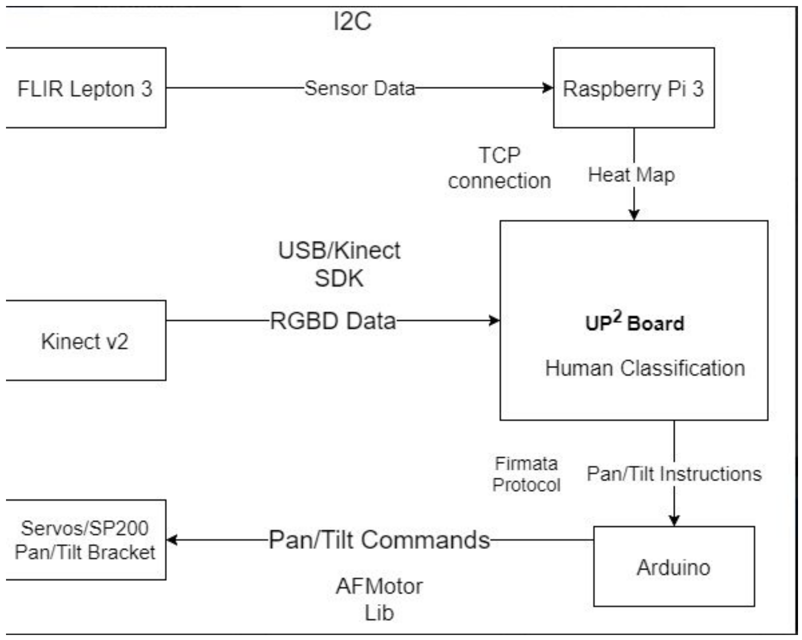
Distributed Climate Control - Final Design Document

By Dan Coen and Conor Sheridan

# Purpose and Goal

**Motivation**: Welcome to the Distributed Climate Control (DCC) project. This system will help businesses, homeowners, and other institutions save money on energy spending by dramatically reducing the amount of heating and cooling spent on air space that no one is occupying, as well as contributing to global efforts to create affordable, sustainable energy technologies. This project specifically aims to provide information specifying how many people are in a room, and where they are situated. This information will be used by a HVAC system which can target specific regions of a room. This project is inspired by the ARPA-E's research and funding of advanced energy systems.

**System Concept:** Our implementation involves using a Kinect RGB-D sensor to capture rigidbodies in the room as well as high resolution images. This data will be combined with a heatmap captured by the flir lepton 3 thermal camera in order to train a machine learning model which can be calibrated to work with variable sized rooms and large groups of people. Both camera modules will be mounted on top of a single automated pan and tilt bracket driven by servo motors, allowing the two sensor modules to synchronize their data easily and accurately. Our system is designed to be affordable, and able to run on modest hardware so that it is accessible to a large market.



## Technologies Used

**Flir Lepton 3:** Flir's highest resolution Lepton module. It is a long-range infrared camera. The sensor will be attached to and powered by a raspberry pi, using I2C commands.

**Raspberry Pi 3:** The third generation of the popular Raspberry Pi single-board computer. As stated above, the Flir Lepton camera will be sending data to this board using I2C commands. There will also be some opencv processing done on the board.

**Microsoft Kinect V2:** This is Microsoft's Xbox console depth and color camera which is used for VR video games. The sensor is one of the most affordable depth and high quality color cameras available. It is commonly used in research projects similar to the DCC as well as robotics. The Kinect provides us a color stream in an array of pixel RGB values. Depth values similarly are given in a point cloud of distances.

**UP Board:** Similar to the Raspberry Pi, this is a single-board computer. We are using this brand specifically because it has more processing power and also uses USB 3.0, which is necessary for connecting to the Kinect. Also, it can run Windows, which is important for using the Kinect for Windows SDK. This board is also where the majority of the processing will take place. It will receive frames from the Kinect via USB 3.0, and frames from the RasPi via TCP connection. Along with this, it will also send commands to move the pan/tilt bracket for the cameras.

**SP200 Pan/Tilt:** This is a standard pan and tilt bracket we are using to mount our cameras onto, and get a full view of the room. It uses 2 standard servos, and is controlled by an Arduino.

**Arduino Uno:** The third board we are using, this time to control our servos. The Arduino gets commands from the UP board via the PySerial library. The Arduino then commands the servos using the standard servo Library.

# Interface Specifications

**OpenCV Dependency - Dan and Conor**: Many modules of this project depend on OpenCV. An install of version 3.0 or higher will be sufficient for running our code. To install OpenCV, visit the following link: https://docs.opencv.org/master/d9/df8/tutorial_root.html. Please note that the Pylepton library also makes use of the python wrapper for opencv, which should be installed as well.

## Flir Lepton - Dan

### Python

**Class Socket:** Contains a simple fixed length server for the raspberry pi to send thermal images to the client. Uses Python INET sockets.

**server_init(self, sock=None)** - Create new socket, unless given existing socket

**raspi_connect(self, host, port)** - Connect to given socket with this self socket

**raspi_send(self, msg)** - Send image data to client. Continues to send data until length of msg is sent. Raises runtime error if connection is broken during sending of message.

**raspi_receive(self)** - Receive message from client. This will be used to get frame requests or desired I2C commands from the client. Receive messages with frame size of 2048. Returns bytestring of image. Raises runtime error if attempt recv returns length 0 string.

**Pylepton:** Groupgets Impementation of simple lepton library.

**capture(flip_v, device)** - Takes a picture using pylepton library, returning an normalized 8 bit image in unsigned integer format. Set Flip_v = True to flip the image horizontally, and device = “/dev/spidev0.0” if using spi port 0 (adjust accordingly). Should be checked for acceptable data.

## Kinect - Conor

**C++**

These are the main classes involved that we will be using for depth, motion, and body tracking, in that order:

Depthframe Class - Represents a frame where each pixel represents the distance (in millimeters) of the closest object seen by that pixel.
CopyFrameDataToArray - Copies the depth frame data into the array provided.

Bodyframe Class - Represents a frame that contains all the computed real-time tracking information about people that are in view of the sensor.
GetAndRefreshBodyData - Gets refreshed body data.

BodyIndexFrame Class - Represents a frame that indicates which depth or infrared pixels belong to tracked people and which do not.
CopyFrameDataToArray - Copies the body index frame data into the array provided.

On all of these classes, the close() method will also be used, which simply releases system resources associated with the frame.

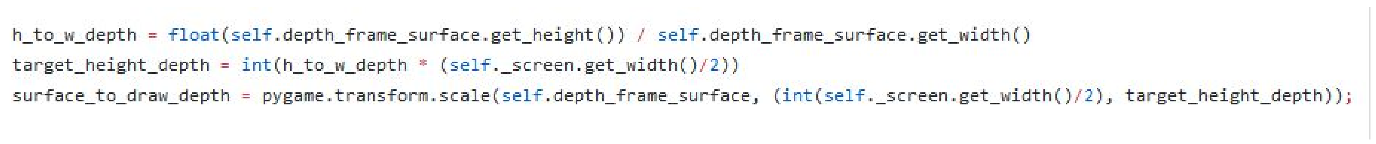
To initialize each reader stream and the sensor itself, you would do it as such, with the body reader as an example:

```
KinectSensor^ sensor = KinectSensor::GetDefault();
 sensor->Open();
 bodyReader = sensor->BodyFrameSource->OpenReader();
```

**PyKinect2**

PyKinect2 allows us to write Kinect applications in Python, which we are using to have a more unified and simplified project. The requirements are pretty straightforward as listed with the link above, and requires the Kinect for Windows SDK that we are already using. You can find more information about the PyKinect2 library here. It explains how to install it there, but the basics are that you need the 32-bit version of Anaconda, comtypes, and the Kinect for Windows SDK v2. It is a fairly simple setup.

Using PyKinect2 allows us to obtain Depth streams as such:



```
h_to_w_depth = float(self.depth_frame_surface.get_height()) / self.depth_frame_surface.get_width()
target_height_depth = int(h_to_w_depth * (self._screen.get_width()/2))
surface_to_draw_depth = pygame.transform.scale(self.depth_frame_surface, (int(self._screen.get_width()/2), target_height_depth));
```

Obviously there is more to displaying it and creating the buffer for it, but this is generally how it is transformed in PyKinect2.

We are running the Kinect for Windows SDK v2 on our UP board, which is running Windows 8.1. The Kinect for Windows SDK v2 can be found here, and is a very straightforward download procedure. We obtained Windows 8.1 through GW's Microsoft Imagine account. We are getting all our Kinect data directly to this board and processing it with opencv, as described at the top. The Lepton data is also being sent to this machine as well.

## Arduino - Dan

### C++ - Serial Communication (Windows)

**Serial Class:** An object used to interface with the arduino by connecting via the coms port on windows.

**ardSetPosition(int position)** - Function to set the position of the pan and tilt bracket. The position corresponds to the portion of the room to be viewed. Returns 1 on error.

```
Ex. |0|1|2|3|
    |4|5|6|7|
```

### C++ - Serial Communication (Arduino)

**setPosition(int position)** - Callback function for serial read when checking for command from windows machine. Sets pan servo and rotate servo appropriately for position given. Returns 1 on error.

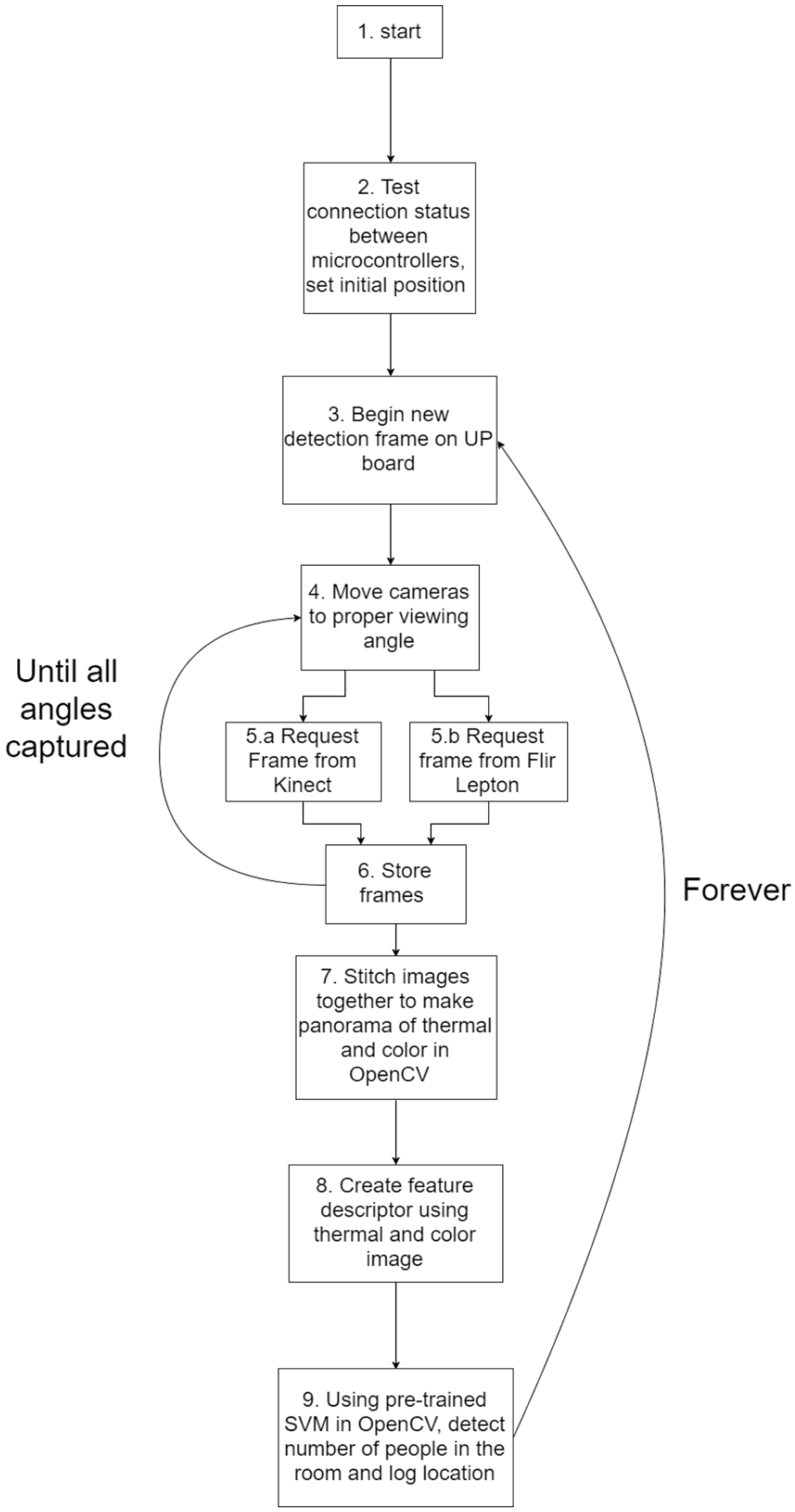
## Timeline

**Combined Image Streaming - Dan and Conor**: We want to be able to request and obtain frames from both the Kinect and the Lepton at the same time to the UP Board. This is in progress, and we expect to be completed by the end of December.

**Training SVM - Dan:** Over break, Dan is going to work on using sample Lepton and Kinect images to train and SVM primarily using a HOG feature descriptor. Details on HOG can be found here. We are planning on having this completed by the end of January.

**Image Stitching - Conor:** Conor will be working on combining separate frames over break. Since we will be capturing multiple images at different angles using the pan/tilt bracket, we need to be able to combine these images to get a full view of the room we are capturing. For this we will be using a panorama stitching algorithm using opencv. More information is here. This is also planned to be done by the end of January.

**Control Flow - Dan and Conor:** Once the parts over break are completed, we will start finalizing our full control flow such that the DCC system is fully functional and usable. The control flow is designed on the following page.

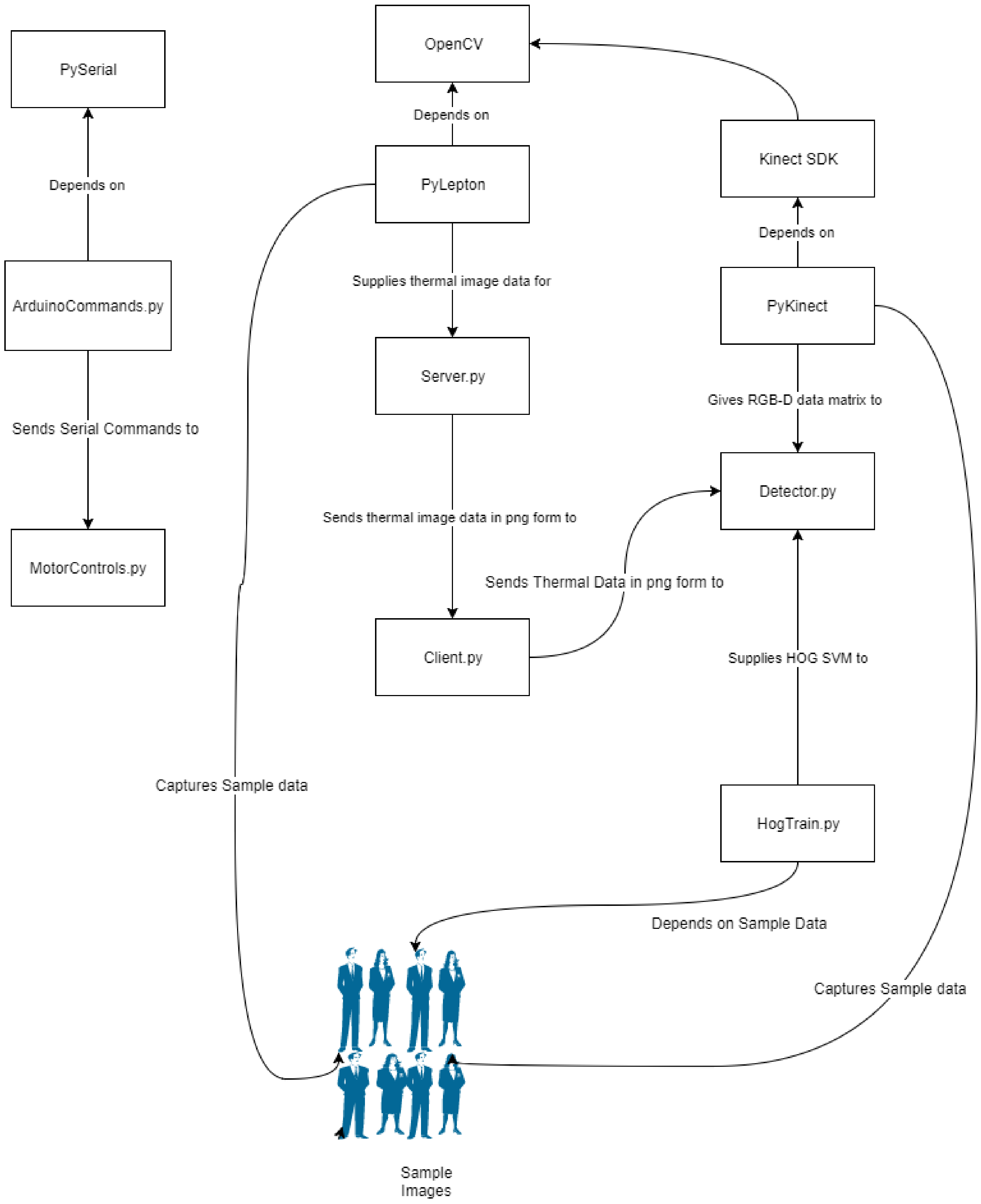

1. start
2. Test connection status between microcontrollers, set initial position
3. Begin new detection frame on UP board
4. Move cameras to proper viewing angle
5.a Request Frame from Kinect
5.b Request frame from Flir Lepton
6. Store frames
7. Stitch images together to make panorama of thermal and color in OpenCV
8. Create feature descriptor using thermal and color image
9. Using pre-trained SVM in OpenCV, detect number of people in the room and log location
Until all angles captured
Forever

**Control Flow Details:**

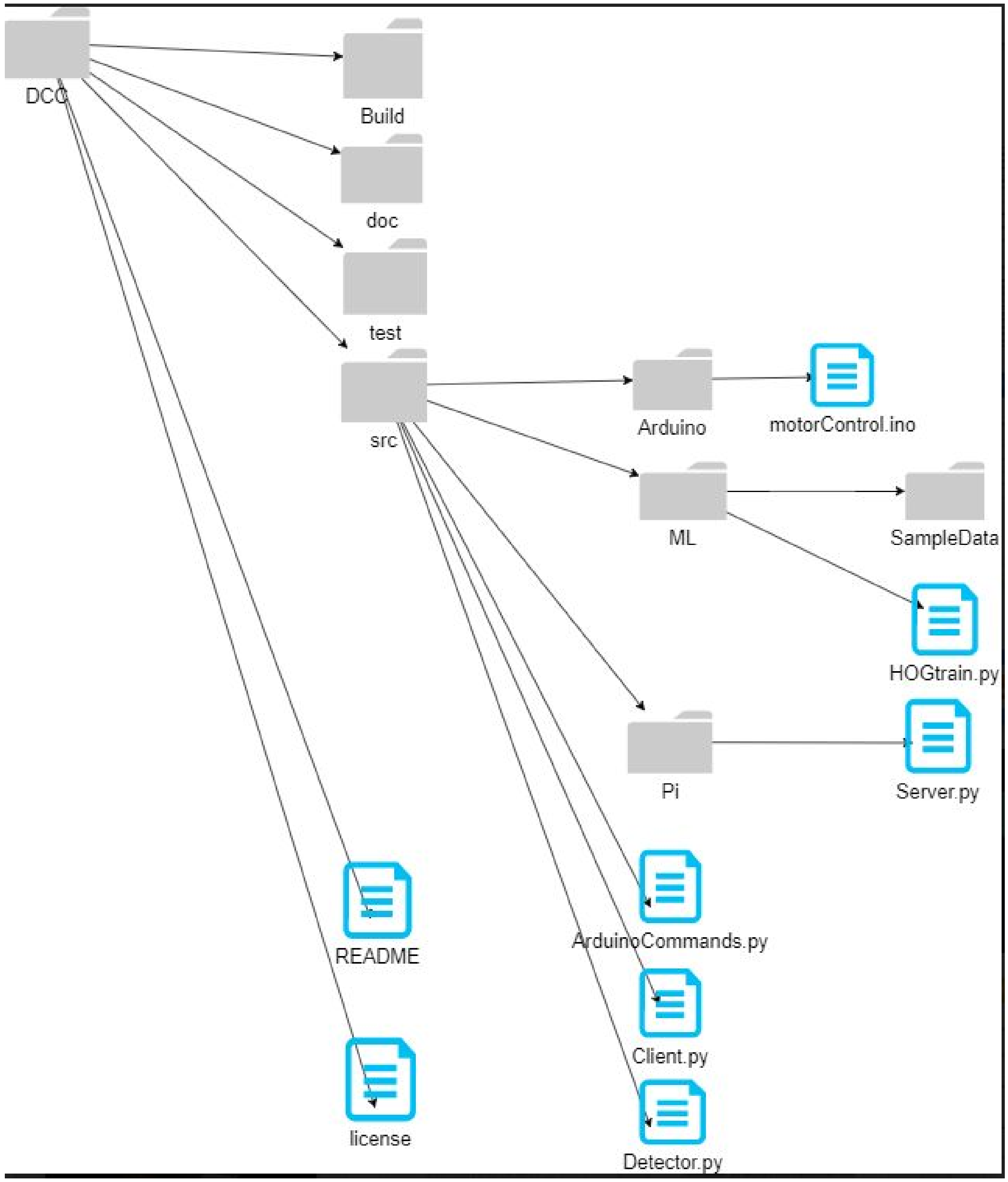
Note: Our initial implementation does not plan to include depth data.
**Step 1.** Main program on UP board begins execution.
**2.** UP board checks connection with arduino and raspberry pi. During this process, handshaking occurs between the arduino's (motorControl.ino) and UP board's (ArduinoCommands.py, pySerial Library) Serial communication as well as the image server on the raspberry pi (Server.py) and the UP board client (Client.py), and the raspberry pi verifies that the sensor is functioning properly (Pylepton Library). The kinect will also be tested to make sure images are appearing properly (pyKinect2 library). Then, the arduino will be commanded to move to the initial position for creating a panorama image.
**3.** The UP board creates the matrix buffers for the new images to be captured and stitched together using OpenCV.
**4.** The Arduino is commanded to move the next location to capture an image for the panorama using serial communication.
**5a.** RGB-D images are requested for the current frame from the kinect. Depth is returned as a depth data matrix which is converted to a grey-scale image.
**5b.** Thermal image is sent over the network as a png from the raspberry pi.
**6.** The frames are stored in non-persistent matrix buffers until all frames are collected.
**7.** OpenCV image stitching done separately on the thermal, color, and data images.
**8.** Use both images to create a feature descriptor which depends on the thermal, color, and depth attributes. This composites the three different images in a sense.
**9.** The real-time stitched image is scanned for objects using the pre-trained SVM in the HOG algorithm in OpenCV. The result is returned, logged, and displayed.

This will hopefully completed in March.
Throughout all of this, we will be unit testing and making sure that our system is working as expected. In the Spring, we are both going to be working on testing and revising our system and making sure our SVM is giving us the results that we expect.

**Dependencies**


PySerial
Depends on
ArduinoCommands.py
Sends Serial Commands to
MotorControls.py
OpenCV
Depends on
PyLepton
Supplies thermal image data for
Server.py
Sends thermal image data in png form to
Client.py
Sends Thermal Data in png form to
Kinect SDK
Depends on
PyKinect
Gives RGB-D data matrix to
Detector.py
Supplies HOG SVM to
HogTrain.py
Depends on Sample Data
Captures Sample data
Captures Sample data
Sample Images

**Directory Structure**



**Basic Description of Directory Structure:**

build - Compiled files

doc - Getting started, info about the project, ourselves, and other information

src - The source code. This contains subdirectories which are named after the system component that is run on other machines, or in the case of ML, the code required to train a machine learning algorithm prior to real-time use of the device.

test - Some automated tests to insure sensors and network are working, and all requirements and dependencies met

README - Information about other files in directory, install instructions, etc.

license - If this becomes and open-source project, we will pick an appropriate license