# **Age Estimation**

# 

**Table of Contents**

[**Age Estimation** 1](#_Toc56458262)

[**Introduction** 3](#_Toc56458263)

[**Environment & Tools** 4](#_Toc56458264)

[**Overview** 5](#_Toc56458265)

[**Development Process** 6](#_Toc56458266)

### 

## **Introduction**

## **Environment & Tools**

**PyCharm version 11.0**

PyCharm is an [integrated development environment](https://en.wikipedia.org/wiki/Integrated_development_environment) (IDE), specifically for the [Python](https://en.wikipedia.org/wiki/Python_(programming_language)). PyCharm has a free version (Community version) which we use in this project, it is based on intelliJ IDEA IDE.

**OpenCV 2.4.12**

OpenCV is an open-source library of programming, its main focus is on computer vision field. While written mainly in C++ OpenCV offers a python interface which we used.

**Cubemos – skeleton tracking SDK**

Cubemos is a pose estimation algorithm that is offered by Intel, the algorithm extracts 18 joints with 3D coordinates and works in real time. The license is not free to use, though it has a free trial of use. Our main use of Cubemos algorithm was comparing the depth result of this algorithm with the depth we generate by projecting 2D pixel.

**OpenPose**

OpenPose is a free to use pose estimation algorithm. Introduced first in late 2017, OpenPose offers a multi-person system to jointly detect human joints key points in real time. Currently with very high popularity among pose estimation algorithms, due to large features it offers such as - number of key points )135), both CPU and GPU versions, an easy to use demo, etc. our use of OpenPose was mainly to test the CPU version.

**AlphaPose**

AlphaPose is a free to use pose estimation algorithm. Introduced first in 2018, AlphaPose provides a multi-person pose estimator with high accuracy. In addition there is variety of models available to use, ranging from 18 to 126 skeleton key points, both CPU and GPU versions. While preserving accuracy rate in both CPU and GPU versions, our work was mainly done using this algorithm.

**Intel Realsense Depth Camera D415 / D435**

We used the Intel Realsense Depth Camera D435 \ D415. The cameras provide us high resolution (1280x720) RGB aligned with depth frame. The frame rate is around 20 fps and it works in real time. Using this cameras allowed us to improve the accuracy of our calculations by calculating in 3D space.

**Intel Realsense SDK 2.0 (python wrapper)**

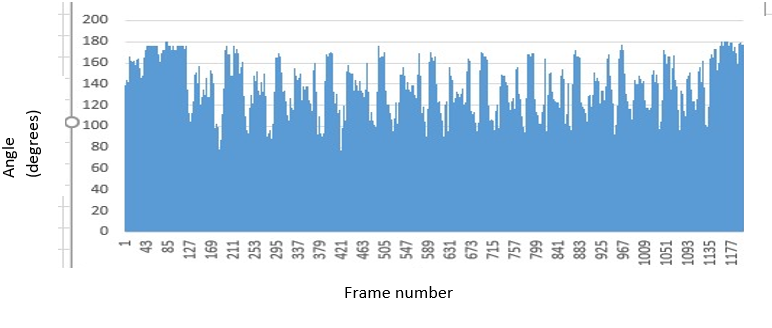
Software development kit that is free to use by Intel, most of it written in C++ and it has a python wrapper which we used in this project. The SDK has a lot of features such as capturing the camera streams, aligning the depth frames to RGB frames and more. The SDK also provides many written code examples, and an organized GitHub (LibRealSense).

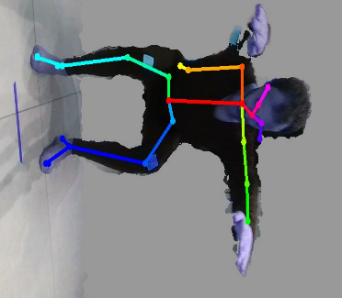
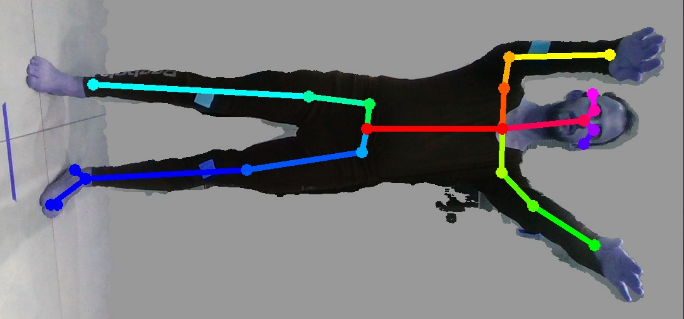
## **Overview**

The main goal of our project was to be able to generate accurate and correct data out of samples captured by depth cameras where subjects are performing sport exercises. After the data was extracted successfully, //not finished

## **Development Process**

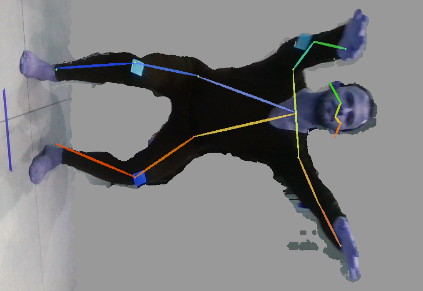
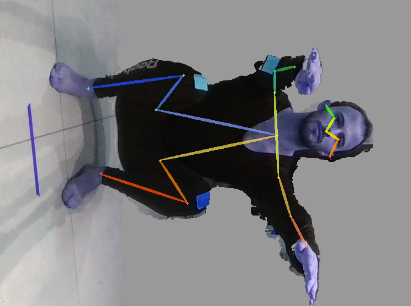
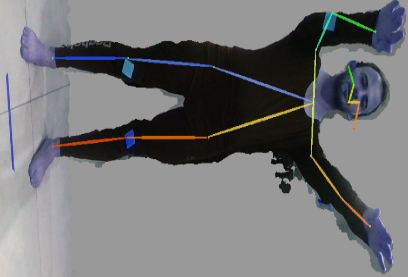
We started our development with a research about pose estimation algorithms, and Intel Realsense SDK. We found out that Intel offers a pose estimation algorithm – Cubemos using Realsense cameras, we decided to try it. We used a bag file (video) sample of a young subject performing repeated burpees exercise. We extracted RGB frames from the bag file, and used Cubemos to extract the skeleton key-points. We displayed the results as a graph where the knee angle is calculated. Note this calculation is done in 2D.



After these results we were still unsure about the accuracy of this algorithm (Cubemos) and decided to investigate other algorithms while the main objective was to find an algorithm with high accuracy of joint estimation. The research led us to OpenPose, a known pose estimation algorithm. We tested the algorithm on a sample of a young subject doing repeated squats and received inaccurate results.  
A large amount of frames has false predications  
as you can see in the pictures.  
We tried to investigate what is causing the low accuracy and found out that the accurate version of OpenPose was available for GPU only, and the version we used was on CPU. Trying to switch to GPU version failed due to very high demands that we didn’t have, Such as Titan X(GPU), 16GB GPU, etc.

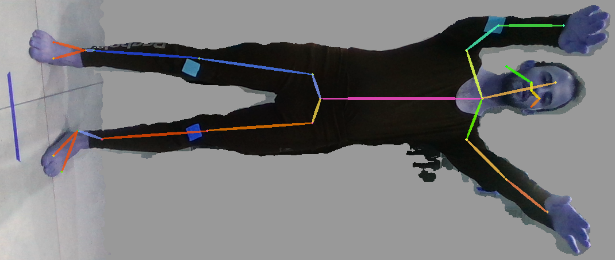
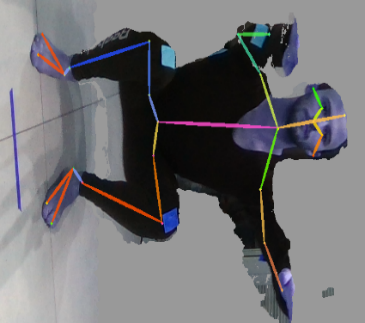
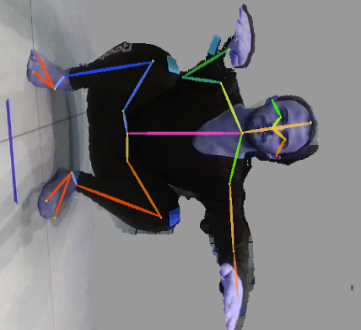
OpenPose skeleton estimation

While searching for pose estimation algorithms, we found an interesting pose estimation algorithm – AlphaPose, a pose estimation algorithm that is focused on high accuracy, AlphaPose also received high score on benchmarks, some even higher than OpenPose. It also has a CPU version that is not compromising accuracy for speed, resulting in low speed but high accuracy with CPU version. We started working with AlphaPose since then.



Alpha pose skeleton estimation, FastPose model.

FastPose model provided us with 18 skeleton key-points, but after discussing specifications with Dr.Agmon we decided we needed more points on the skeleton, especially neck point, hip points, ankle points. therefore, we switched FastPose model to Halpe model, which is 26 skeleton key-points model.



Alpha pose skeleton estimation, halpe model

At this point we were ready to start working on preliminary proof of concept. Including a working pipeline, data collection, skeleton extraction, angle calculation and results display.  
We started the work using Intel Realsense SDK with python wrapper and wrote a couple of scripts.

*LogGenerator* – script that takes a .bag file and runs it on a loop while breaking it into aligned RGB and depth frames. The scripts runs for a fixed amount of time trying to capture new frames via their timestamps. When the loop finishes we create 2 log files. The log files are basically lists of assigned RGB frame timestamp to the closest Depth frame timestamps. First log is assigning frames with unique RGB timestamps key, and the second log is assigning frames with unique depth timestamps key.

*Main* – this is our main script where we run most of the calculations. The script uploads the bag file, a log file, and a skeleton-key point json file. We start by breaking the .bag file to aligned frames and immediately stopping the playback. The playback is paused to avoid frame loss when executing calculations in between frame captures. While the playback is paused we find the matching RGB frame from the log file, and matching skeleton result to that frame. We use projection function (provided by Realsense SDK) to project the pixel point at the RGB frame to the correspondent XYZ point in 3D space. From this point we calculate the desired data.

After writing and testing the scripts, we tested it on short bag files.  
Displaying results of 5 second long depth video, where subject is performing 1 hand raise.  
Results are calculated in 3D space.

At this point we were ready to try longer videos.  
We used samples of a young subject doing repeated burpees for 40 seconds, both from side and front camera positioning.

We calculated the subject’s knee angle while doing burpees and received 2 kinds of results. One by using log which is matching RGB frames to depth frames with the closest timestamps, and the other one by using log which is matching depth frames to RGB frames with the closest timestamps.  
Results with color log, from side angle at 150 cm from camera:

Results with color log, from side angle at 150 cm from camera:

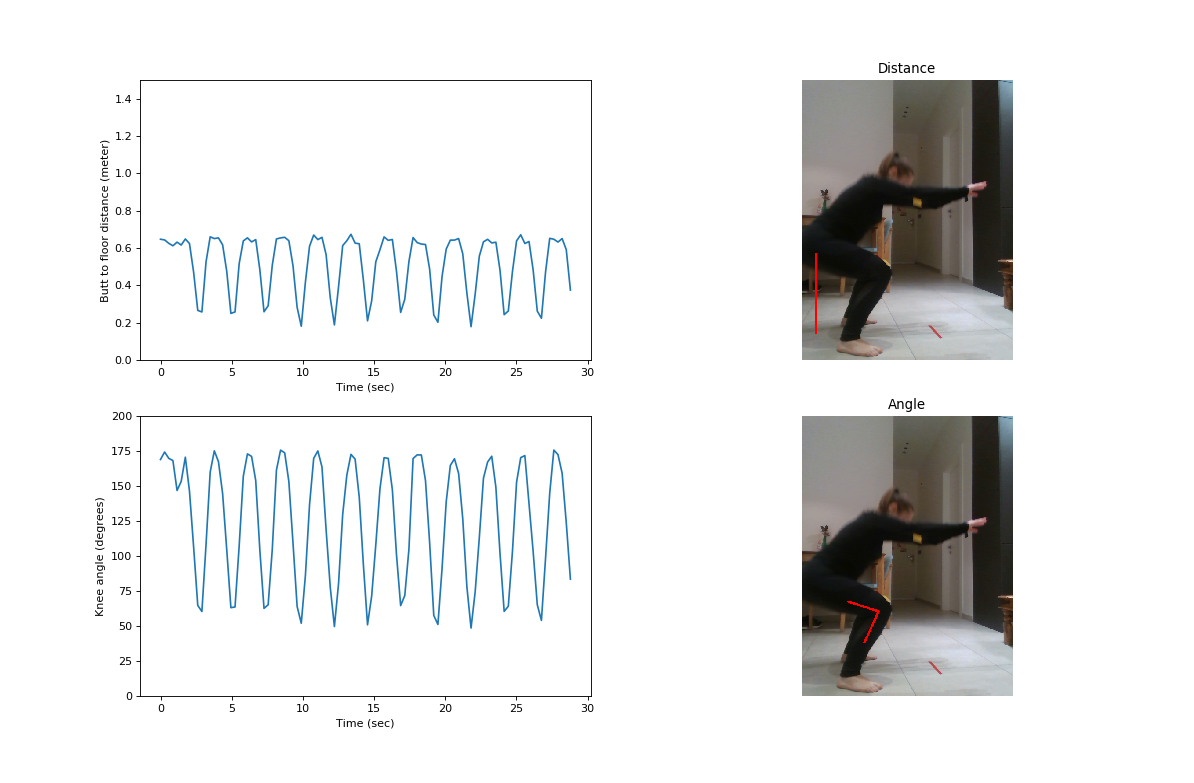
We noticed that while working with depth log produced more frames, it also produced a noisier graph. We decided to keep working with log by color (RGB frames are unique and for each RGB frame we map a depth frame with closest timestamp).

When trying to extract certain data such as head tilt angle, body tilt angle, we encountered a problem at the calculations which led us to the realization that the projection of a 2d pixel into a 3D space results in a depth point which lays on the outer body. In other words, when projecting onto a person shoulder for example, the depth we get is the distance from his shirt and not from the skeleton itself. This is very problematic, working with outer projection could tilt the vectors in relate to person size.  
  
In an attempt to resolve this issue, we tested Intel’s pose estimation algorithm – Cubemos, which has an integrated pose estimation algorithm with Intel’s depth cameras. Cubemos offers 3D skeleton key points extraction. Our hope was to find out that Intel’s algorithm handles the outer projection of depth points and returns a normalized depth estimation in relate to the skeleton.

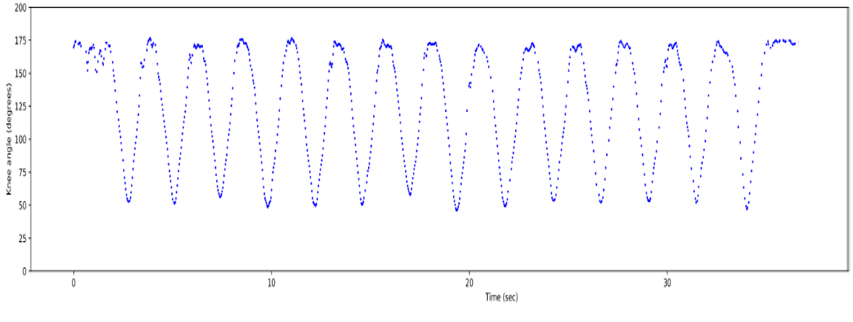
We run a test to check it, the test included extracting skeleton key points in 3D with Cubemos algorithm and comparing them with the same 2D point with projection function that is provided by Realsense SDK.

|  |  |  |
| --- | --- | --- |
| **Regular projection of the pixel** | **Cubemos (intel’s algorithm)** |  |
| 1.1170001 | 1.11 | Nose |
| 1.11 | 1.11 | Right shoulder |
| 1.149 | 1.14 | Left shoulder |
| 0.97500006 | 0.98 | Right hip |
| 1.07 | 1.10 | Left hip |
| 1.0630001 | 1.06 | Right elbow |
| 1.199 | 1.19 | Left elbow |
| 0.98100007 | 0.98 | Right wrist |
| 1.108 | 1.10 | Left wrist |

The depth was the same on both algorithms. In addition, we looked into the source code of Cubemos algorithm, and found out they use same projection function that we use on the pixel to extract the depth with no further calculations. Their process is identical to ours so unfortunately Cubemos did not solve our problem.

We created a graphic visualization script that creates a video which consists of the original video with basic drawing of the part we measure, and a graph that plots the calculated data as the video goes on.  
One of the challenges in generating the graph was to get rid of the outliers. We came up with a heuristic way to differentiate these points. It is done by calculating the median of the last 3 points, then comparing the new point in compare to the median and checking if it’s within a reasonable range from it (e.g. > 1.3X and < 0.7X). If so, we append the point to the graph, else we dump the point and expand the error range, each time a valid point is captured we reset the range to the default values. In addition, to smooth the graphs we used Gaussian filtering.

A frame from the graphic visualization



Graph after eliminating outliers and Gaussian filtering (original graph is above “Dana\_Squat\_1.5\_side\_color”)