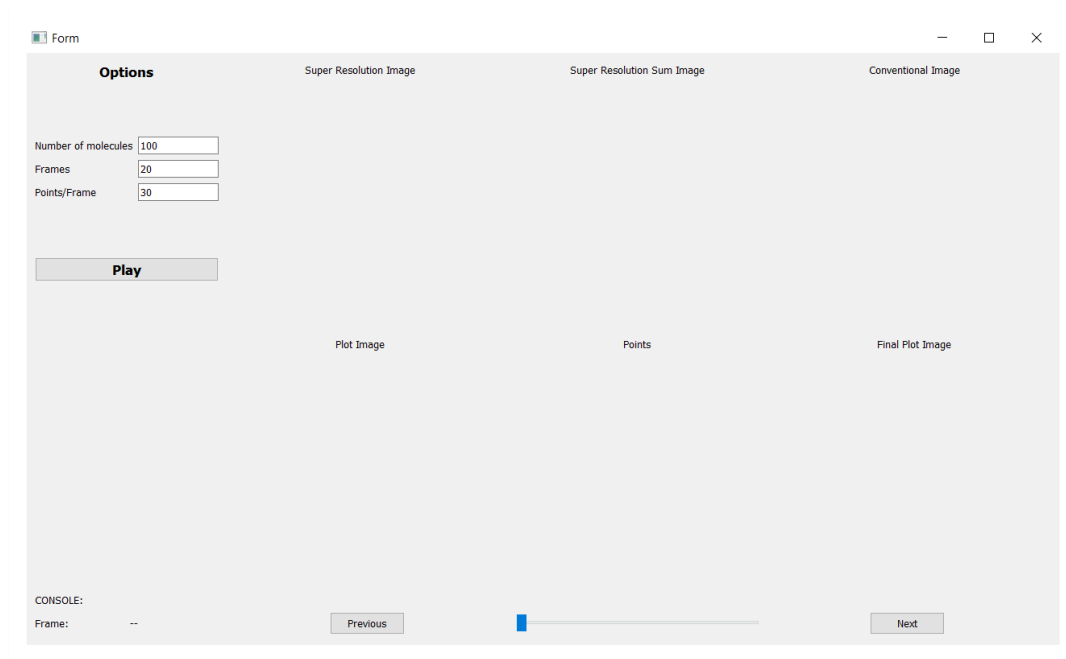
Important points in the operation of the application

The code covers individual calls of the "print" function which informs about the correct execution of a given function or the adoption of input data. The program running in the IDLE environment will be able to monitor not only the graphical interface of the application but also the console which will display the called print functions. The correct operation of the application should look as follows.

```
RetranslateUI - DONE
Generate Window - DONE
Window Called - DONE
Function
N_mols: 100
Frames: 20
Points: 30
Points - DONE.
Dir_Files - DONE
Created Images: 1
Created Images: 2
Created Images: 3
Created Images: 4
Created Images: 5
Created Images: 6
Created Images: 7
Created Images: 8
Created Images: 9
Created Images: 10
Created Images: 11
Created Images: 12
Created Images: 13
Created Images: 14
Created Images: 15
Created Images: 16
Created Images: 17
Created Images: 18
Created Images: 19
Created Images: 20
Images - DONE.
```

Application use

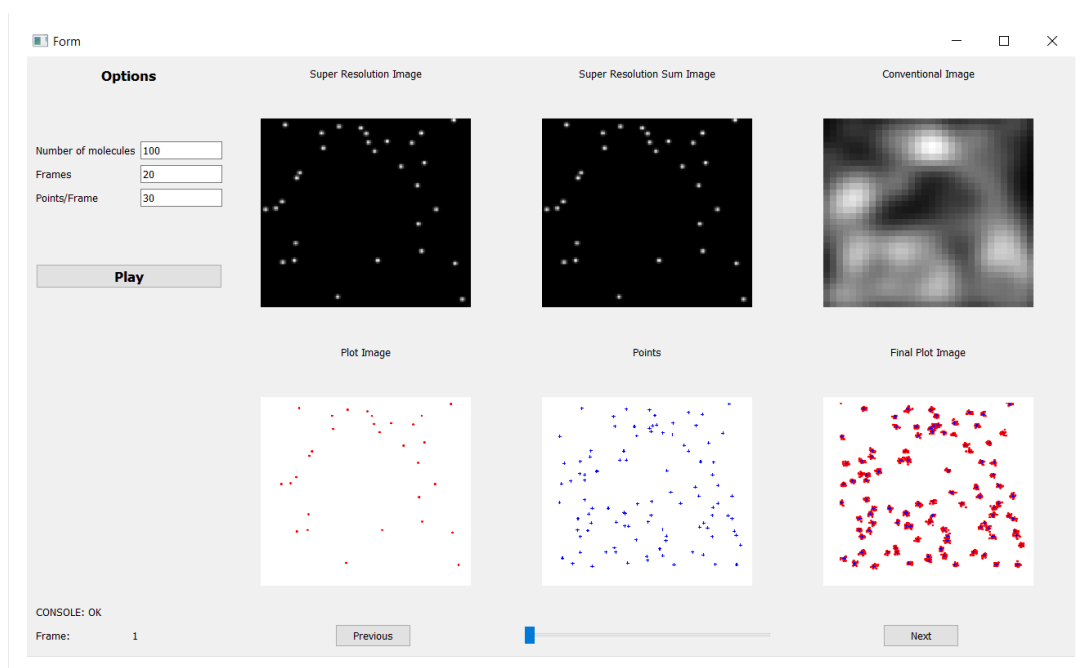
Using the application is extremely simple. The entire interface consists of just a few buttons and one slider. They enable the user to



The screenshot shows the application window titled "Form". It features a "Options" panel on the left with three input fields: "Number of molecules" set to 100, "Frames" set to 20, and "Points/Frame" set to 30. Below these is a "Play" button. The main area contains six placeholders for images: "Super Resolution Image", "Super Resolution Sum Image", "Conventional Image", "Plot Image", "Points", and "Final Plot Image". At the bottom, there is a "CONSOLE:" section with "Frame:" set to "--", a "Previous" button, a slider, and a "Next" button.

interact with the software.

After entering the appropriate options, press the "Play" button. This will trigger a series of algorithms that will start generating frames. Images



This screenshot shows the application after the "Play" button has been pressed. The image placeholders are now filled with generated data: "Super Resolution Image" and "Super Resolution Sum Image" show sparse white points on a black background; "Conventional Image" shows a blurred grayscale image; "Plot Image" shows red points on a white background; "Points" shows blue points on a white background; and "Final Plot Image" shows red points on a white background. The "CONSOLE:" section at the bottom now shows "Frame: 1". The "Previous" and "Next" buttons are still present.

will be saved in folders that will be created in the folder that contains the application file.

After generating the frames, user can change them using the **Next** and **Previous** buttons or by using the slider located at the bottom of the application. If user wants to generate new frames, set the options that are satisfactory and press the "Play" button again.

After using the application, it should be turned off like in the other system applications. Depending on the operating system, this method may vary, but usually it is a cross in the upper right corner of the application.



If user enter incorrect input data, the application will inform the user about the cause of the error showing one of the error messages.

CONSOLE: N_mols to low

CONSOLE: Frames to low (min = 10)

CONSOLE: Points to low (min = 1)

After the operation of generating the frames, the application should show the message: **CONSOLE: OK.**

In addition, there is a field in the application that shows the currently displayed frame.

Frame: 5

The minimum and required input data that the program accepts is:

- i) Minimal value of 10 frames.
- ii) Number of points higher than 0, less or equal to value of number of molecules.
- iii) Minimal value of 1 molecules.

Conclusion

Software development is not an easy process. It requires not only programming knowledge but also a strategic and systematic approach to the discussed issue. Creating an application requires creative thinking and the right approach in the same measure as coding.

It is not only important to know the functions that need to be implemented in the software being developed. It is equally important to think about a convenient interface that will ensure comfortable use of the application, the time at which the application performs its calculations and tests that check whether the software is reliable even in uncontrolled conditions.

What surprises the most is the fact how much work on the application prevails around it. This is, for example, finding the right environment, writing concise and meaningful documentation or commenting on the most important lines of code.

A well-written application will reliably serve its users. The created application will allow to be an irreplaceable tool in situations in which it is necessary to simulate and show the behavior of molecules. In addition, it works well at the moment when there is no access to advanced equipment that will allow you to see protein behavior live or in situations where person wants to show images known in the laboratory but for a larger audience which laboratory would not fit.

Software development is a path that should not be overcome as soon as possible. It is necessary to approach the topic of creating applications slowly and gradually. Only this action will allow to create a program that will be both well-written and well-structured. In addition, this approach will make such software reliable even in the heaviest environment.

Bibliography

- 1) <https://www.python.org>
- 2) <https://www.riverbankcomputing.com/static/Docs/PyQt5/>
- 3) <https://www.qt.io>
- 4) <https://stackoverflow.com>
- 5) <https://docs.scipy.org/doc/>
- 6) <https://pillow.readthedocs.io/en/stable/>