Healage Kinect Developer Manual

Table of Contents

Overview

The Healage Kinect System was developed using Python 3.9 with the PykinectV2[1] and the Microsoft Kinect V2 SDK[2]. The software requires the XBOX Kinect V2, along with the python3 wrapper and the SDK mentioned above, further requirements will be listed in the requirements section, also take a look at the requirements.txt as that also lists all necessary python3 libraries. The overall program has the ability to run on multiple patients without closing the full application. For now, please be aware that in this manual, multiple runs of either the gait or the kyphosis index measurement on a single patient will be referred to as an **instance**.

Hardware Requirements

- USB 3.0 Port or USB 3.0 Hub with external power supply.
- Windows 10 (Recommended) or Above[1]
- Physical Dual Core 3.1 GHz or Faster Processor[2]
- 4 GB of Memory
- 64-bit Processor (Note that ARM chips are not support by this software, as the SDK only supports intel chips)
- DX 11 Capable Graphics
- Microsoft Kinect V2 Sensor, with power hub USB cabling.
- Latest Microsoft Kinect V2 Drivers (Connect Kinect to your computer, and Go to Windows Update to Obtain[3]

1 The Program is only supported on a Windows Device, if you're considering cross platform development consider looking into the libfreenct2 library.

2 This will not work with a virtual machine, you must have a physical windows device, i.e. will not work with parallels, virtualbox, vmware, etc.

3 Downloading the Kinect V2 SDK will not automatically install the updated driver, you will need windows update for this

Library Requirements

- pandas
- numpy
- matplotlib
- pyinstaller
- requests
- opencv2-python
- Any other libraries required by the PyKinectV2 Library: https://github.com/Kinect/PyKinect2
    - **Note:** DO NOT USE A CONDA Environment, otherwise this will cause issues when building the program with pyinstaller

Recommended Skills

1. Understanding of Object Oriented Programming in Python3
2. Understanding of using PyQT5, specifically its multithreading along with its signal manipulation functions.
3. An understanding of backend vs frontend in web development, as this will help with understanding how the UI was created
4. Understanding of 3d arrays.
5. Understanding of Git

Obtaining The Kinect Data

The program is divided into multiple files and multiple derived classes. Before diving into this lets first begin with basics of obtaining data from the Kinect V2. The PykinectV2 library handles a lot of the low-level information conversion from the official Microsoft Kinect V2 SDK, however, it will only provide you the raw data as an np.ndarray, therefore we must code convert it to something that openCV can use so that we can display the feed from the Kinect V2.

Below is an example code block of how we instantiated the PykinectV2 library, obtain depth frames, then convert it to the a usable image for openCV2[4]:

```python
from Resources.pykinect2 import PyKinectV2
from Resources.pykinect2 import PyKinectRuntime

from Resources.CVResources import imageEditor as IMPROC

class Example:

    def setupKinect (self) :
        # Instatiate PyKinectV2 Library, then set it to get Depth Frames
        self._KinectDev = PyKinectRuntime.PyKinectRuntime (PyKinectV2. FrameSourceTypes_Depth)
        # Set the height and width for the open window 424x512 should be the resolution
        self._Height, self._Width = self._KinectDev.depth_frame_desc.Height, self._KinectDev.depth_frame_desc.width
        # Insantiate Our Implementation of The Class that will convert raw depth image data to open usable data
        self._OpenVDepthHandler = IMPROC.CVEDITOR_DEPTH(self. _Height, self. _Width, "Kinect V2 Gait Analyzer")


    def handleNewDepthFrames (self) -> np.ndarray:
        # Check with the PykinectV2 Library if there is a new depth frame
        if self._KinectDev. has_new_depth_frame():
            # Save the last depth image, and the 3d array that contains our deth data
            self.frame = self._KinectDev.get_last_depth_frame(),
            self.frameDatareader = self._KinectDev._depth_frame_data
            # Convert this raw depth image to an open usuable one and store it in two variables
            self.displayFrame, self.frame = self._OpenVDepthHandler.convtImg(self.frame)
```

After conversion of the depth frame, two images will be returned:

1. A Normal black and white image, which will be used for person/motion detection
2. An onion layered image for display in the openCV window

A lot of the functions that require any modifications of the image and obtaining positioning of a patient in the camera is taken care of in the Kinect-UI-Frame-web-2/Resources/CVResources/imageEditor.py file along with the kyphosisEditor.py file, so it would be beneficial to ensure you understand the functions in these files.

4 For this implementation a custom function that handles frame conversion was implemented in the Kinect-UI-Frame-web-2/Resources/CVResources/imageEditor.py and it is called in this example

Kyphosis Implementation

Class Overview

Kyphosis Calculations includes 5 classes :

1. class kyphosisControl : Used to control the Kyphosis UI (UI implementation will be discussed later)
2. class Kyphosis : The main class to handle the measurement of the Kyphosis Index (thoracic curvature)
3. class CamParams: Used to obtain camera settings i.e. focal point and principal. The settings from this are used to calculate distance of the key spinal points (C7, T12, and L1) to calculate kyphosis index.
4. Class MOUSE_PTS() : Stores the depth values of a given key spinal point, stores the x,y coordinates in the ‘real world’, and holds the average depth of these points over a given amount of frames.
5. class KyphosisEditor.py: Used to handle selection of the spinal points using the mouse, and displaying these selected points.

Implementation Summary

To implement, threads must be set up first in order for this to work with the UI. In the UI we first instantiate the Kyphosis class, and then move it to a thread within the setupThreads() function in the KyphosisControl UI class. The KyphosisControl UI class then handles binding the UI buttons to the proper function in the kyphosis class and the viewfinder controls key spinal point management and displaying a frame.

Obtaining Kyphosis Data

Obtaining depth frames and the data associated with distance is slightly different for the

```
1  from Resources.pykinect2 import PyKinectV2
2  from Resources.pykinect2 import PyKinectRuntime
3
4  from Resources.CVResources import imageEditor as IMPROC
5  from Resources.CVResources.kyphosisEditor import KyphosisImg
6
7
8  class Example:
9
10     def setupKinect(self):
11         # Instatiate PyKinectV2 Library, then set it to get Depth Frames
12         self._KinectDev = PyKinectRuntime.PyKinectRuntime(
13             PyKinectV2.FrameSourceTypes_Depth)
14         # Set the height and width for the open window 424x512 should be the resolution
15         self._Height, self._Width = self._KinectDev.depth_frame_desc.Height, self._KinectDev.depth_frame_desc.Width
16         # Insantiate Our Implementation of The Class that will convert raw depth image data to openCV usable data
17         self._OpenVDepthHandler = KyphosisImg(self._Height, self._Width,
18                                               self.WINDOWNAME)
19
20     def handleNewDepthFrames(self) -> np.ndarray:
21         # Check with the PykinectV2 Library if there is a new depth frame
22         if self._KinectDev.has_new_depth_frame():
23             # Save the last depth image, and the 3d array that contains our deth data
24             self.frame = self._KinectDev.get_last_depth_frame(),
25             self.frameDataReader = self._KinectDev._depth_frame_data
26             # Convert this raw depth image to an open usuable one and store it in two variables
27             self.displayFrame, self.frame = self._OpenCDepthHandler.convtImg(self.frame)
```

kyphosis index than presented in the Obtaining The Kinect Data section. Please view the example below:

Pay attention to line 17, here we see that instead of what we did previously, we are now instead making a call to the kyphosisEditor class and placing the height, width, and a windowName as parameters.

Ideally, the handleNewDepthFrames() function should be called within a while loop with a boolean flag that indicates whether the kyphosis index program is finished. This can be seen in the runtime() function within the kyphosis.py file.

Kyphosis Functions Documentation

This section will provide an in depth description of each function.

runtime()

This function is called from the UI when the user elects to run the Kyphosis Index measurement program. This function takes care of

1. Setting up the Kinect
2. Main measurement loop to consistently obtain depth frames and associated depth data
3. Obtaining the proper factor camera settings that account for its natural angle. 4
4. Handling of user mouse presses to select the key spinal points (C7,T12, L1)
5. Calling of helper functions to analyze and obtain the kyphosis index
6. Upon completion of an instance, the program averages the patient's kyphosis index and uploads this average to the Healage portal.

kyphosisSetup()

This function is called once the runtime() function has been called in the UI code. The purpose of this function is to set up the program to retrieve depth data along with depth frames from the PykinectV2 library and instantiate a window to display a Kinect frame to.

handleOpenCVEvents()

This function is called consistently through the program loop. The viewfinder has the ability to take user input of where the key spinal points should be, internally the coordinates of these selected points are handled by the kyphosisImg class (located in kyphosisEditor.py). Within this class the points are saved to a list of ptStrcut objects, which has variables of x,y and these points can be returned using the ptsStruct's getXY() function which is then appended to the spinalLandmarksArr.

gatherIntrinsics()

Simple obtains the intrinsic values of the camera; all intrinsics are saved in the CamParam class, and easily obtainable through the CamParam's getIntrinsics() function.

captureDepthData()

When the user presses 'Analyze' in the UI this function will run taking a total of 5 depth frames, and obtain the distance of each individual point over the course of these 5 frames. It will then set the average of each individual point in a FOR loop. This function is only called when all the following is TRUE:

1. The 3 key points of the spine are selected (C7, T12, L1)
2. The user has pressed the 'Done' button in the UI

There is one trick to be aware of though:

1. spinalLandmakrsArr is an array of MOUSE_PTS objects.
    a. The rationale behind this involves the understanding that points selected in the viewfinder require an equation in order to find their respective depth.
    b. The PyKinectV2 library and the Microsoft Kinect V2 SDK store data in a 3D matrix, so we must find the corresponding depth (distance from the kinect) of a point using the equation:

$$depthFromKinect = [(y * width) + x]$$

Note: This does not have to be a concern, since a function called getDepth(depth_frame_data_from_pykinectV2, x, y) was implemented which will simply return the depth of any given point. However, this is only used to obtain the depth of an x,y point in camera space, therefore we must find the coordinate in the 'real world'. More about this will be mentioned in the convt_XY_Pts_To_Distance() function.

convt_XY_Pts_To_Distance()

Recall that our self.spinalLandmarkArr is an array of MOUSE_PTS objects, so each object stores the x,y coordinate of each dot placed (should be on the C7, T12, S1) on the screen to measure kyphosis. However, these are coordinates in the viewfinder which hold a depth from the Kinect, but in order for us to calculate the distance between these points later on, we must convert them to world coordinates and distance. This conversion is taken care of in the _convtXYToDistance() function which serves as a helper function.
The equation for conversion as follows within the _convtXYToDistance:

$$distanceX = ((pointX - cameraPrincipalX) * depthValOfXYCoord)) / cameraFocalX$$

$$distanceY = ((pointY - cameraPrincipalY) * depthValOfXYCoord) / cameraFocalY$$

After calling the helper function the real world distance and coordinates are added to the respective MOUSE_PTS object in the array. getKyphosisIndex() :

This function will call a helper function which will get the distance between the C7 and T12. It will then attempt to find the value farthest from the camera using the getDeepest() function, which will return the x,y,z of the furthest point. The returned z is then used to calculate kyphosis index. The equation for kyphosis index calculation as follows:

$$kyphosis = (apexOfKyphosis\ (a.ka.\ returned\ Z) / lenC7toT12) * 100$$

doCalculations()

This function handles running the calculations in proper order.

1. First you must ensure that all 3 landmarks (C7, T12, L1) are properly selected in the viewfinder.
2. X,Y coordinates of each landmark must have a distance and a real world distance associated with them
3. Obtain the kyphosis Index
4. Append the data to an array that holds the kyphosis result, so that they can be averaged later if the user decides to run multiple tests on a single patient.
5. Send the calculated index to the UI so it can be displayed to the user.
6. Increment the Program Runtimes of the current instance.
7. Update flags to allow the program to run again on the same patient.

handleImgDisplay()

This function will obtain the X,Y coordinates selected on the screen, and display a visible dot on the selected points in the viewfinder. It will also handle prompting the user of what to do on the UI.

GAIT Implementation

Class Overview

Gait Calculations includes several classes:

1. Class controlPanelGait() : Used to control the Gait UI.
2. Class GaitAnalyzer() : The class that is used to handle collection of data for instant velocity and frame by frame analysis of Gait speed.
3. Class GAIT() : The parent class of GaitAnalyzer(), this contains all necessary functions other than those of collecting and computing instant velocity and frame by frame data. More about this will be discussed later.
4. Class Graph() : This class takes care of plotting instant velocity and frame by frame. It primarily plots Velocity vs. Distance, which are shown in two different lines; one line represents Velocity vs. Distance of frame by frame analysis and another line represents the Velocity vs. Distance of instant velocity analysis.
5. Class Timer() : Used to control the timer of the program, so that we can calculate instant velocities and frame by frame velocities using the time tracked from this class.

Implementation Summary

To implement threads must be set up first for this to work with the UI. In the UI we first instantiate the GaitAnalyzer() class, and then move it to a thread within the setupThreads() function of the controlPanelGait() class. The controlGait UI class then handles binding the UI buttons to the proper functions in the GaitAnalyzer and GAIT class. Be aware that GaitAnalyzer is a derived class of the GAIT class, therefore there was no need to instantiate the GAIT class itself. Also, do take note of the multiple qt signals that are used, since the UI is a bit more complex than the Kyphosis Program and requires a lot of signals, the signals can easily be seen by viewing all objects with a .emit(). Also take note that we are using a custom made timer class, so do take a look through the timer.py file. The basic algorithm for collecting gait data is as follows:

1. Obtain an initial image of the surroundings (to allow for person/motion detection)
2. Obtain the frame difference and place a box around the difference along with a dot that points to the center of the box.
3. Track this point, program will set flags accordingly; flags are set when:
    a. The patient reaches the beginning of the measurement zone
    b. The patient reaches the end of the measurement zone
    c. When calculations are finished and ready to display
4. Save velocity, time, distance data to a dictionary for their respective measurement type, in other words data collected for instant velocity measurement is stored in the dictionary for instant velocity measurement and similarly frame by frame measurement.
5. Place the data from these dictionaries into two pandas dataframes, one pandas dataframe is for the instant velocity measurements, and the other for the frame by frame measurements.
6. Clear the dictionaries, and for each instance the data is kept in the pandas dataframe, until the provider selects ‘Switch Patient’ or ‘Logout’
7. Display the current calculated average velocity of the patient from the most recent test. Along with displaying options to show the Velocity vs. Distance graph of the most recent test.
8. Once the provider presses save and quit on an instance, then the program will display the averages of all the calculated average velocities, average all the data from each distance of each measurement type (i.e. all velocities similar distance ranges in the instant velocity measurement are averaged together), then send this data to the healage site, along with providing options to see the graphs of all tests of the current instance, or show the averaged data graph.
9. If the provider chooses to test the same patient again, in other words the ‘continue’ button is pressed then reset program flags and empty the storage dictionaries mentioned in step 5, append the most recent test average velocity to an array, and clear the graph.
10. If the provider selects ‘Switch Patient’ or ‘Logout’ reset all variables and storage dictionaries and dataframes.

GAIT Functions Documentation

Class GaitAnalyzer Functions

runtime()

This function is called from the UI when the user selects to run the gait speed program. This function takes care of:

1. Setting up the kinect
2. Main loop that controls collecting frame data, instant velocities, and the average of an iteration of an instance.
3. Obtaining the the initial image needed for motion detection and object tracking
4. Upon completion of an instance all instant velocities of each 1 foot distance are averaged to create a final graph, and all the average velocities across the instance are all averaged together.

handleNoInitFrame()

At the beginning of an instance the user is prompted to either select a previously captured initial image or to capture a new one. It is recommended that the user capture a new initialization image at the beginning of an instance for the best detection. Within this function the program checks to see if an image was either selected or captured, if not the user is prompted in the UI to obtain an image. This does not do anything but simply ensure that the UI reports the appropriate status here and disables the appropriate button.

_convt_init_img()

This function handles the conversion of the initialization image to an openCV usable image. This will set the initial image to a simple image to be compared against by consecutive frames to capture the distance of the patient. This also sets the program flag of self._InitFreameConvtd to true once the image has been converted, allowing the rest of the program to run.

frameByFrame()

This function analyzes distance changes between a certain amount of frames, in this case it is set to n=5, this can be modified by adjusting the class variable GaiyAnalyzer::_FramesToRead. It is important that this global variable is adjust if you would like to change the frames to count before calculating a distance difference, since the equation necessary to calculate speed between every n frames requires this to calculate distance traveled.

Within this function we first ensure that all appropriate flags are correct, to prevent any errors. Afterwards we then obtain the current distance of the patient and the current time difference which is the current time minus the time the patient reached the beginning of the measurement zone. Then the previous distance obtained from the previous frame is placed through a function that ensures that the program isn't collecting large negative differences which could be the result of noise. Do note that the function that checks for negative distance travel is called find_min_from_max_frame(), this function will be discussed in further detail later.

Once all the data necessary for speed calculation is complete, the current speed is calculated using a basic $speed = distance/time$ equation. The equation is as follows:

$$currSpd = ((currDistance - prevDistanceFrameBFrame) / unitConversionConst)$$
$$currSpd = currSpd / FrameConst$$

FramesConst is simply the frame to read (in this case 5) divided by the frameRate of the kinect (in this case 30).

Once the current speed is calculated the data is appended to a temporary storage dictionary. Then the current distance recorded gets saved in the variable that holds the previous distance . The structure is as follows:

```
frameStore = {
        'Iteration_ID' : [],
        'frame_ID' : [],
        'curr_distance': [],
        'velocity' : [],
        'time' : []
}
```

find_min_from_max_Frame()

This function will obtain the midpoint of the detected motion, along with the width and height of a contour which was detected by opencv from the function in the parent class, this will be explained more later. With this information the function will evaluate every distance within the detected contour, and look for a depth that will not result in a negative value, since it is assumed the patient does not walk in the negative distance. This will then return the found distance value.

calculateInstantVelocity()

This function will first obtain the instant velocity the patient was at when they first crossed into the measurement zone. After a velocity was obtained, the program is allowed to continue calculating the instant velocities of each 1 foot distance.

This function will record all distances that are less than the current threshold in a dictionary, i.e. we are looking for the instant velocity of a patient at 1 meter, therefore we will collect all distances and times that are less than 1 meter, once we find a distance greater than the distance we are looking for then we increment the distance in this case to 2 meters, so we are now looking for all distances and times that are greater than 1 meter but less than 2 meters.

Going back, now that we have found all distances and times that are less than the current distance we are looking for we sort the list in ascending order, in which we choose the 0 index, this choice was arbitrary and could be improved upon however. We then return this distance along with the time associated with this distance. After these values are returned and appended to a temporary storage dictionary called iv_store.

The iv_store dictionary is structured as such:

```
iv_store = {
        ‘iteration_ID’  : [],
        ‘distance_ID’ : [],
        ‘curr_distance’ : [],
        ‘velocity’ : [],
        ‘time’ : []
}
```

finishUp()

This handles the finishing of an instance. Within this function the temporary storage dictionary of frame_store and iv_store will be placed into the function GAIT::copyToDataFrame(), which will be converted to a dataframe that will hold the information of all the data from an instance. The parent class will eventually use this data to create a final graph that displays two lines, one for the results of the frame by frame algorithm and the other for the instant velocity algorithm. Along with this the average of all the average velocities will also be displayed on the graph and as a pop up.

Class GAIT Functions

programFinished():

This function simply calls the appropriate class functions to handle the end of the instance.

This function takes the average of all calculated averages of a single instance, then sets up the graphs to show this average. Once complete this function will send the data to the Healage Site for viewing along with closing the connection to the Kinect device.

saveToDataBase():

This function will take all the average of all the gait speed data points and bind them to the patient name and id, then structure it appropriately for the healage site to parse through it. This function calls a function that was created in the uploadData.py file, which structures the data appropriately.

getCurrGaitSpd():

This function will be called from the ui, so that the ui shows the appropriate gait speed of an iteration in the current instance.

reportGait():

This function checks that the program flags of Endreached and calculations allowed are set appropriately, then handles the insertion of gait data points into the graph, and then calls a function that calculates the average gait speed of the current iteration.

doGaitSpeedCalc():

Simply calculates the gait speed using a basic $speed = distance / time$ then appends that data to an array that holds average gait speeds for each iteration of an instance. Then resets the appropriate program flags.

handleGeneralDistance():

This function obtains the area in which motion is detected using frame difference calculations which are abstracted by a simple call to the getObjectMidPoint() function from the imageEditor.py file. Here we simply obtain the area that the person may be in and identify the midpoint of this, then use that midpoint's x,y location on the screen to identify its distance from the Kinect. Then this distance is constantly checked to ensure that we have not passed the end of the measurement zone. Once we have passed the end zone, this function triggers the program to perform gait speed calculations. However, since it is possible for the midpoint to be impacted by noise, we will find the point with the closest distance to the Kinect, to ensure a better estimate on gait speed.

_fullReset():
This function will completely reset all variables and data storage data structures. This is used when the user decides to either switch patients or logout of the program.

_reinit():
This function resets flags that trigger the gait speed calculation and any timers. This is primarily used if we want to run another measurement on the same patient, in other words we want to run another iteration of an instance.

removeNoise():
Since the calculations of data points may have some bad values we simply apply some box plot methods to reduce bad data, such as finding the interquartile range, upper and lower fences, and then removing data that falls outside of the upper and lower fence.

copyToDataFrame(iv_store:dict, frame_store:dict):
This function will take a temporary instant velocity and frame data dictionary and append it to the dataframe of the current patient. In the Class of GaitAnalyzer (a child class) we are temporarily storing the data points of the current iteration, the storage that holds these data points in the child class are then emptied once an iteration is complete for simplicity and space purposes. However, we do want to keep the iteration data, so that we can perform a final averaging of all collected data so that we may upload it to the Healage site. This provides a simpler approach as the pandas library makes dataframe manipulation easy and operations become quicker. To see the structure of the temporary storage dictionaries view the parent class' documentation above.

__setupAvgGaphHelper():
This function structures our pandas dataframe properly and calculates averages of each data point so that they can be graphed by matplotlib in the graph.py file. This will return the data frame that has any n/a values emptied along with any outliers removed.

UI Classes

Summary:

The UI python files that handle the UI are broken into 3 files:

1. KinectAnalyzer.py
2. gaitSpeedUI.py
3. kyphosisUI.py

In this section I will be using the terms of frontend and backend code, since this seems like the easiest way to think of the program overall. The frontend code will consist of anything that involves the UI, such as actions to perform when a button is pressed. Backend code will refer to classes that handle the actual measurements; **the backend classes are class Kyphosis, class GaitAnalyer, and class GAIT.**

These files all work together to handle all UI windows within the program. Each file however, calls upon other files located in the UIResources folder. The UIResources folder contains the python code that creates the design and UI elements of each window, it is important to note that each window has its own python file, along with its own variable names for each element. These files were not coded from scratch, rather a program called PyQt Designer was used to generate these files.

PyQT Designer is a program that allows us to create windows in a visual way, without having to guess spacing and the keywords for different window elements, in other words, it abstracts away all the complicated stuff. PyQT5 generates files called .ui files which can be converted to python files by using the command **pyuic5 fileName.ui -o outputFileName.py -x**

As far as it goes you should not have to mess with these python files, as any edits to them will be deleted once you recreate the window using the above command. Within the UIResources folder there is a subfolder called uiFormat, this folder contains the .ui version of all the .py windows. To change the layout or just any part of the window, simply open pyqt5 designer and then open the .ui file that belongs to a window, and edit from there.

The UI windows are overall simple in logic, since it primarily involves making signals that will cause a change in the current ui, an example of this is after the user presses login and the window for two factor authentication is triggered. So, this section will only go over the classes and functions that I felt would be a little tricky to grasp as opposed to providing explanations per class function.

Some important functions to keep in mind are the .show() method for each window which will simply show the window, along with the setupUI() function which is used to instantiate a window.

It is also important to note that pyqt5 has a wide variety of window types, here we primarily use the QDialog Window type.

Instantiating the Backend Code

In order for the UI to remain responsive while the backend code runs, we must perform some multithreading, which is easily taken care of by the PyQT5 library (the UI library). View the code below to see the demonstration:

```
98      def setupThreads(self):
99          self._GaitProgramThread = QThread()
100         self.gaitProgram.moveToThread(self._GaitProgramThread)
101         self._GaitProgramThread.start()
102
103
```

Looking at the code above the following steps occur:

1. Create an instance of the QThread library
2. Move the instance of the GaitAnalyzer class to a thread
    a. You should take a look at the Parent class (class GAIT) of GaitAnalyzer, within the parent class you can see that we are making the parent class of GaitAnalyzer inherit the properties of QThread, which makes the code above possible.
3. Start the Thread

.setupUI():

There are two ways to instantiate a window, the first is to use a built-in function to create the window based on a .ui file or creating a window based on the .py source file. In this program it is preferred to use the .py source file method, since .ui files will not be included when the program is built using pyinstaller (there is a way, but this leads to the issue of the folder holding the .ui files being deleted by accident by the user).

In order to instantiate a window, the source file and the class in the source file must be imported, an example of this can easily be seen in any of the python files listed in the introduction to this section.

After importing the file, we must first create a class that will hold the root window, for example the program selection window or the gait control panel window. An example of doing this can be seen below:

```
10  # UI Imports
11  from Resources.UIResources.UI import controlCenterGait, gaitSpdGeneral, gaitSpdAverageUI, trackbar, errorLog
12  # Gait Runtime Program
13  from Runtime.programRuntimes.gaitRuntime import GaitAnalyzer
14
15
16  # Directory Path
17  _ProgramPath = os.path.dirname(os.getcwd())
18
19  class controlPanelGait(QDialog):
20
21      trigger_Program_restart = pyqtSignal(bool)
22      trigger_patient_switch = pyqtSignal(bool)
23      def __init__(self):
24          super(controlPanelGait, self).__init__()
25          self._Window = controlCenterGait.Ui_Dialog()
26          self._Window.setupUi(self)
27          #self._Window = loadUi(os.path.join(_UIPath, "controlCenter.ui"), self)
28
```

Let's go step by step and see what's happening.

1. Looking at line 11, here we are importing the python file controlCenterGait.py from our uiResources folder, which will be the window that shows user interactable buttons for the Gait Measurement Program.
2. Looking at line 19 we are creating a class called controlPanelGait, which will be a child class of QDialog, since the gait control panel window is a QDialog window, this allows us to use functions that are needed in this class from the parent class.
3. Looking at line 25, we are creating a class variable called _Window, which will be an instance of the controlCenterGait's UI_Dialog() class.
    a. A UI_Dialog() is a class that is generated from the .ui files when we convert them into python source files, as mentioned in the introduction to this section.
4. After instantiating the window from the python source file, we must call the setupUI() function so that we can have access to all the elements of the window, and then perform the appropriate functions.

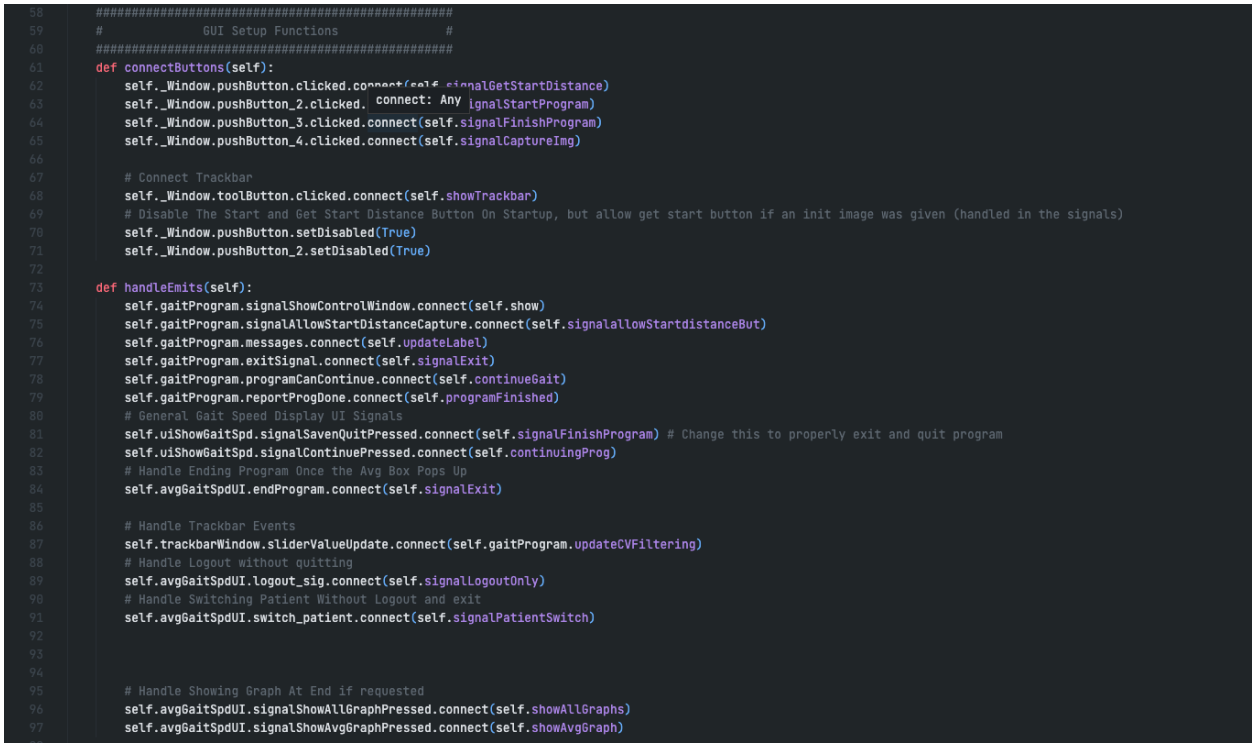
Class ControlPanelGait:

All ui files will follow the same logic overall, just with functionality adjusted based on the intended purpose of each window, so I will provide the general logic using the ControlPanelGait class.

ControlPanelGait as mentioned above, handles the user interactable buttons during a gait speed analysis. Once the gait speed analysis is done, a window that shows the gait speed of an iteration of an instance will be created and shown, in this scenario the control panel for the gait measurement is the parent window and the window that displays speed is the child window.

In order for the window to work, we must first instantiate it and then display it as mentioned in detail above. After, we must link each button and create signals in the child windows that will tell this parent window what to do.

Let's first discuss linking the buttons. After instantiating the window, we now have access to the buttons which are simply class variables. Each button has an action method attached to them, one method that will be used often is ._clicked.connect(). This function allows us to perform a function when the button is pressed, for example, if I were to press the ‘analyze’ button it would call the function that starts the analysis of a patient’s gait. Examples of doing this are located in the image below in the connectButtons() function.



```
    ################################################
    #              GUI Setup Functions             #
    ################################################
    def connectButtons(self):
        self._Window.pushButton.clicked.connect(self.signalGetStartDistance)
        self._Window.pushButton_2.clicked.connect: Any ignalStartProgram)
        self._Window.pushButton_3.clicked.connect(self.signalFinishProgram)
        self._Window.pushButton_4.clicked.connect(self.signalCaptureImg)

        # Connect Trackbar
        self._Window.toolButton.clicked.connect(self.showTrackbar)
        # Disable The Start and Get Start Distance Button On Startup, but allow get start button if an init image was given (handled in the signals)
        self._Window.pushButton.setDisabled(True)
        self._Window.pushButton_2.setDisabled(True)

    def handleEmits(self):
        self.gaitProgram.signalShowControlWindow.connect(self.show)
        self.gaitProgram.signalAllowStartDistanceCapture.connect(self.signalallowStartdistanceBut)
        self.gaitProgram.messages.connect(self.updateLabel)
        self.gaitProgram.exitSignal.connect(self.signalExit)
        self.gaitProgram.programCanContinue.connect(self.continueGait)
        self.gaitProgram.reportProgDone.connect(self.programFinished)
        # General Gait Speed Display UI Signals
        self.uiShowGaitSpd.signalSavenQuitPressed.connect(self.signalFinishProgram) # Change this to properly exit and quit program
        self.uiShowGaitSpd.signalContinuePressed.connect(self.continuingProg)
        # Handle Ending Program Once the Avg Box Pops Up
        self.avgGaitSpdUI.endProgram.connect(self.signalExit)

        # Handle Trackbar Events
        self.trackbarWindow.sliderValueUpdate.connect(self.gaitProgram.updateCVFiltering)
        # Handle Logout without quitting
        self.avgGaitSpdUI.logout_sig.connect(self.signalLogoutOnly)
        # Handle Switching Patient Without Logout and exit
        self.avgGaitSpdUI.switch_patient.connect(self.signalPatientSwitch)



        # Handle Showing Graph At End if requested
        self.avgGaitSpdUI.signalShowAllGraphPressed.connect(self.showAllGraphs)
        self.avgGaitSpdUI.signalShowAvgGraphPressed.connect(self.showAvgGraph)
```

HandleEmits():

Looking at the image above, you can see a function called handleEmits(). Since we are dealing with a few windows, along with multithreading of the gait analysis program, there must be signals that inform the parent window so that it can react properly. Lets look at an example, look at line 76. Here we are updating the status message that is presented to the healthcare provider; this status message simply directs the provider of which button to press, depending on the current state of the gait analysis. For example, once the gait program has been calibrated it will enable the ‘analyze’ button and directs the provider to press the ‘analyze’ button, and once the analyzation has started automatically disable the ‘analyze’ button, to prevent confusion. The actual variable that holds this signal is called ‘messages’ (self.messages), this is of the type pyqtSignal. This signal is declared in the gait.py’s GAIT() class, which is accessible from this UI since we had instantiated the GAIT class or rather the GAIT’s child class GaitAnalyzer() in the constructor of this ControlPanelGait() class. Within the GaitAnalyzer() this signal emits status messages to the UI, which then triggers the UI to update the message that is shown to the medical provider on the UI. This same logic holds for the other signals, even the ones that are sent from child windows.

Last Notes:

Overall the other windows controlled by the other 2 python files function with the same logic, to best understand how the UI works it is better to look through the code of each, and follow the functions that are called. Also, there should be no need to call any other variables besides the signals from backend classes (class Kyphosis, class GaitAnalyer, class GAIT), since the UI should only be calling functions of these classes, in other words, if something needs to be changed in the backend classes you should create getter functions or only calling functions (i.e. setupAvgGraph()) that were made in these backend classes, that can be called from the UI.

Building the Program

To build the program we will be using a program called pyinstaller. Pyinstaller is an easy way to create an .exe file so that it can easily be run by the medical providers. The command to build is:

pyinstaller --noconsole --onedir --specpath ~/Desktop/Kinect --name "Healage Kinect" --icon=app.ico main.py

For further details please view the build.txt information file, as it provides some small tips and pointers.

References

[1] GitHub - Kinect/PyKinect2: Wrapper to expose Kinect for Windows v2 API in Python
[2] Kinect for Windows SDK | Microsoft Learn