Comparing Pose Streams

Programming Assignment #4
CS253 Spring 2014
Due 6:00pm Wednesday, March 2nd, 2016

Motivation

In PA3, you read in a pose stream (a.k.a. pose video), normalized it, and then measured the distances between its frames. In PA4 you will build on the ability to read pose streams, normalize them, and measure distances between frames. Now, however, you will begin to apply these steps toward our eventual goal: recognizing human actions in pose streams. More specifically, in PA4, your code will read in two pose streams from two different files, and you will measure the distances between every frame in one file and every frame in the other. This means that, for example, if one stream contains 100 poses and the other contains 200 poses, you will compute 100 x 200 = 20,000 distances.

As these assignments get more complex, two things should be happening. The first is that your programs are doing a lot more work. In the example above, 20,000 distance calculations is a lot. This makes it important to understand when your data is being copied and when it isn't. True, we are not grading you on efficiency (yet), but you want your program to run quickly so that do lots of development and debugging cycles. So be aware of your data.

The second is that modularity and encapsulation are becoming more important. Having two streams instead of one should not be a problem if your code is modular. If it is a problem, now might be the time to redesign it. I might also look at the table of distance measures and ask myself if that, too, should be an object.

Task

In PA4, your program will take three filenames as arguments. The first in the name of the input file, which is a sequence of poses in the same format as for the previous assignments. As in PA2 and PA3, you should read in and normalize the sequence of poses. The second argument is the name of another sequence of poses. You should read in and normalize this sequence as well. Note that the two sequences are normalized independently of each other. Also, one of the video sequences is shorter than the other. From here on out, we refer to the shorter sequence as the short sequence, and the other sequence as the long sequence. If the two sequences have the same number of frames, let the sequence read from the first filename parameter be the short sequence.

Let N be the number of poses in the short sequence, and M the number of poses in the long sequence (so N ≤ M). You will write an output file (using the 3rd argument for its filename) with N rows, one row per frame in the short sequence.

Each row will contain M distance values, one for every frame in the long sequence. Most importantly, the jth entry of the ith row should be the Euclidean distance (as defined in PA3) between the ith frame in the (normalized) short sequence and the jth frame in the (normalized) long sequence.

The values between rows in the output file should be printed as doubles (using <<) and separated by a single space. There should be no blank rows in the output file. The output file may or may not end with a newline (your choice). It should be an error if either file contains no poses.

Submitting your program

You will submit your program through Canvas. You should submit a single tar file, and this file should contain all the source files of your program *and a makefile*, but no object or executable files. The makefile should create an executable program called PA4. To grade your programs, the GTAs will write a script that untars your file in a clean directory, runs 'make', and then tests PA4 on novel inputs. If your program does not compile, whether because of an error in your makefile or an error in your code, you will receive no credit on any of the test cases. Note that as always, we will test your code on the department Linux machines.

Hints

1. If you have to scroll to see both the bottom and the top of any method, it is too big. Break it up into smaller methods. I know I have said this before, but not everyone is listening.
2. If you find the same code fragment in two or more places in your code, make it a method (or function). Repeated code fragments lead to bugs, particularly as your code gets larger.
3. Remember the principal of encapsulation. Objects combine data with code that operates on that data.
4. Your program is starting to get large. Test each piece of it independently. It means writing test code, but saves time in the long run.