UNIVERSITÄT SIEGEN

FAMS
Fertigungsautomatisierung und Montage

Naturwissenschaftlich-Technische Fakultät
Department Maschinenbau
Fertigungsautomatisierung und Montage
Univ.-Prof. Dr.-Ing. Martin Manns

Human motion retargeting to generate robot trajectory - replication of human-guided path.

Bachelorarbeit

im Studiengang Industrial Electronics and Automation

von

Jon Ander Santos Granero

(Erasmus – Universidad de Cantabria)

Matrikel-Nr. 1510919

Betreuer:

Univ.-Prof. Dr.-Ing. Martin Manns
M. Sc. Tadele Belay Tuli

September 2020

Abstract:

In this project we propose and study a methodology to generate a robot motion by the retargeting of human motion captured using an HTC Vive system. As a way to test the outcome, the human motion was obtained by the performer trying to mimic or follow a robot end effector in real time. Our task is to take the human motion as an input, use several techniques to process it and then reconstruct from it the motion that the performer was trying to achieve. We then compare this motion to the original motion recorded from the robot itself.

The processing of the data consists of applying PCA to obtain a two-dimensional projection, computing the approximate period of the movements by the location of points, sampling average points at points evenly spaced in time along the cycle, and then using a B-spline to reconstruct a continuous, smooth and closed curve.

A generic method is developed utilizing Python script, except for a first preprocessing step realized with Blender.

The results of our study show that from a dataset with variation in the order of 10 to 20cm (the human motion) we obtain a result with an error in respect to the motion recorded from the robot in the order of 1 to 5cm.

Table of Contents

Chapter 1 Introduction

1.1 Robot-human interaction

Robots are currently used in the manufacturing industry performing repetitive operation cycles, which must be carefully adjusted. For this reason, when planning an installation, it is important to know the accuracy and, more importantly, the repeatability of a robot for a particular action. Accuracy refers to how closely a robot can reach a target position, while repeatability refers to the ability of a robot to return to the same position several times. This means that a very high repeatability should lead to very similar movement cycles, regardless of whether they are close to the target.

At the same time, collaboration between robots and human operators has been a busy research area since the first applications of robotics, which continues to grow in importance as the use of robotics becomes more widespread. In industrial plants, the risk of accident is minimized by maintaining robots and people separate; no human presence is allowed in an active robot's workspace. By keeping the interaction at a minimum, the robot can be set to work without the risk of it endangering a human.

However, there is a trend nowadays of having robots interact more closely with people, not only in the manufacturing industry, but also in many other areas. As examples we can consider surgical robots in medicine; viewed as toys or marketing tools in the entertainment industry; and even in homes as cleaners or assistants. This means that, to ensure peoples safety and to achieve a fruitful cooperation, the movement of these machines must be carefully controlled. Not only the position and speed, but also the force that the robot exerts and its response to unexpected forces should be the subject of study.

As part of this robot-human interaction, we often want robots to replicate motions of humans or other agents. This can be in order to get the precise goals of the motions, or sometimes for aesthetic or psychological reasons (we want the robots to look “natural”, “human/animal” or at least “not scary”).

1.2 Objective

Robots are generally programed in one of three ways (British Automation and Robot Association, 2020):

- **With a Teach Pendant**: The most utilized method. The teach pendant is the interface between the operator and the robot. Using it the operator can program the robot point by point; the operator manually programs each point in a sequence. The points can also be programmed in by their coordinates in different coordinate systems. This method is easy and somewhat intuitive; however, one disadvantage is that it can be time-consuming to program in complex motions.
- **Lead through method**: Using this method, the robot is physically guided through the motions it will later repeat. While many collaborative robots have this function, it is not used often. While it is probably the easiest and most intuitive method, it is susceptible to inaccuracies introduced by the operator who programmed it.
- **Offline programming**: This method consists in using computer aided design to program and simulate the robot’s application. One of the biggest advantages is that the robot itself is not required for the program to be made and therefore the production line does not need to be interrupted to make changes. It also allows for more complex motions to be programmed, which would take longer amounts of time to be programmed manually.

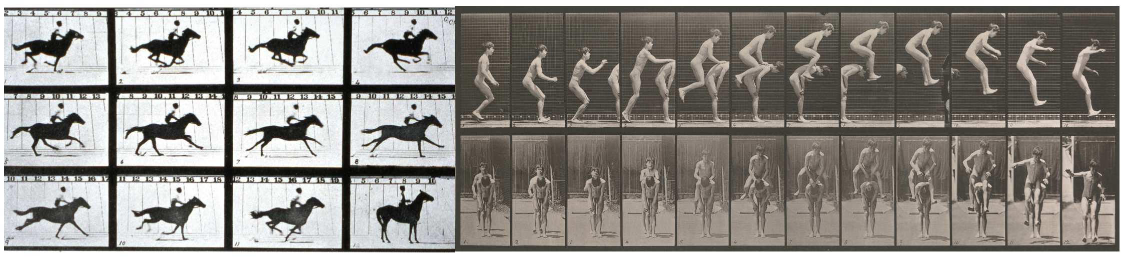
Our goal is to develop a methodology similar to the lead through programming, but which does not require the robot to be available for the procedure. An operator will perform a motion which will be recorded using motion capture equipment, and from this a path will be derived that can be later programmed into the robot, to be followed by the end effector. In this way, we have the robot precisely replicating the motion of the operator, who is free to make more complex motions more easily than in any form of point-by-point programming.

This is where motion retargeting is utilized. We start with motion capture data of a human replicating several repetitive motions of a robot arm in real-time. We will take this data from the human motion and try to reconstruct a precise mathematical trajectory that is supposed to imitate the original robot motion. We will finally compare that trajectory to the actual motion of the robot, which was also recorded as a reference.

1.3 Motion capture and motion retargeting

Motion capture is the process of recording the motion of the different parts of an object, animal, or person. To this end, different points in the object are marked and its motion is recorded by one or several cameras depending on the intended goal. If several cameras are used, the 3D position of each marked point at each time can be recovered and stored as numerical data.

Capturing with precision the motion of people or animals has interested artists and scientists since long. Early pioneers of capturing real life motion before the cinematographer existed were Eadweard Muybridge and Étienne-Jules Marey who, around 1880, realized several series of photographs of animals (typically horses galloping) or humans in which they marked several points in order to follow their motion or put a background with marks that allowed to make measurements. See the following figure for examples of Muybridge's work.



(a) *(b)*

Figure 1: Eadweard Muybridge's "The horse in motion" (a) and "Children playing leapfrog" (b), c.1880. Muybridge devised a system of several cameras whose shutters where triggered in a short period of time, either by wires triggered by the horse's chest while galloping or manually by turning a wheel with several switches in it. Part of Muybridge's motivation for his horse series was that one of his mentors, businessman and racehorse owner Leland Stanford, thought that the depictions of horses galloping traditionally made by painters, with all four feet in the air, the front ones pointing forward and the back ones pointing backwards were not accurate. Muybridge devised a method to take 12 to 16 consecutive photographs during a single jump of the horse and proved that Stanford was indeed right. The horse has all four feet in the air while galloping, but only while they are collected beneath his body, and not while they are extended to the front and back.

Motion capture is often used in combination with motion retargeting: the motion captured is transferred to a different subject (real, such as a robot, or imaginary, such as a character in a fantasy or animated film) so that the new subject reproduces the original motion. In the second half of the twentieth century motion capture and motion retargeting have found many practical applications. Among its uses we can mention:

- **Crash tests** in the automobile industry, where commercial or prototype cars are crashed having mannequins (known as "dummies") or, at the beginning, dead human bodies as passengers. Several marks are made in the passengers or the structure and the crash is recorded in order to analyze which parts of the car structure or of the security devices (seatbelts, airbags) need to be improved to minimize harm.

- **Entertainment industry** (animation, film, videogames), where the motions of real actors are captured and then transferred to the characters in the movies. The first movie including a main character fully created by motion capture was *Star Wars I: The Phantom Menace* (1999, the character is Jar Jar Binks (IMDb, s.f.), (Gray, 2014)), but the technique became so common in the next years that two of the three nominees for Best Animated Movie at the 2006 Academy Awards used motion capture and the next year the Pixar movie *Ratatouille* included a stamp labelling the film as "100% Pure Animation – No Motion Capture!" in its ending credits.
- **Biomedical applications**, where *gait analysis* (i.e., the systematic analysis of human motion) is used be it to assess and treat patients with reduced mobility, to improve the performance of professional athletes, or to produce protheses that accurately reproduce the human motions.
- **Security or military** situations in which the recordings of security cameras are analyzed in order to detect potential threats, robbers, etc.
- **Manufacturing industry**, where recording the movement of operators in real time can reduce the risk of accidents. For instance, it can allow robots to recognize operators and avoid causing safety hazards in safety-critical environments. (Kubota, et al., 2019)

Our end goal is to obtain a motion path that can be programmed into the robot, from a path obtained from a human operator. This process is called motion retargeting. Motion retargeting consists, broadly, in translating movements made by one actor into another. It has applications outside of robotics; for instance, it is used in animation for one of two reasons: applying an existing motion to a character is often more worthwhile than creating a new animation and applying a real-life movement to something that only exist digitally can help make it more physically accurate, and therefore more believable. It is also used to translate movements made by real life performers to an animated character.

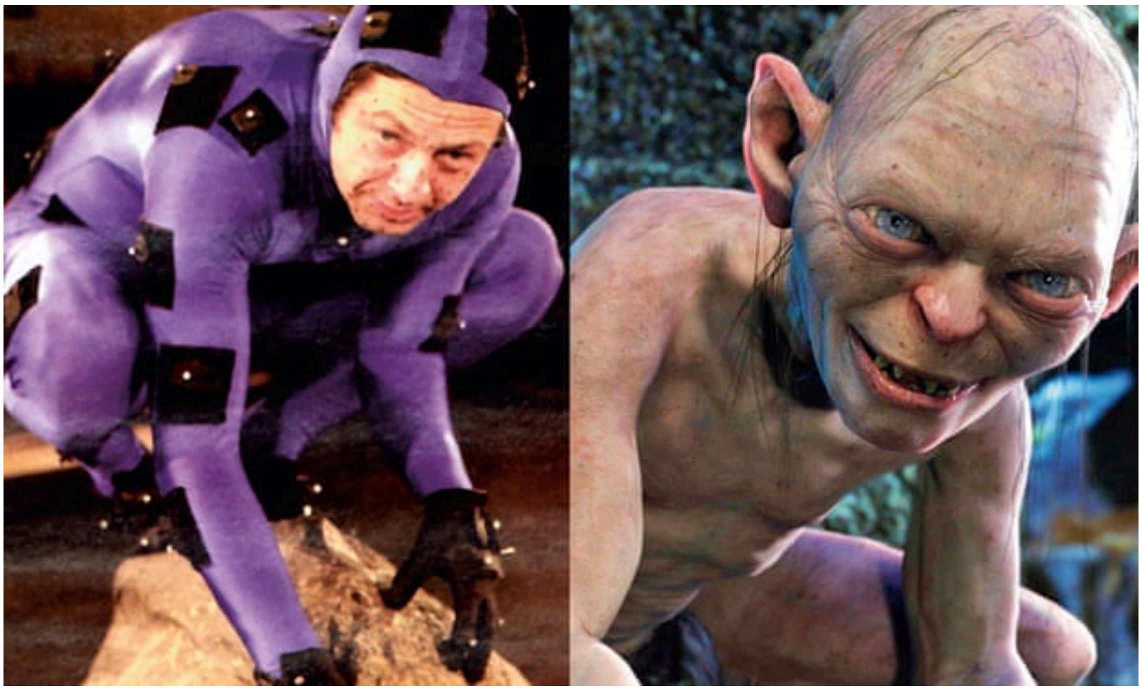


Figure 2: One of the best-known uses of motion capture in the entertainment industry, Andy Serkis plays Gollum in the film The Two Towers, 2002. https://www.theguardian.com/global/2015/dec/06/andy-serkis-film-hobbit-king-kong-fungus

In robotics, motion retargeting is commonly used for robot programming. For instance, in a manner analogous to how animated characters can recreate movements made by humans, humanoid robots can be programmed to replicate movements made by human performers. In both cases, it is common that adjustments have to be made to the motion before it can be applied to the end actor. An example of motion retargeting for a humanoid robot can be seen on Figure 3.

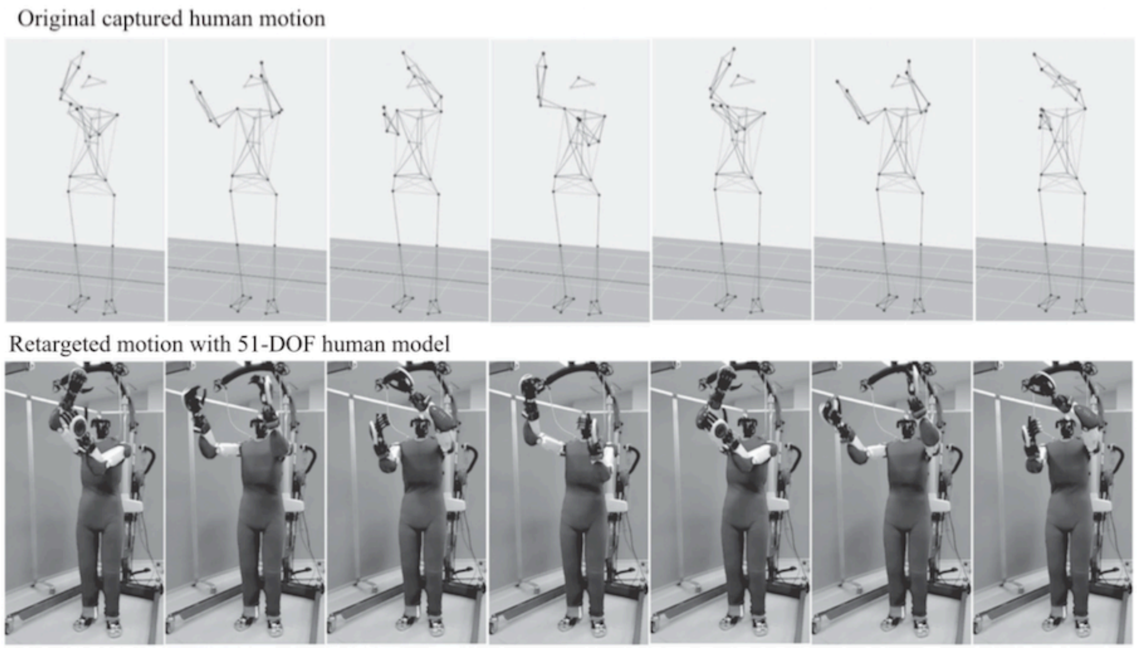

Original captured human motion
Retargeted motion with 51-DOF human model

Figure 3: Example of motion retargeting for a humanoid robot. Top, the motion recorded from a human performer. Bottom, the robot following the movements after retargeting (Ayusawa & Yoshida, 2017).

We can also see it is applicable to manipulators and not only to fully humanoid robots in Figure 4.

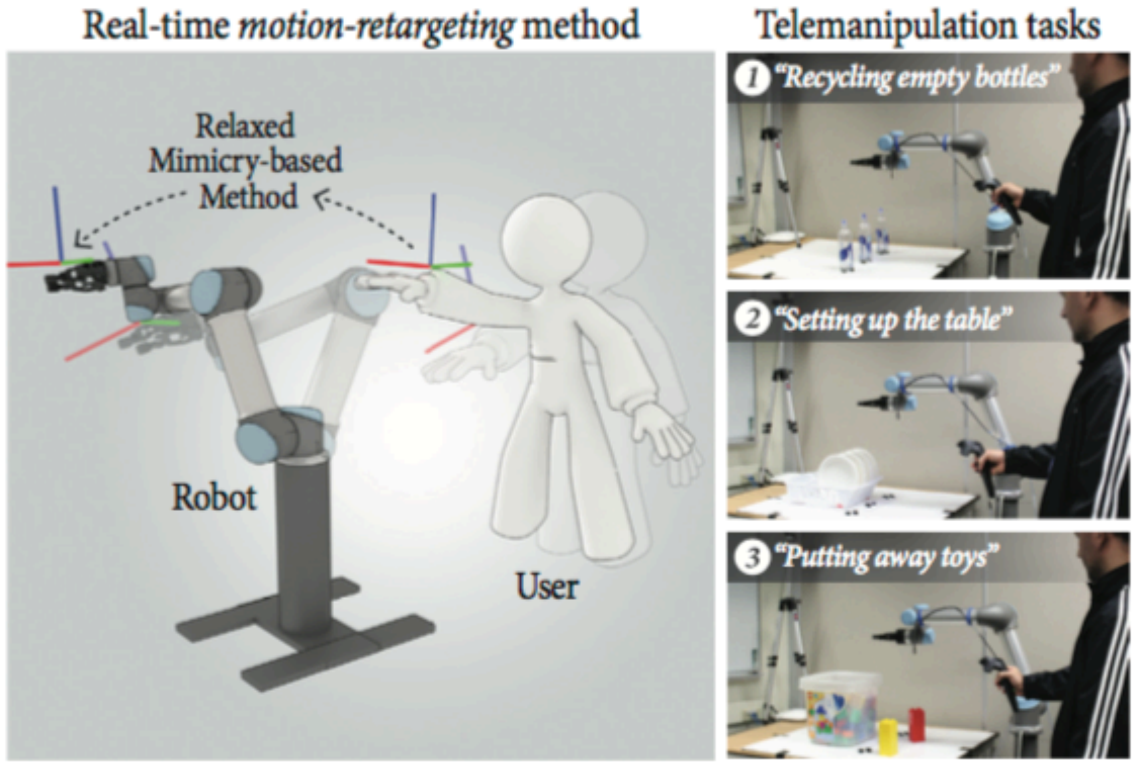

Real-time motion-retargeting method
Relaxed Mimicry-based Method
Robot
User
Telemanipulation tasks
1 "Recycling empty bottles"
2 "Setting up the table"
3 "Putting away toys"

Figure 4: Example of motion retargeting applied to a robot arm manipulator (Daniel Rakita, 2017). In the referenced project, they propose a method of real time motion control through mimicry.

And another example in Figure 5:

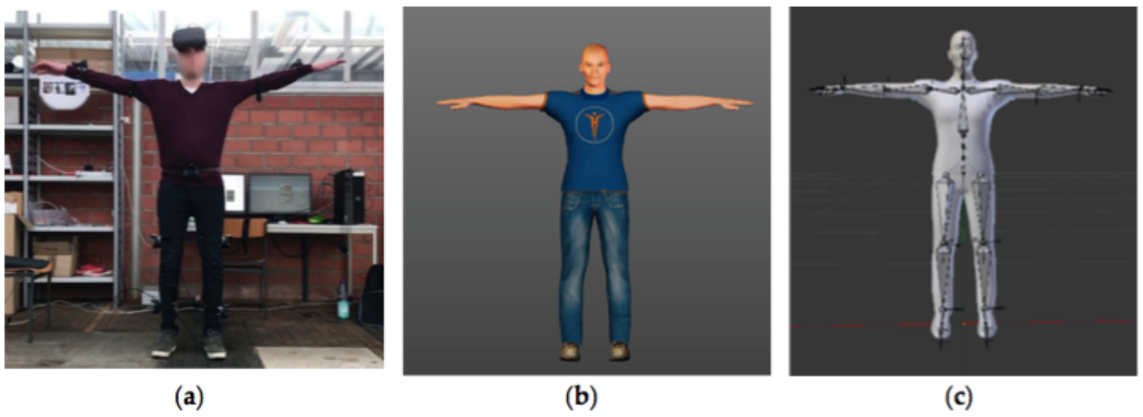


Figure 5: Another example of retargeting, this time from a human to a humanoid 3d Model (Tuli & Manns, 2020). In this paper they use retargeting to track the motion of an operator in a 3D environment, allowing the robots program to know where the operator was and allow for collaboration.

Generally, motion capture equipment obtains the motion of each joint in an object or person in the three-dimensional space, but it is almost never useful to directly apply the movement of the performer to the robot or character. Sometimes the target is specifically made to resemble the actor with the purpose of minimizing the changes required, or because the target is meant to represent the actor. There are three main reasons why adapting the movement might be required.

First, the performer and the target to which we want to apply the motion (be it a robot or an animated character) can be different in size, proportions, or even their whole structure. This means the movement data does not translate well and will lead to errors if directly applied to the target. In some cases, it cannot be applied at all. In either case, it will have to be adapted to the new skeleton they are applied to.

Second, we might simply not be interested in all the data provided by the measurement. In these cases, the relevant joints can be selected, and the motion for other joints can be deduced from these by applying constraints.

Third, there are errors in the movement of the performer that need to be corrected. This can mean mistakes made by the performer, variation inherent to real world movement or errors in the measurement and recording of the data.

Since our objective is to retarget a motion produced by a human to an UR robot arm, we face all three of the previous reasons to adapt the data. The robot is not humanoid, meaning we will not be able to apply the movement recorded from a human without extensive changes. Even if we could, it would not be useful, as the performer was only following the desired path with his right hand. Instead, we can select the joint corresponding to the right wrist and extract only its movement from the whole motion capture file. This is the movement that will be processed to obtain a path for a single of the robot's joints, corresponding to the tool it would be using. The processing of the data is also meant to remove the error and variation in the movement.

The way the robot is designed, it easily allows for inverse kinematics to be used to calculate the motion for the other joints once the desired movement of the tool is known, and this is how it is usually programmed.

1.4 Our work

Our starting data was obtained in a previous (unpublished) study within the *Lehrstuhl für Fertigungsautomatisierung und Montage* (Chair for Production, Automation and Assembly) of the University of Siegen. The data consists of files in the Biovision Hierarchy (*.bvh) format. That is, they are *ASCII files that contain motion capture data for three-dimensional characters, used by "3ds Max's Character Studio" and other 3D animation* *programs to import rotational joint data; developed by Biovision as a standard format to save biped character motion data*. (FileInfo Team, ret. 2020).

We have eight plus eight files, containing eight motion captures from the robot arm plus eight simultaneous motion captures from a human who was trying to exactly replicate the robot's motion in real time. The robot used is a Universal Robots UR5, a lightweight robot – that is, a light robot designed to work with humans which is not designed for a specific task or environment. The UR5 specifically has a reach of 850 mm, a payload of 5 kg and weights 20.6 kg. Each of the eight motions consists of repetitions of a basic cycle, tracing a circle in four of them and a square in the other four.

The different tasks that we need to perform with this data are described in detail in **Error! Reference source not found.**. They are implemented in the python Programming language except where noted:

- **Section 3.2 : Preprocessing the data with blender and bvhplot.** Python is not able (at least not directly) to read the original files, which are in the .bvh format. Our first task is to use package *bvhplot.py*, previously developed by the research group, to extract from these files the information that we need and store it in a way that can be readily used by *python*. We wrote a script *BVHtoMatrices.py* for this task. Before running the script, we needed to change the reference system in the *.bvh files to the one that *bvhplot* expects. We did this using the free and open-source 3D computer graphics software "Blender". Blender is also used in other parts of the project to visualize the motions that we are working on at different stages in a 3D environment.

- **Section 3.3 : Computation of periods.** Each file contains several periods, or cycles, of the basic motion (between 50 and 130 approximately). We want to separate each file into its individual periods for two reasons: on the one hand we are only going to reconstruct a single period; on the other hand, the different cycles are not going to be exactly equal. We are later going to extract an "average" of the individual cycles to minimize the imperfections. These imperfections are certainly present in the human motion, but also in the robotic one, to a lesser extent, due to the fact that the method of measurement is completely external to the robot; the data was obtained with a motion capture rig.

- **Section 3.3 : Projection via PCA.** The motion we want to reconstruct is 2-dimensional, or at least it should be. The same imperfections mentioned in the previous paragraph make it not lie exactly in a plane, so we want to project it back to a plane. One possibility would be to just forget the third coordinate in all data points, but the human reproducing the motion may have inadvertedly introduced an inclination to it, so that the (approximate) plane in which the human arm is moving is not really horizontal. Moreover, we want our scripts to work in a relatively general context in which finding manually what the projection plane is would be unfeasible or, at least, undesirable. Thus, a better approach is to use the statistical method of *Principal Component Analysis* to find the plane that best fits the data points, and then project to that plane. We give some theoretical background on PCA in Section 2.1 , although we are really not going to need much of it. For our computations we use the PCA tools contained in the Python module SciKit-learn (Pedregosa, et al., 2011), so that from the programming point of view the PCA part is for us a "black-box".

- **Section 3.5 : Selection of a sample of points in an "average cycle"**. In this section we do two things, but they are performed at the same time:
  - *Reducing the number of points*: We have in the order of 300 to 450 data points per cycle in the motion. This is good for the final comparison of our reconstructed motion with the original one, but we want to reconstruct a motion from a smaller set of points, again for two reasons: on the one

hand, in a real life situation where a huge number of individual motions may need to be stored/analyzed/reconstructed, we may want to save storage space by not keeping the whole input data but only a representative sample from it**.** On the other hand, we want to obtain a “smooth” reconstructed motion. Having too many points is actually an obstacle for that since the individual points are approximate: we need them to be significantly more separated from one another than the error produced by the approximation; having less points (20 or 30 per cycle) we can apply interpolation techniques.
    - *Averaging*: Once we find the (ideal) instants within each period at which we are going to sample, we compute the average point for “those” instants over all the cycles, so that our sample represents an average cycle.

- **Section 3.6 : Construction of trajectories via splines.** Once we have our list of sample points, we want to find a continuous curve from them, that is, to interpolate. The technique that we use is to construct a *spline* (a piece-wise polynomial parametrized curve) using our sample points as *control points*. Splines are not really interpolating the data set, since the curve they produce does not pass exactly through the points. We give up this feature in order to have another important feature in the final motion: even if it consists of several polynomial curves glued one after another, we want the gluing to make the curve not only continuous but also *twice differentiable*: the velocity vector changes continuously (first differentiability) and also the acceleration vector or, equivalently, the curvature radius of the curve, is continuous. We achieve this by using “degree-3 C^2 B-splines”. The “B” here stands for “basic” splines, the type of splines most used for curve reconstruction. We give some theoretical background on the B-splines that we are using in Section 2.2

- **Section 3.7 : Comparison with the original motion.** We here compare the results obtained for the human motion to the original robot’s motion. More precisely, we compute three things: (1) we perform PCA to the robot motion and compare the normal vectors to the fitted plane for the robot and human. (2) we repeat the computation of period for the robot and compare the outcome to what we got for the human. (3) we look at how far is each point of the B-spline reconstructed for the human for the original motion of the robot (we first translate the reconstructed human motion so that they it has the same center as the robot’s).

Before going to the actual description of the tasks we have performed (in **Error! Reference source not found.**), we are including a Chapter 2 a theoretical description of two of the tools we use: Principal component analysis and B-spline curves.

Chapter 2 Theory and background

Throughout the work we are going to manipulate the motion capture data that we are giving in several ways. Some of the things that we are going to do are very basic from the mathematical point of view, such as computing distances, averages, etc. and they need no further explanation.

But two of the techniques we use are a bit more sophisticated: Principal Component Analysis (which projects a given high-dimensional data to lower dimensions) and B-spline curves (which interpolates a curve of the desired degree and differentiability passing close to a set of given points). In this chapter we give some theoretical background for them.

2.1 Projecting to 2D: principal component analysis

Although the motions that have been recorded lie in 3D, they are supposed to be two-dimensional, and the third coordinate in them can be considered “noise” or undesired measurement that we want to remove before doing anything with the data. We want to get rid of this extra component before doing anything else with the data for two reasons: (a) it is a first step in our goal of reconstructing the original robot motion, since the robot’s motion is (almost) truly 2-dimensional and the third dimension in the human motion is an error introduced by the human. (b) having a motion as close to periodic as possible for the human will make the subsequent steps (finding the period and finding sample points in the “average cycle”) much more accurate. It can be seen in the graphs for the captured motion of the human that a good amount of the dispersion among different cycles comes from the third dimension, the one we want to eliminate.

This is an example of “dimensionality reduction”: the process of projecting data with a “large” number of variables to a subspace of lower dimension. Several benefits of dimensionality reduction are:

- It can save storage space and transmission time, if we need to send our data.
- It can also save processing time since whatever we want to do with the data later will be much more complicated if we have many variables.
- It can help in finding correlations between the variables, or in analyzing other statistical parameters. A first example is fitting data by a line, which is nothing but reducing dimensions to one.

Of course, we do not want a random projection; we want the one that keeps as much as possible of the original information in the data. There are several different methods for achieving this, but by far the most used one is *Principal Component Analysis*.

In fact, some sources (Bartholomew, 2010) mention Harold Hotelling as the inventor of Principal Component Analysis. The truth is that, although Hotelling discovered it independently and gave it its name, the method had already been invented by Karl Pearson in 1901 (Sewell, 2007). Some advantages of Principal Component Analysis over other dimension reduction methods are:

- It does not require to “forget” some of the data, as *feature selection* methods do. That is, the result that we obtain is affected by any single data point. (In some contexts, PCA can be used giving different weights to the data points, if some are more significant than others, but we will not do this).
- It is *linear* in the double sense of producing a linear subspace as output and that all the computations that need to be done are linear algebra, which works very fast in practice. In some applications the data is assumed to lie in a lower dimensional space which is not linear (e.g., surface reconstruction) and there one really needs to use nonlinear methods, but this is not our case. We are looking for a plane.

We employed a standard PCA library from the Python module SciKit-learn (Pedregosa, et al., 2011), which is an open source tool. Below we summarize what Principal Component Analysis does.

The starting point is a set of multidimensional data points, $p_1, \ldots, p_n$, where $p_i = (x_1^i, \ldots, x_k^i)$. From this we compute the covariance matrix M, a square matrix of size $k \times k$ that has in its i, j entry the covariance of the vectors $(x_i^1, \ldots, x_i^n)$ and $(x_j^1, \ldots, x_j^n)$. (In particular, it has the variance of $(x_i^1, \ldots, x_i^n)$ in its ith diagonal entry).

Since the covariance matrix is symmetric, it can be diagonalized via an orthonormal change of coordinates, and the next step is to compute this diagonalization. This gives us a diagonal matrix D whose entries are the eigenvectors of M, but also an orthonormal matrix B such that

$$D = B^{-1} \, M \, B.$$

The columns of B are the basis in which M becomes diagonal, that is, the eigenvectors corresponding to the entries of D, in the same order. These eigenvectors are the "principal components" of the data.

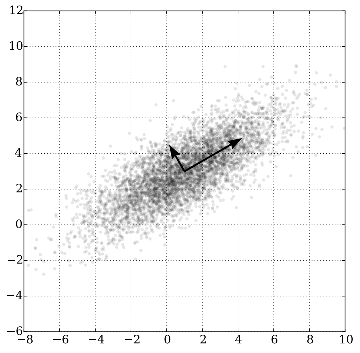


Figure 6: The idea behind PCA. The two black vectors represent the principal directions, together with their magnitudes. Source: Wikipedia, user nicoguaro. https://en.wikipedia.org/wiki/Principal_component_analysis

To reduce dimension, we simply project the original data set orthogonally to the linear subspace generated by the first few eigenvectors with the largest eigenvalue. Depending on the context, the number of dimensions to be kept can be decided a priori (as in our case, where we are reducing from three to two) or it is decided once the eigenvalues are known, by either posing a threshold below which all eigenvalues are neglected or by clustering the eigenvalues and taking those in the biggest cluster.

We can give two interpretations of the principal components, one geometric and one statistical:

- From a geometric point of view, the covariance matrix represents the ellipsoid that best fits our data; its eigenvectors are the principal directions (axes) of the ellipsoid and its eigenvalues are the lengths of the ellipsoid semi-axes.
- From a statistical point of view, for each unit vector u we have that $v^{-1} \, M \, v$ represents the variance along the direction of u. That is, the first principal component is the direction where the data has the highest variance, the second principal component is the direction with the highest variance among those orthogonal to the first, etc.

The use of PCA that we make can be considered a "toy example", since we are only reducing dimensions by one, from three to two. But PCA is also used in full form in motion capture and related areas, since the data of the motion contains trajectories of several points (typically, joints in a robot or particularly important points in a human's body or face). This makes each data point to contain many variables (three per joint if only position is measured, six if also orientation is measured) which seems to imply that the moving object has many degrees of freedom. But sometimes one knows or thinks that this is not the case; the angles and positions of different joints can be heavily correlated, be it because of mechanical constraints (length of parts or bones) or by dynamical features (the motions of the different parts are not independent but respond to a simpler pattern that can be modeled with a few variables. This is explained for example in (Du, et al., 2016), where a version in which the different scale of the several moving parts is considered, and in the references mentioned there.

2.2 Interpolation between the points: Splines

We now address the following question. We are given a number of points $p_1, \dots, p_n$ in space (we do not need, or actually use, that our points lie in a plane after performing the PCA), together with their associated times $t_1, \dots, t_n$. We want to find a parametrized curve $S(t)$ defined on the interval $t_1 \leq t \leq t_n$ that approximates the original motion, that is, with $S(t_i) \approx p_i$.

Two natural solutions for this problem are interpolating by line segments and interpolating by a polynomial curve of degree $n-1$, but each of these solutions has disadvantages.

Interpolating by line segments.

This means we take

$$S(t) = \frac{(t_{i+1} - t)p_i + (t - t_i)p_{i+1}}{t_{i+1} - t_i}, \qquad t_i \leq t \leq t_{i+1}.$$

in each interval. This gives, in this interval, a linear function that goes from p_i at $t = t_i$ to p_{i+1} at $t = t_{i+1}$. The problem with this approach is that the curve $S(t)$ does not have a continuous derivative at the ends of the intervals. That is, this describes a motion where the velocity vector changes instantly, which on the one hand is not realistic and in the other hand, if passed onto a robot, would produce very sudden turns that could damage the machinery.

Put differently, it would be desirable that the functions we use to construct our curves have at least the first derivative, and perhaps higher order ones, continuous. A curve whose first k derivatives are continuous is called C^k*-continuous*.

Approximating by a polynomial of degree n-1.

If we look at the X and Y coordinates of S separately, these are functions of which we know n values. If we assume the two functions to be polynomials, that is,

$$X(t) = a_k x^k + \cdots + a_o,$$
$$Y(t) = b_k x^k + \cdots + b_o,$$

we can use the values that we know to find the coefficients a and b, as long as the number of coefficients to be found is not larger than the number of values that we know. That is, we can find a solution as long as the degree of the polynomials is at least $n-1$, and if we take $k = n-1$ the solution is going to be unique.

This method may look like it is the best possible, because it gives a curve $S(t)$ of the smallest possible degree with the following desirable properties: (a) it exactly goes through our points, ($S(t_i) = p_i$ for every point), and (b) *all* the derivatives are continuous. But it has the disadvantage that the polynomial curve obtained makes a lot of turns that were not supposed to be there. That is, the curve that we interpolate is exactly where it has to at the values t_i, but it may be very far from what we want at intermediate values, and it may have unnecessary changes in direction. This is called *Runge's phenomenon* (Ford, 2008) and is illustrated in the following Figure:

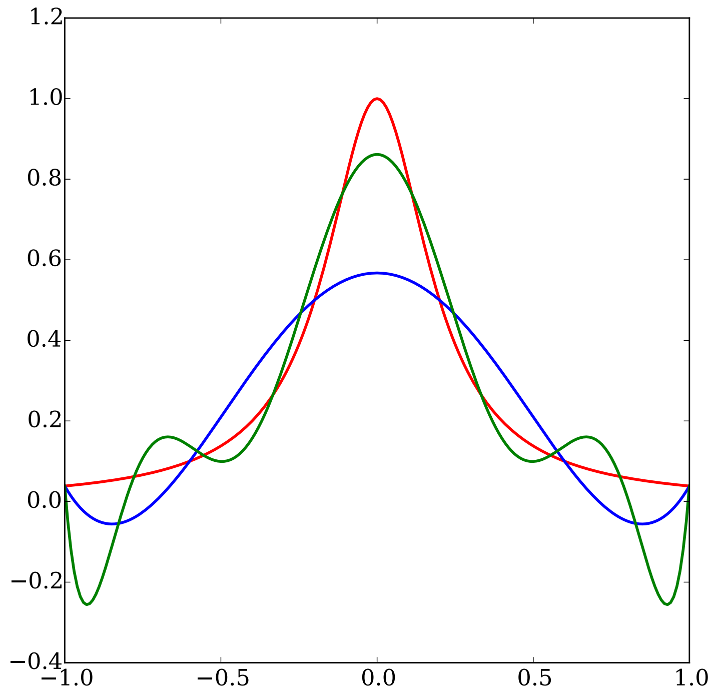


*Figure 7: **The Runge phenomenon**. The red curve is the graph of the so-called Runge function f(x)=1/(1+25x^2) in the interval [-1,1]. The blue curve is constructed dividing the interval in five equal pieces and interpolating a polynomial of degree five. The green curve does the same with nine pieces and degree nine. As seen in the picture, the interpolating polynomials have additional ups and downs that were not present in the original curve. The phenomenon persists (and gets worse and worse) in higher degrees. Source: Wikipedia, user nicoguaro. https://en.wikipedia.org/wiki/Runge%27s_phenomenon*

To solve these issues, we are going to use B-splines. Our main source for this topic is Section 1.4 of the book (Patriakalis & Maekawa, 2010).

A *spline* is a parametric curve that is defined piecewise by polynomials. That is, we have a curve $S(t)$, defined in an interval $a \leq t \leq b$, but we consider the interval divided into pieces by certain points $a = t_1 \leq t_2 \leq \cdots \leq t_n = b$ and require that in each subinterval $[t_i, t_{i+1}]$ we have that S is a polynomial curve. The values $t_1, \dots, t_n$ where we change from one polynomial to the next are called the *knots* or the *knot vector* of the spline. A priori this is not necessary, but in our application the knot will consist of the sampling values for our points and the distance between every two consecutive knot values (the “step”) is always going to be the same.

A spline is of degree k if the polynomials in it are of that degree, and it is C^k*-continuous* if the first k derivatives of each polynomial at its final point coincide with the first k derivatives of the next polynomial at the starting point. For example, the approximation by line segments that we described above is a C^0-continuous spline of degree 1, and the approximation by a single polynomial is a C^∞-continuous spline of degree $n - 1$. There are several ways of constructing splines, but we are going to look at *B-splines*.

B-Spline functions: Recursive definition and properties

The B-spline curves will use certain functions called the B-spline functions that we define here. If we are given a knot vector $t_1 \leq t_2 \leq \cdots \leq t_n$, we define for each order $k = 1, 2, 3, \dots$ the B-spline functions of order k for that knot in the following recursive way. Observe that “order k" means the same as “degree $k - 1$”:

- For order 1, we define

$$S_{i,1}(t) = \begin{cases} 1, & t_i \leq t \leq t_{i+1} \\ 0, & otherwise \end{cases}$$

- For higher order, we define

$$S_{i,k}(t) = \frac{t - t_i}{t_{i+k-1} - t_i} S_{i,k-1}(t) + \frac{t_{i+k} - t}{t_{i+k} - t_{i+1}} S_{i+1,k-1}(t)$$

One remark is that strictly speaking this definition gives us only functions $S_{i,k}(t)$ with i up to $n-k$ and not n, because in order to define the function $S_{i,k}$ we need to use the knots up to t_{i+k}. To solve this, in the usual theory the last entry t_n of the knot is repeated k more times; that is, we consider that we have additional values t_{n+1}, …, t_{n+k} in the knot, except they are all equal to t_n. In our case there is a more convenient solution. Since we are seeking a periodic curve, *we consider an infinite periodic knot that never ends, but compute the functions $S_{i,k}$ only for one period*.

The most important properties of these functions for each fixed k are:

- They are of degree $k-1$, and they are C^{k-2}-continuous, which is the highest continuity with which we can glue two different polynomials of degree $k-1$. (Here, saying that $S_{i,-1}(t)$ is C^{-1}-continuous means that it is not continuous at all, which is clear in its definition since it is a step function).
- Each $S_{i,k}$ is positive in the interval $[t_i, t_{i+k}]$ and zero outside that interval. This implies that for a fixed value of t only k of the functions $S_{i,k}$ are nonzero. More precisely, if t is in $[t_i, t_{i+1}]$ then only $S_{i,k}$, $S_{i-1,k}$, ..., $S_{i-k+1,k}$ are nonzero at t.
- They are a “partition of unity”. That is, their sum equals 1:

$$\sum_{i=1}^{n} S_{i,k}(t) = 1$$

B-Spline functions: Explicit form

We are now going to give the explicit form of the B-spline functions assuming that the knot is uniformly distributed. That is, the difference $t_{i+1} - t_i$ is the same for all i. Denoting this difference h we can simply substitute $t_i = h\,i$ in the formula defining the B-spline functions. That is:

$$S_{i,k}(t) = \frac{1}{k-1}\left[\left(\frac{t}{h} - i\right) S_{i,k-1}(t) + \left(i + k - \frac{t}{h}\right) S_{i+1,k-1}(t)\right]$$

We are interested in cubic splines (splines of degree three) which in the formulas means order $k = 4$, so we compute the explicit formulas for $k = 1,2,3,4$ one by one. The computations have been made by us via the recursive formulas, but we omit details and give only the final result:

- For $k = 1$ there is nothing to say: $S_{i,1}$ is the step function that takes the value 1 in the interval $[t_i, t_{i+1}]$ and zero elsewhere.
- For $k = 2$, $S_{i,2}$ is a combination of $S_{i,1}$ and $S_{i+1,1}$, so that it is zero outside the interval $[t_i, t_{i+2}]$. In this interval it takes the following form:

$$S_{i,2}(t) = \begin{cases} \dfrac{t}{h} - i, & t_i \le t \le t_{i+1} \\ i + 2 - \dfrac{t}{h}, & t_{i+1} \le t \le t_{i+2} \end{cases}$$

  That is, the function goes from 0 to 1 in $[t_i, t_{i+1}]$ and from 1 to 0 in $[t_{i+1}, t_i]$, linearly. This function is plotted in green in Figure 8.

- For $k = 3$, $S_{i,3}$ is a combination of $S_{i,2}$, $S_{i+1,2}$ and $S_{i+2,2}$, so that it is zero outside the interval $[t_i, t_{i+3}]$, and it takes the following form in the three subintervals, plotted in red in Figure 8:

$$S_{i,3}(t) = \begin{cases} \frac{1}{2}\left(\frac{t}{h} - i\right)^2, & t_i \le t \le t_{i+1}, \\ \left(\frac{t}{h} - i - 1\right)\left(i + 2 - \frac{t}{h}\right) + \frac{1}{2}, & t_{i+1} \le t \le t_{i+2} \\ \frac{1}{2}\left(i + 3 - \frac{t}{h}\right)^2, & t_{i+2} \le t \le t_{i+3} \end{cases}$$

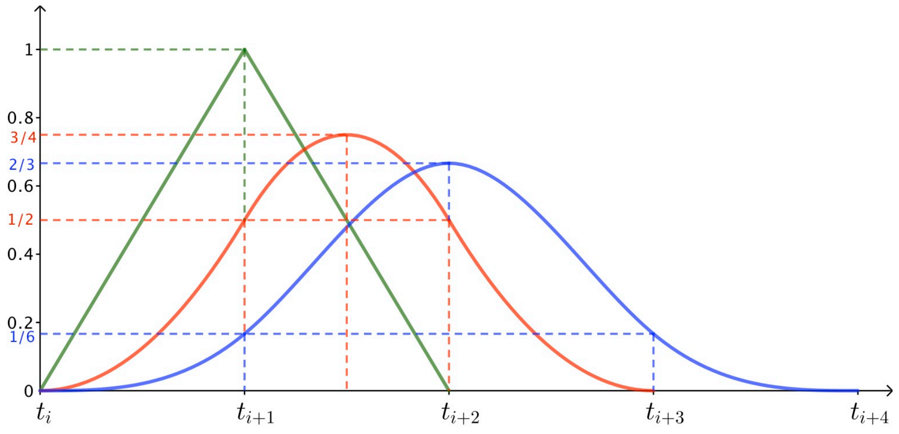


*Figure 8: The **B-spline functions** $S_{i,k}$ for $k = 2$ (green), 3 (red) and 4 (blue). The last one is the one we are going to use. It is a C^2 cubic spline function.* Figure made with GeoGebra

- For $k = 4$, $S_{i,4}$ is a combination of $S_{i,3}$, $S_{i+1,3}$, $S_{i+2,3}$ and $S_{i+3,3}$, so that it is zero outside the interval $[t_i, t_{i+4}]$. Its form in the four subintervals is:

$$S_{i,4}(t) = \begin{cases} \frac{1}{6}\left(\frac{t}{h} - i\right)^3, & t_i \le t \le t_{i+1} \\ \frac{2}{3} - \frac{1}{2}\left(\frac{t}{h} - i - 2\right)^2 \left(\frac{t}{h} - i\right), & t_{i+1} \le t \le t_{i+2} \\ \frac{2}{3} - \frac{1}{2}\left(i + 2 - \frac{t}{h}\right)^2 \left(i + 4 - \frac{t}{h}\right), & t_{i+2} \le t \le t_{i+3} \\ \frac{1}{6}\left(i + 4 - \frac{t}{h}\right)^3, & t_{i+3} \le t \le t_{i+4} \end{cases}$$

  The function $S_{i,4}(t)$ is plotted in blue in the figure.

One final comment about the spline functions is that $S_{i,k}(t)$ is symmetric around the value $t/h = i + k/2$. This implies that their formulas become simpler if we shift the t variable by $hk/2$, so that they become symmetric around $t/h = i$. Also, for this reason it is a bit better to take k to be even: this makes the maximum of the function $S_{i,k}(t)$ coincide with one of the knot values, namely $t_{i+k/2}$.

From now on we are going to do this shift and also make the change of variable $u = (t - t_i)/h$ (the new variable u goes from 0 to 1 in each subinterval). With this the formulas for $S_{i,4}(t)$ become:

$$S_{i,4}(u) = \begin{cases} \frac{1}{6}(2 - |u|)^3, & 1 \le |u| \le 2, \\ \frac{2}{3} - \frac{1}{2}u^2(2 - |u|), & 0 \le |u| \le 1. \end{cases}$$

Equation 1: The B-spline function of order 4 (degree 3 and second-order differentiable)

This is the form spline function that we use in what follows and in the rest of the work. The following figure shows this function:

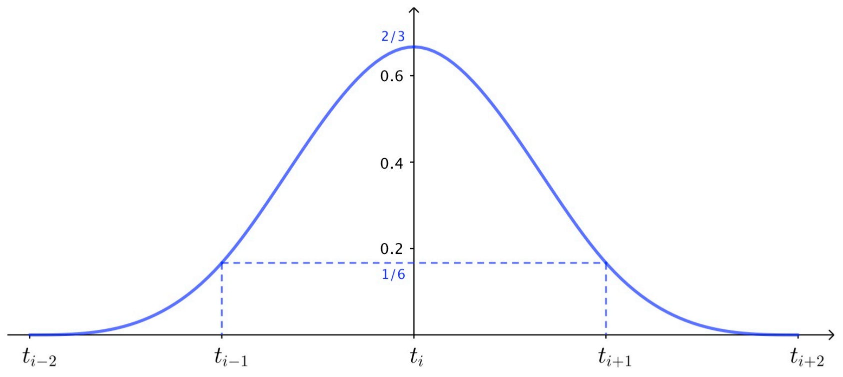


Figure 9: The B-spline function $S_{i,4}(t)$ after shifting by $2h$. Its symmetry is at $t = t_i$ instead of $t = t_{i+2}$.

B-Spline curves

In order to construct a B-spline curve, besides the knot vector $t_1 \le t_2 \le \cdots \le t_n$ we use the points $p_1, p_2, \ldots, p_n$ associated to them. The spline of order k associated to this knot vector and these control points is defined as the parametric curve

$$S(t) = \sum_1^n S_{i,k}(t)\, p_i.$$

By the properties of the B-spline functions, S is a piecewise polynomial function of degree $k-1$ and it is C^k-continuous. That is, in our case $k = 4$ we are constructing a C^2-continuous cubic spline.

Since the functions $S_{i,4}$ are always ≥ 0 and their sum equals 1, for each value of t the point $S(t)$ is a barycenter of the control points, where each point p_i is considered to have relative weight equal to $S_{i,4}(t)$. As t varies, the relative weight of different points varies.

Also, since each $S_{i,4}(t)$ is nonzero only in the interval $[t_{i-2}, t_{i+2}]$, for each value of t at most four points get positive weight. More precisely, if $t \in [t_i, t_{i+1}]$ the points with positive weight are $p_{i-1}, p_i,$ p_{i+1} and p_{i+2} and their respective weights are the following, written in the same variable $u = (t - t_i)/h$ that we used above:

$$S_{i-1,,4}(u) = \frac{1}{6}(1-u)^3 = \frac{1}{6}(1 - 3u + 3u^2 - u^3) \ ,$$

$$S_{i,4}(u) = \frac{2}{3} - \frac{1}{2}u^2(2-u) = \frac{1}{6}(4 - 6u^2 + 3u^3),$$

$$S_{i+1,4}(u) = \frac{2}{3} - \frac{1}{2}(1-u)^2(1+u) = \frac{1}{6}(1 + 3u + 3u^2 - 3u^3),$$

$$S_{i+2,4}(u) = \frac{1}{6}u^3.$$

Equation 2: The B-spline function $S_{i,4}(t)$. These are the same formulas as in *Equation 1*, except instead of writing the four polynomial pieces for a fixed i we are writing the four that are nonzero in a fixed subinterval $[t_i, t_{i+1}]$. These are the formulas implemented in our code.

Sometimes Equation 2 is written in matrix form as follows (Hamilton, 2019):

$$S(t) = \frac{1}{6}(u^3, u^2, u, 1)\begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{pmatrix}\begin{pmatrix} p_{i-1} \\ p_i \\ p_{i+1} \\ p_{i+2} \end{pmatrix}, \qquad 0 \le u \le 1,$$

Plugging in $u = 0$ we see that at the knot values only three of the coefficients are nonzero. Indeed, we have:

$$S(t_i) = S_{i-1,4}(t_i)p_{i-1} + S_{i,4}(t_i)p_i + S_{i+1,4}(t_i)p_{i+1} = \frac{1}{6}p_{i-1} + \frac{2}{3}p_i + \frac{1}{6}p_{i+1}.$$

That is, $S(t_i)$ lies in the triangle $p_{i-1}p_ip_{i+1}$ and much closer to p_i than to p_{i-1} and p_{i+1}. This guarantees the curve $S(t_i)$ approximates well the original curve that we want to reconstruct.

Chapter 3 Methods and implementation details

3.1 *Starting point*

Our starting data is as follows: 16 bvh files, split into 8 couples recorded together. Each couple consists of the motion recorded from the operator generating the desired movement, and the motion of the UR5 robot performing the same movement. Since our goal is to obtain a motion path from the movement of the human, the files containing the robot movement will only be used for comparison purposes.

The performer was standing in front of the robot and replicating with his/her right hand the movement of the robot's tool (but not trying to mimic the whole arm's movement, only the position of the tool). The robot was programmed to perform a simple periodic motion, and the motion of the robot and the performer were captured simultaneously in two different *.bvh files, at a rate of 60 frames per second. This was done eight times, half of them with a circular motion and the other half with a square motion:

Files	Type	No. of frames	Time (secs.)	Time (mins.)
R11.bvh; H11.bvh	Circle	20571	342.85	slightly below 6
R17.bvh; H17.bvh	Circle	39001	650.02	almost 11
R20.bvh; H20.bvh	Square	34991	583.18	slightly below 10
R21.bvh; H21.bvh	Square	25141; 33251	419.01; 554.18	slightly below 10
R23.bvh; H23.bvh	Circle	31411	523.52	below 9
R24.bvh; H24.bvh	Circle	33471	557.85	slightly above 9
R25.bvh; H25.bvh	Square	50221	837.02	about 14
R26.bvh; H26.bvh	Square	42441	707.35	slightly below 12

*Table 1: The eight plus eight input files containing motion captured from the robot (F**.bvh) and human (H**.bvh). Observe that in one of the motions (files R21.bvh; H21.bvh) the number of frames, hence the total time, captured for the robot and the human are different*

As seen in the Table, one of the motions (number 21) has a significantly different number of frames stored for the human and the robot. We do not know the reason for this, but it is not really an obstacle for our work, since we are interested in reconstructing a single cycle of both, and each cycle has about 300 or 400 frames (5 to 7 seconds).

Each of the files (both for the robot and the human) contains data for several joints, but we are only going to use one of them: the tool of the robot and the hand of the human.

All files are in the Biovision Hierarchy (*.bvh) format. The equipment used to record the motion were HTC Vive trackers.

The settings for the recording were different too. Files H11.bvh and H20.bvh were recorded using forward kinematics. Files H17.bvh and H21.bvh were recorded using inverse kinematics. Motions H23.bvh to H26.bvh were recorded with a single tracker at the joint that interested us (the hand of the operator), with no kinematics.

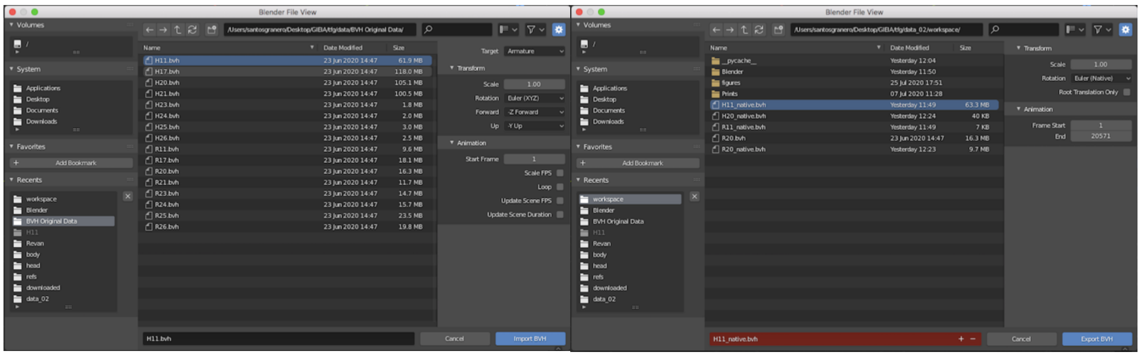
3.2 *Preprocessing the data with Blender and bvhplot*

Since all of our original data is saved in BVH files, we need functions that read them and extract the data we want to use, to give it to the python procedures. At the beginning of the project we were given code that does this, namely the script *bvhplot.py* developed by the research group (Copyright Martin Manns).

This script generates an object of a custom class "BVHMotion" which includes all of the information from the bvh file. In particular, the data for each individual joint and frame can be extracted via the function *get_cartesian_joint_frame*. From there we create a new class named "BVHToMatrix", which requires an object of class "BVHMotion" and the name of the joint that we are interested. The motion of this joint is put in an attribute called "cartesian_frame" and is the main data used by most of the functions in this class. It is a three-column matrix with one dimension for each coordinate axis, and one row for each frame recorded in the original file. Since this part of the computation is the one that takes the most time (one minute versus one or two seconds combined for the other parts) we decided to store the matrices obtained as separate files, so that in the rest of the work we directly use those matrices as input, in order to avoid having to recompute them. The eight plus eight files containing these matrices are called *R**_pickled or H**_pickled,* where ** denotes the numbered label of the motion. We also have a separate script file *BVHtoMatrices.py* to perform this task, separate from the script *main.py* that does all the rest.

We do, however, come across one problem; the *bvhplot* package assumes that the data is provided in a format with a native Euler angle order, while the functions in *bvhplot* use a Cartesian format. To solve this, we could have modified the code in *bvhplot* in order to transform the data into a native Euler angle. However, we have considered that a simpler approach is to use the 3d software Blender to solve the issue for us. Blender can import and export motion data in BVH files. What we do is we import the original bvh data into Blender specifying what type of rotation we use. By default, it is set to "native", so we change it to "xyz". This opens two new fields where we can specify which axis and direction is considered "forward" and "up" in the dataset we are importing. We set the "up" option to "-Y", as that is how the axis were set up when the data was originally measured. The "forward" option is less important. By setting it to "-Z" we say that the operator is further along the Z axis than the robot, looking in the direction of -Z.

Afterwards we export the motion keeping the "rotation" tab as "native", so that we get a new bvh file with data expressing exactly the same original motion, but with respect to the angle orders needed by *bvhplot*. WE keep the original filenames, adding "native" to them (e.g., file *R11.bvh* becomes *R11_native.bvh*)


Blender File View

/Users/santosgranero/Desktop/GIBA/tfg/data/BVH Original Data/

Name	Date Modified	Size
H11.bvh	23 Jun 2020 14:47	61.9 MB
H17.bvh	23 Jun 2020 14:47	118.0 MB
H20.bvh	23 Jun 2020 14:47	105.1 MB
H21.bvh	23 Jun 2020 14:47	100.5 MB
H23.bvh	23 Jun 2020 14:47	1.8 MB
H24.bvh	23 Jun 2020 14:47	2.0 MB
H25.bvh	23 Jun 2020 14:47	3.0 MB
H26.bvh	23 Jun 2020 14:47	2.5 MB
R11.bvh	23 Jun 2020 14:47	9.6 MB
R17.bvh	23 Jun 2020 14:47	18.1 MB
R20.bvh	23 Jun 2020 14:47	16.3 MB
R21.bvh	23 Jun 2020 14:47	11.7 MB
R23.bvh	23 Jun 2020 14:47	14.7 MB
R24.bvh	23 Jun 2020 14:47	15.7 MB
R25.bvh	23 Jun 2020 14:47	23.5 MB
R26.bvh	23 Jun 2020 14:47	19.8 MB

Target: Armature
Transform
Scale: 1.00
Rotation: Euler (XYZ)
Forward: -Z Forward
Up: -Y Up
Animation
Start Frame: 1
Scale FPS
Loop
Update Scene FPS
Update Scene Duration

H11.bvh

Blender File View

/Users/santosgranero/Desktop/GIBA/tfg/data_02/workspace/

Name	Date Modified	Size
__pycache__	Yesterday 12:04	
Blender	Yesterday 11:50	
figures	25 Jul 2020 17:51	
Prints	07 Jul 2020 11:28	
H11_native.bvh	Yesterday 11:49	63.3 MB
H20_native.bvh	Yesterday 12:24	40 KB
R11_native.bvh	Yesterday 11:49	7 KB
R20.bvh	23 Jun 2020 14:47	16.3 MB
R20_native.bvh	Yesterday 12:23	9.7 MB

Transform
Scale: 1.00
Rotation: Euler (Native)
Root Translation Only
Animation
Frame Start: 1
End: 20571

H11_native.bvh

Figure 10 Import and export screen for bvh files in Blender. Notice inputs for rotation on both, and direction on the "import" screen.

This way of changing the rotation angle through Blender has advantages and disadvantages versus doing it through code. Since it is a single task that blender does for us with just a few clicks, we decided that it was faster overall to do it manually over the 12 files that we had versus writing code to modify the way *bvhplot* handles the bvh files. If we had to do this for a large number of files it would be worth it to spend extra time coding, as the import and export process in Blender is much slower than running the code would be. As an added bonus, importing the motion into Blender also allowed us to visualize the whole human and robot it in the form of a “skeleton”, as seen in Figure 11 and Figure 12.

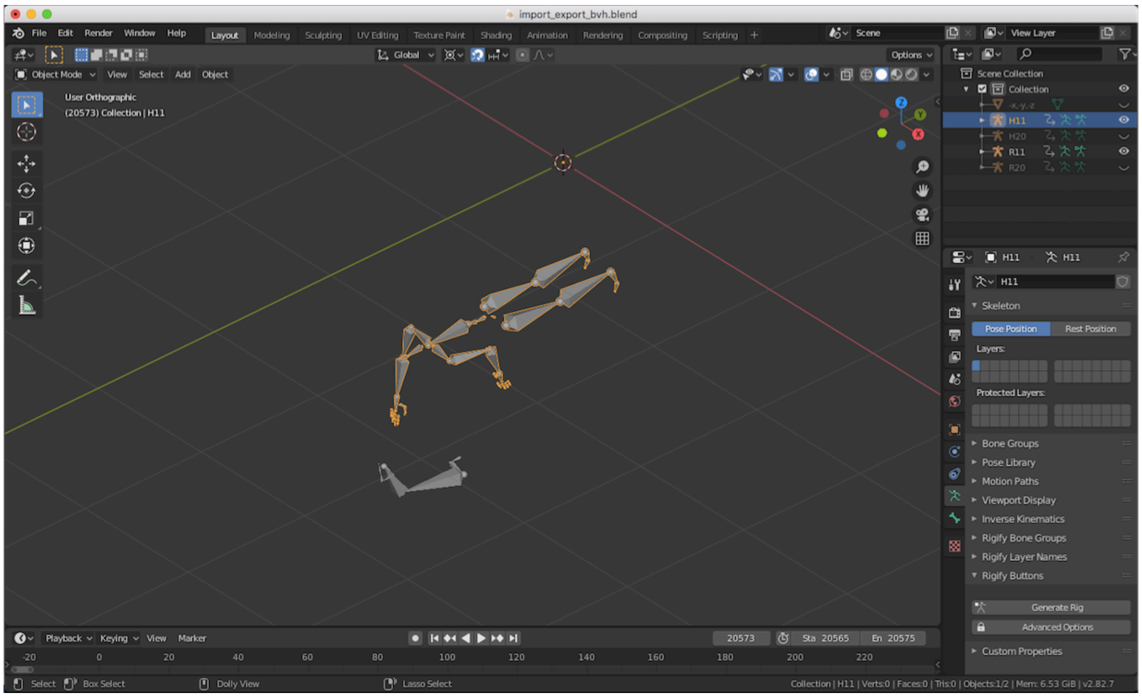


Figure 11: Blender viewport, showing the imported files H11.bvh (highlighted) and R11.bvh. Notice that the figures are sideways, as Blender utilizes Z as the vertical axis, while our data used Y.

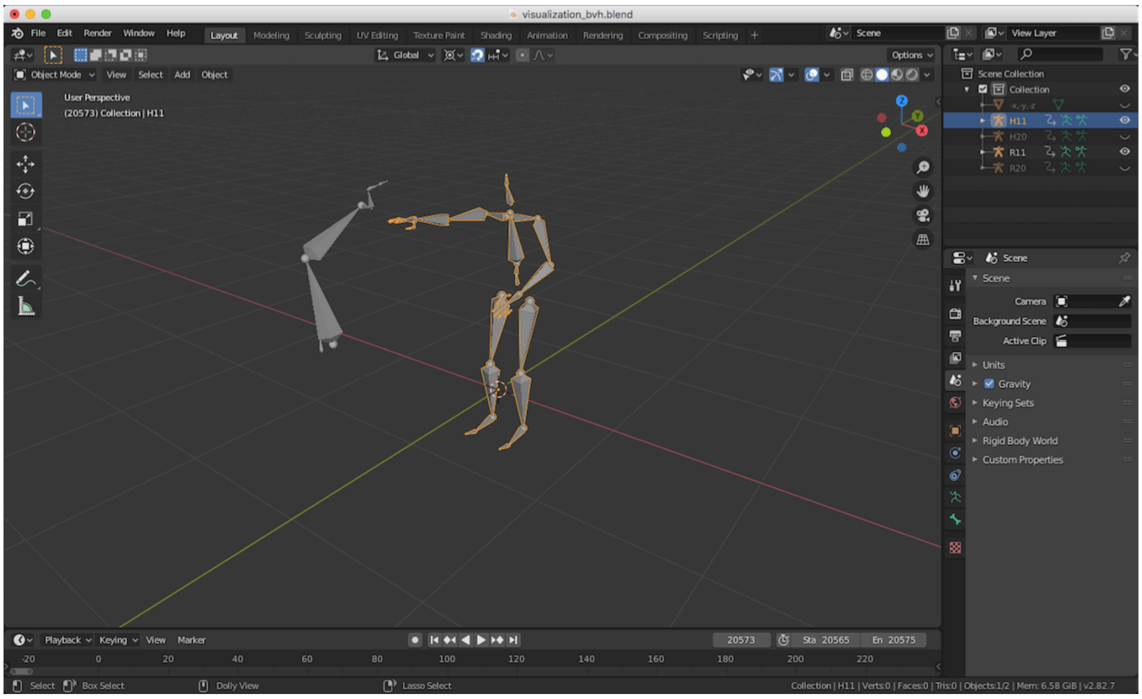


Figure 12: To properly visualize the skeletons from the previous figure, we can simply rotate their object in Blender.

3.3 Projection via PCA

From here on, *all the functions and scripts mentioned have been completely written by us* (except, of course, for the use of different python packages). Also, we stop using the "BVHMotion" functions and even "BVHToMatrix" class that we created and use our custom functions exclusively.

All scripts are written in a way that they are not specific for a particular motion file. Instead, all the functions are generic, meaning that they can be used with all 16 of the files that concern us and in theory with any file containing an $n \times 3$ matrix that represents a periodic motion (the motion is assumed to be approximately planar, since one of the things we do is to planarize it). We have, however, manually cropped one of the input files in order to remove a part of the motion that was not part of the measurement. (See details in Chapter 4).

The main part of our code defines the functions.

The first thing that we do is to use Principal Component Analysis to project our input motion to the "best fitting plane". As explained in section **Error! Reference source not found.**a PCA transformation can simplify the data we work with significantly by projecting it to lower dimension but trying to keep the most significant features. In our case this becomes very useful, as we know beforehand that the motions that we will be studying were supposed to be flat, so any deviation from a flat motion was an error in the first place.

To apply the transformation, rather than implementing PCA ourselves (which amounts to computing a covariance matrix and doing some linear algebra to it) we use the PCA tools in the ready-made Python module SciKit-learn (Pedregosa, et al., 2011).

In our script all the PCA computation is contained in a single function *pca_from_array*(self, frames) which has an input a set of frames (that is, labels for some rows of our $n \times 3$ matrix of points) and gives as output the three eigenvectors and eigenvalues of the covariance matrix, together with the matrix of points projected to the plane of the biggest two eigenvalues. (In a first version of our script the function returned only the first two eigenvectors and eigenvalues, but we think that the third one is also very significant:

- The third eigenvalue allows us to verify that the input was indeed "close to 2-dimensional", since this is equivalent to this eigenvalue being significantly smaller than the other two. In fact, this is one of the reasons why we decided to crop the motion from the file H25: performing the PCA to the full motion gives three similar eigenvalues, which makes the PCA projection to two dimensions be useless.
- The third eigenvector is the normal vector of the projection plane, and having it allows us to measure how close the fitted plane is to being parallel to the plane fitted for the robot.

3.4 Computation of periods

The next thing we want to do is to compute the period and number of cycles of each motion. For this we have two functions:

aprox_frame_select:

Its input is a matrix *curve* whose rows represent the points of a (periodic) motion, a base frame *f* (the label for one of the rows), and a length *d* to be considered a threshold radius. The function first computes all the frames that fit in a ball of radius *d* around the base frame, and considers each interval of them (that is, each set of consecutive frames in the ball) part of a different individual cycle. The function then returns one point (the middle one, rounded up or down) from each of these intervals. The number of them approximates the number of cycles in the whole motion.

In order to detect intervals among the frames that fit in the ball we go through the list of these frames; every time two consecutive selected frames are not consecutive in the original motion, the first one is the end of an interval and the second one is the beginning of the next.

_find_period_:

This function calls the previous one and takes as a first, approximate, average period the total number of frames divided by the number of cycles detected there. We call this *temp_av_period*, defined as

$$T = \frac{f_{n-1} - f_0}{n}$$

where *n* is the number of frames output by *aprox_frame_select* and f_0 and f_{n-1} are the first and last of them, respectively.

The main error that this might result in is an issue where the selection of points missed a cycle because the motion in that cycle deviates too much from the normal path and did not go through the area around the target point for “aprox_frame_select”. This would result in a gap in the list of frames which should be twice the period. Another source of error is that sometimes the motion may go slightly out from the ball and back again in the same cycle, so that we will get two (or more) intervals corresponding to the same cycle. But this will result in a gap that is much smaller than the period.

In order to correct these two effects, we compute the lapses between every two consecutive selected frames in the list. That is, the difference $f_i - f_{i-1}$ for each i. We discard those that deviate more then 25% from the “temporary period” and take the average of the remaining ones as the true period. This way we get rid of the time measurements that are too long because of a missed cycle and those that are too short because of a broken cycle.

As base frame we use by default the middle one in the motion, and as *d* we use 0.1. Observe that *d* needs to be at least in the order of the average deviation of the human from the intended trajectory, and much smaller than the diameter of the trajectory. Our motions have diameter close to 0.4, so 0.1 seems appropriate. At any rate, the computation of the period is not very dependent on the choice of *d*, although some of the things that we do later are. Both parameters (base frame and radius) can be input by the user.

3.5 Selection of sample points for average cycle

The task now is as follows. We think of the motion we are given as periodic (with the period computed in the previous section) and want to compute a certain number of positions of the point for equally spaced points along the period, taking the average of the positions of the object at that same moment in the different cycles.

If the motion was exactly periodic and we knew the exact period, the solution would be simple. Let T be the period (in frames) and let n be the number of points that we want to compute in a period. Then, the frames corresponding to these points in the first period are

$$floor\left(\frac{iT}{n}\right), \quad i = 0, \dots, n-1,$$

and those of the other periods are obtained adding multiples of p to that. So, for each given i the point we are looking for is

$$p_i = average\left(point(floor\left(\tfrac{iT}{n}\right) + kT), \ for\ all\ k\right),$$

Equation 3

Where “for all k” means for all the integer values of k that make the result lie within our total range of frames. This gives us the desired points $p_{1,} \dots, p_n$.

However, doing this the points we get turn out not to be “very good” when compared to the actual trajectory. We have solved this in two ways, in two different functions. The first one works reasonably well for circular motion but not for square ones, the second one works well in both cases and is the one we take for the subsequent steps:

<u>*frame_select*</u>:

We apply exactly the formula in Equation 4, except we first remove outliers, using for this the function *_remove_outliers*. That is, after computing the average, we discard the individual frames $floor\left(\frac{iT}{n}\right) + kT$ for which the position is “too far” from the average (we take by default a circle of radius $d = 0.05$) and recompute the average without the outliers.

The problem with this method is (we believe) that if the period we have computed is not very good, the error in the period propagates along the different cycles so that the points whose average we compute in Equation 3 do not correspond to the same phase along a period. In fact, the points may end up being almost uniformly distributed along the period, in different cycles. The removal of outliers corrects this because it makes the average be taken by points that are close to one another in space, hence also (presumably) in phase, but the drawback is that we are potentially discarding too many cycles. In order to avoid this, we implemented a second approach:

<u>*create_cycle_sample*</u>:

The idea is the same, except we change the part “$kT\ for\ all\ k$" in Equation 3 to a list of frames that should truly correspond to points in the same phase. These points are simply the same list of points that we used in *aprox_frame_select*: the mid-points of the segments of curve inside a certain ball centered at a base point chosen as reference. Let us call L this list of points (or, rather, of labels for frames). Then, for each i we compute

$$p_i = average\left(point\left(floor\left(\frac{iT}{n}\right) + l\right),\ for\ all\ l\ in\ L\right),$$

Equation 4

The effect of this is that even if our value of p is approximate, the error we are committing in the part iT/n is never bigger then the original error in T, and it does not propagate from one cycle to the next.

3.6 Reconstructing trajectory via B-splines

Once we have the points $p_1, \dots, p_n$ along an “average cycle”, we compute a closed (or periodic) spline curve from them by simply using Equation 2. We have created two functions for this, the main one and an auxiliary one:

<u>*create_spline_points*</u>: This has as input the period T, a frame t from the motion, and the cycle points $p_1, \dots, p_n$ computed in the previous section. The function computes the parameters i and u that we need to plug in Equation 2 and returns the value of the spline curve S for those values. Remember that i is the integer such that t lies in $[\frac{iT}{n}, \frac{(i+1)T}{n})$ (that is, $i = floor(\frac{T}{n})$) but reduced modulo n so the $0 \leq i < n$, and u is the relative position of t along that interval, that is, $u = \frac{nt}{T} - i$, and goes from 0 to 1 along the interval.

<u>*create_spline*</u>: it has the same input except for t, and calls <u>*create_spline_points* for</u> $t = 0, \dots, T-2$. We chose $T-2$ rather than $T-1$ as the last point because with $T-1$ the curve obtained seemed to have “defects” at the end of the cycle. We attribute that to the fact that if the value of T is overestimated, taking $T-1$ gives us that the initial point and the final point are too close (or, even worse, $T-1$ may be already part of the next cycle) making the spline curve turn too much (or even backwards). The only danger of using

$T-2$ wuold be to have one less sample point in the cycle, but the smoothness of the spline curve makes that not be too serious.

3.7 Comparison with original robot motion

In order to evaluate the accuracy of our model, we use the motion of the robot as a point of comparison. All the computations so far use only the data in the human captured motion. Once we have the reconstructed spline we want to compare it with the original motion. For this we do several tests and measures:

- **Projection direction**: the third eigenvalue obtained in the PCA computations is orthogonal to the first two, that span the projection plane, so that it equals the normal vector to the plane that PCA selects as best fit (the plane to which we project). One first measurement that we perform PCA for the robot and compare that normal direction to the direction that we had obtained for the human. The result is that it ranges between 0.7 and 4 degrees, except in motion number 20 where it is larger than 10 degrees.
  Observe that tis measurement does not tell us anything about the reconstruction. It is much more related to how good was the human at imitating the robot. Yet, we think of this angle as an interesting parameter.
- **Period**: We apply to the motion the same period computation that we applied to the human and compare the computed periods. In all cases the discrepancy between the two is in the order of 1% or less, except in motion 21 where it is 2.5% (see section 4.6).
- **Pointwise distance**: We translate the reconstructed spline so that it has the same barycenter as the original robot motion, and compute how far is each point in the reconstructed motion (that is, each of the 300-400 points in the spline curve that correspond to frames in a period) from the robot motion. The distance to the robot motion is computed as the minimum to all the frames in the original motion, without projecting it via PCA.
  This gives is a vector of 300-400 distances, and we then give as data the maximum among them (that is, the largest distance between a pint in the reconstruction and the original curve) and the average (the average distance between points in the reconstructed curve and the original motion).
  In the first four motions the error is relatively big (between 4 and 7 cm, which is about 10 to 20% of the side of the square or diameter of circle that the robot describes). In the four last motions the error is much smaller: the average is in the order of 1 cm or less, the maximum never goes above 1 cm.

3.8 Real use-case: Importing into RoboDK

When using this method to obtain a real program to be used by a robot, we will now need to export our curve to a robot programming tool. First, we save the matrix as a *comma separated value* (CSV) text file, and to that end we use the package *pandas,* which allows us to work with data frame variables.

```
import pandas as pd

from pandas import DataFrame

df = DataFrame(reconstruction) # Name of variable to export

export_csv = df.to_csv (r" save location and file name ")
```

For the explanation, we will use as an example the simulation and programming software *RoboDK* (RoboDK INC, s.f.). Once we have a CSV file, we can import it directly into the program by clicking the *Utilities* tab and selecting *Import Curve*. We then simply select the file with the points we want to import. If it were necessary, this would then modify the points in RoboDK.

Alternatively, we could import the matrix with the sample points only and then perform the interpolation directly in RoboDK.

Chapter 4 Experimental results and discussion

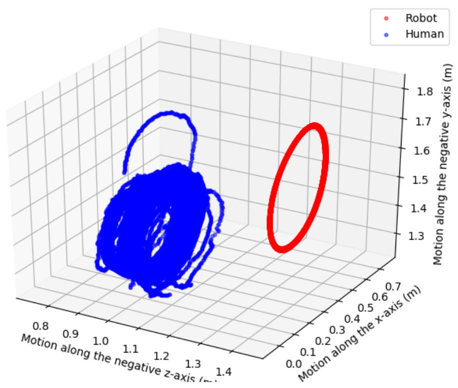
We here discuss the results obtained at each stage of the process for the different input motions.

4.1 Preprocessing the data with Blender and bvhplot

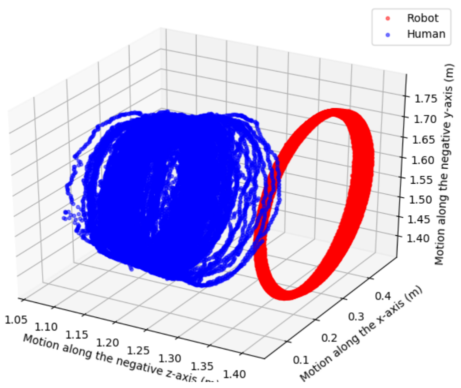
As mentioned in Section 3.2 we here do three things:

- Modify the *.bvh files via the application *blender* so that they are rewritten with respect to native Euler angles.
- Use the package *bvhplot* to extract from each bvh file a matrix that contains the data for the joint we are interested in. For the robot the joint we want is always *tool0*. For the human we are interested in *hand_r* in motions 11, 17, 20 and 21, and we want *Tracker_Robot* for motions 23, 24, 25 and 26. Our script selects the appropriate joint automatically, based on the number provided.
- Save the extracted motion for a single tool in a file using *pickle*.

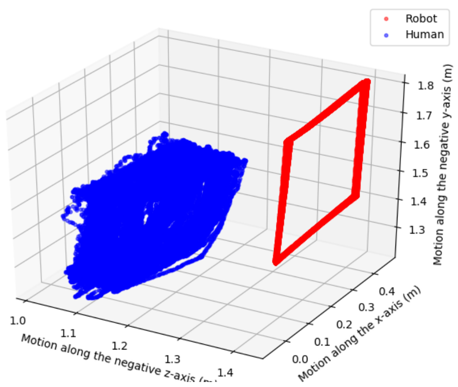
However, there is a fourth thing that we have done, which is to manually "amend the input". To see what this means and why it is needed, let us look at plots for the motions that we have obtained. In the following eight figures, the human captured motion data is plotted in blue and that of the robot is in red:



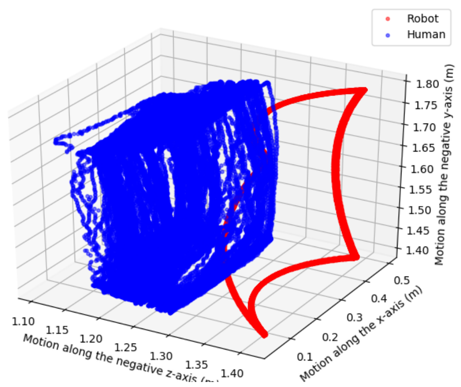
Motion number 11



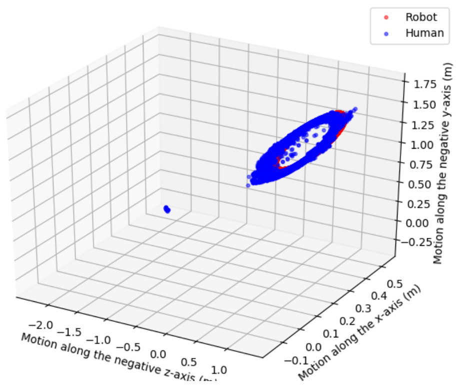
Motion number 17



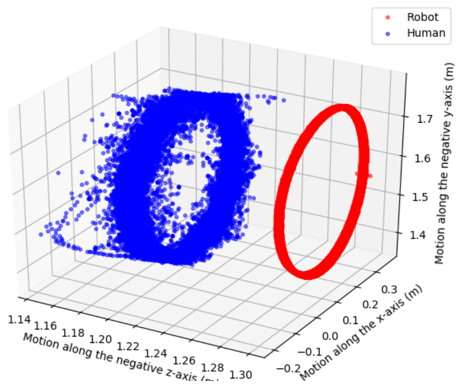
Motion number 20



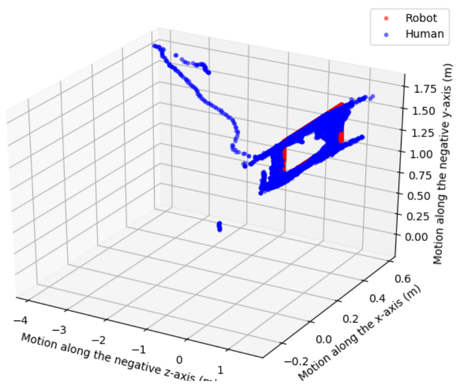
Motion number 21



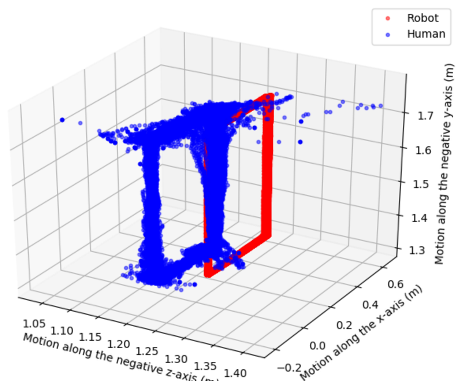
Motion number 23 (*)



Motion number 24



Motion number 25 (*)



Motion number 26

Figure 13: The original captured motions of the human (blue) and robot (red). Motions 23 and 25, marked with a (*) have defects that cannot be explained by human error. Some points in the blue pictures appear 4 meters away from where they should be. We decided to manually correct these errors, since otherwise the input files would be unusable.

In most of the pictures we see what one should expect: the robot's motion is very precise, deviating only a couple of centimeters (the thickness of the red line) from the intended trajectory. This error is far higher than the repeatability of the UR5 as stated by the manufacturer (Universal Robots A/S, s.f.), which is 0.1mm. Therefore, we suppose this error is caused by the motion capture equipment.

The human motion is much less precise and deviates some 10 cm, sometimes even 20 cm from its trajectory. But in motions 23 and 25 we see something much more drastic: in both of them we see "a few" isolated blue points about 3 or 4 meters away from where they should be (far left in motion 23, bottom in motion 25); additionally, in motion 25 we see that at some point the human motion starts "drifting away" from its path, going about four meters away from it in the positive direction of the coordinate z (the motion is supposed to stay close to z=-1, and it goes up to z=4).

Looking at the actual matrices, we see where these "defects" lie in the motion:

- In motion 23 (31411 frames in total), frames 23136 to 23155 are deviated about 4 meters. Inspecting the data, we also see that right prior to that, frames 23031 to 23135 are "stalled": the position for those frames is exactly the same. The following table contains an excerpt of the data:

FRAME	X	Y	Z
23028	-0.059859	-1.49371	-1.20373
23029	-0.051512	-1.485	-1.20649
23030	-0.055407	-1.48462	-1.20597
23031	-0.040755	-1.47521	-1.20731
23032	-0.040755	-1.47521	-1.20731
23133	-0.040755	-1.47521	-1.20731
23134	-0.040755	-1.47521	-1.20731
23135	-0.040755	-1.47521	-1.20731
23136	0.5299	0.424417	2.25271
23137	0.534392	0.418654	2.25969
23138	0.53673	0.420973	2.26052
23139	0.531342	0.416824	2.26812
23140	0.532757	0.413798	2.27268
23141	0.530574	0.412762	2.26726
23142	0.53042	0.409357	2.27089
23143	0.529582	0.417824	2.26789
23144	0.532147	0.405995	2.29051
23145	0.530298	0.405832	2.28708
23146	0.52683	0.394684	2.2846
23147	0.52693	0.392054	2.2766
23148	0.525944	0.385529	2.27579
23149	0.524489	0.380759	2.26647
23150	0.525333	0.377496	2.27479
23151	0.524908	0.375356	2.2767
23152	0.524908	0.375356	2.2767
23153	0.524908	0.375356	2.2767
23154	0.524908	0.375356	2.2767

23155	0.524908	0.375356	2.2767
23156	0.316706	-1.51477	-1.20688
23157	0.31707	-1.51761	-1.20606
23158	0.314487	-1.52293	-1.20649

Table 2: The defects in the input human motion number 23

- In motion 25 (50221 frames in total), we see two things. First, in frames 45750 to 45980 we see the same as in motion 23: a sequence of exactly 105 frames (frames 45856 to 45960) with the same position, followed by 20 frames (45961 to 45980) where the fames are deviated some four meters from where they should be. This accounts for the “blue points in the bottom” that we see in the figure. But we also see that starting in frame 6125 approximately, the motion starts drifting away from where it should be at a speed of some 5 to 10 cm per frame (which would correspond to an actual speed of about 3 to 6 m/s, or 10 to 20 km/h). This happens up to frame 6262, after which the motion goes back to where it should be.

We are not sure how interpret these defects, but they are clearly erroring in the motion capturing. At first, we thought this could be caused by the operator not following the movement at the beginning or end of the motion, so eliminating the first and last frames in the files would resolve it. However, the errors appear near the middle of the frames in the files, so we believe them to be caused by an interruption in the recording or an error of another kind. We could have written a script to automatically remove “outliers” from the motions, but we believe that would be contrary to the spirit of this work. We want our scripts to work for generic inputs, and we cannot pretend to be able to detect and correct all possible errors in the input. Writing scripts to correct these particular errors after inspecting them is not better than correcting them manually, so we have simply corrected them manually, as follows:

- For the two cases where the error starts with 105 equal frames followed by 20 obviously wrong frames we have simply made the latter 20 frames equal to the first 105. The rationale behind this is that the first 105 frames are already an error in the input, but an error that should not affect our analysis much and is not much worse than the errors introduced by the human (for example, in motion 20 we see that the human took a “shortcut” through the diagonal of the square in one of the cycles, in motion 26 we see that the human went some 20 cm too far along one of the sides of the square, etc). Besides, there might be other parts where the system got stalled and gave repeated frames without us detecting it, so that this phenomenon can perhaps be considered part of the process.
- For the case where the motion drifted away, (frames 6125 to 6262 in H25) the error seems to be only in the z coordinate, which was supposed to stay constant and close to -1.20. So, we made that coordinate equal to -1.20 for those frames.

The results of these modifications were stored in two additional *pickle* files, called *H23B_pickled* and *H25B_pickled*. More precisely, for file H25 we do

```
for i in range(45961,45981):
    originalH[i]=originalH[45960]
for i in range(6125,6263):
    originalH[i,2]=originalH[6125,2]
```

and in file H23 we do

```
for i in range(23136,23156):
      originalH[i]=originalH[23135]
```

After these modifications we get the motions of Figure 14:

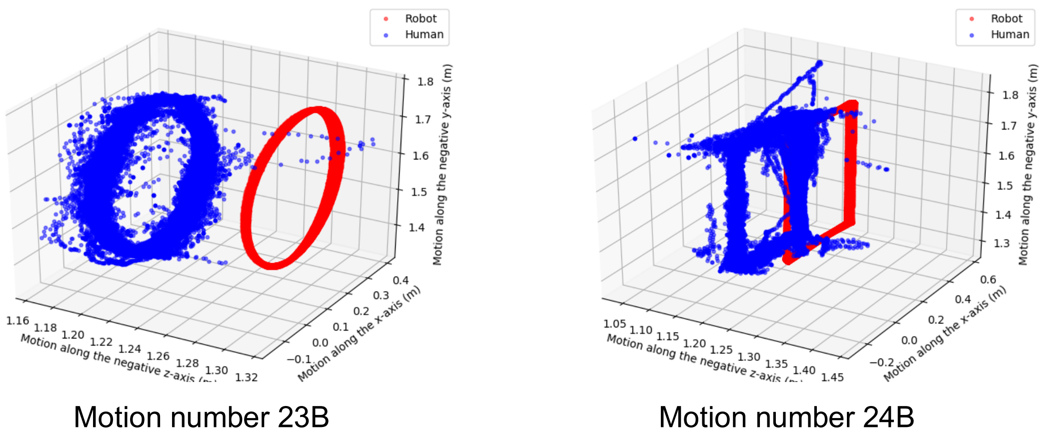


Figure 14: The motions 23 and 25 after manual correction of the human motion.

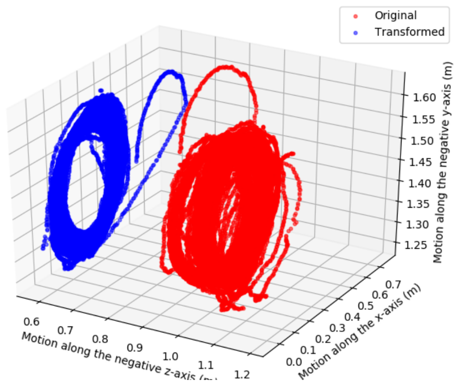
4.2 Projection via PCA

Once we have our input data in the form we want it, our next task is to perform PCA to it and project it to the first two principal components. The following table shows the three eigenvectors for each motion. As can be seen, the third one is always significantly smaller than the first two. This indicates that the motion is indeed “close to two-dimensional”. In fact, the third eigenvalue is a good initial measure of how much error the human committed throughout the motion.

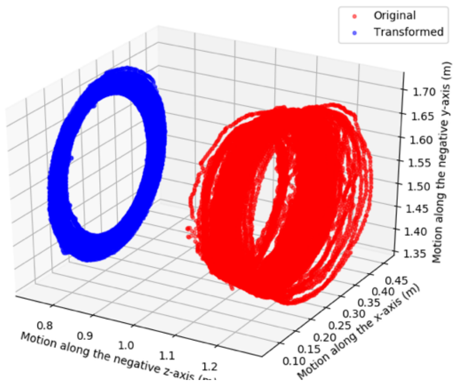
motion	1st eigenvalue	2nd eigenvalue	3rd eigenvalue
11	0.52214654	0.42014255	0.05771091
17	0.49374691	0.47549916	0.03075393
20	0.5476785	0.41687136	0.03545014
21	0.49846176	0.47161215	0.02992609
23B	0.5077688	0.49148385	0.00074734
24	0.5050543	0.49427463	0.00067107
25B	0.51167182	0.48715022	0.00117796
26	0.50766904	0.49136263	0.00096834

Table 3: The three eigenvalues obtained from PCA, for the eight motions. Observe that the first two are always similar (with difference between 25% and 2% depending on the motion) but the third one is always significantly much lower, as corresponds to a motion that is almost planar.

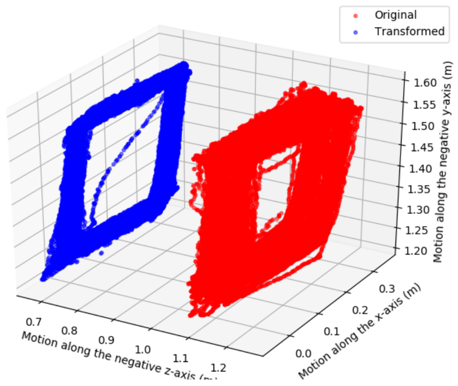
If we want to translate the data in Table 3 to deviation lengths we need to take two things into account: the first one is that the values correspond to variances, so we need to take the square root of them in order to get standard deviations. The second one is that the PCA package that we are using gives the output in normalized form: the three variances are normalized to have sum equal to one. For example, in motion 20 (the one with the largest third eigenvalue), the square root of the variances gives 0.74, 0.65, and 0.19. This means that the standard deviation in the third component is about 20-30% of the deviation in the first two. Since the motion is a square of side about 40cm, its deviation in the first two components should be in the order of 20 cm, which means the deviation in the third component is in the order of 4 to 6 cm.



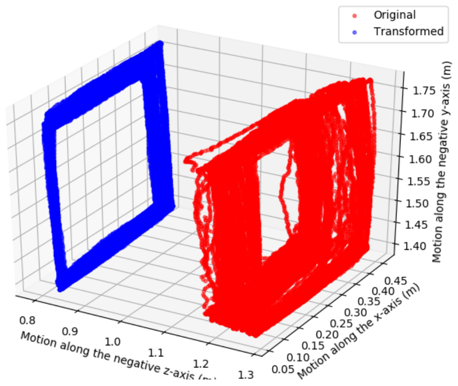
Motion number 11



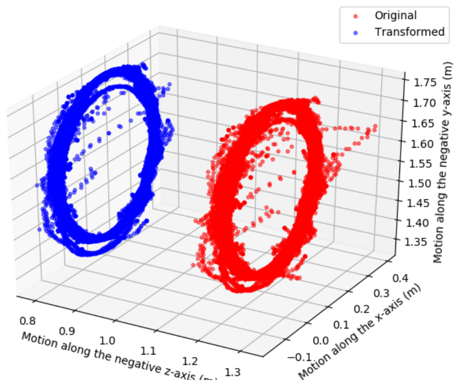
Motion number 17



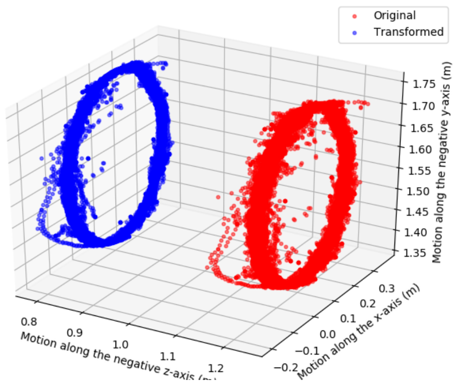
Motion number 20



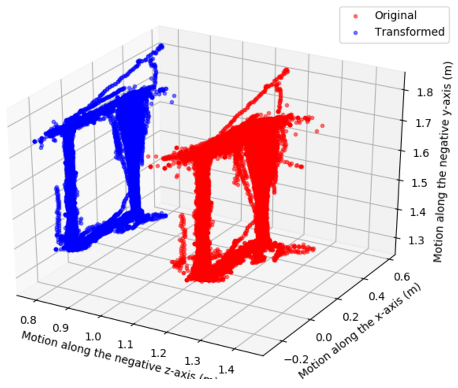
Motion number 21



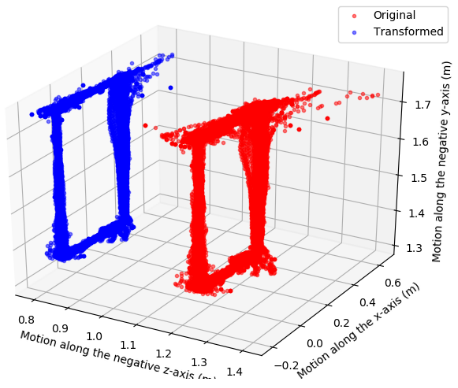
Motion number 23B (*)



Motion number 24



Motion number 25B (*)



Motion number 26

Figure 15: The original captured motion of the human (red) and the planarized version after applying the PCA transformation. In the plot, the transformed version has been translated 40 centimeters along the Z axis for clarity. In 23 and 25 the manually corrected motions have been used. Compare to Figure 16.

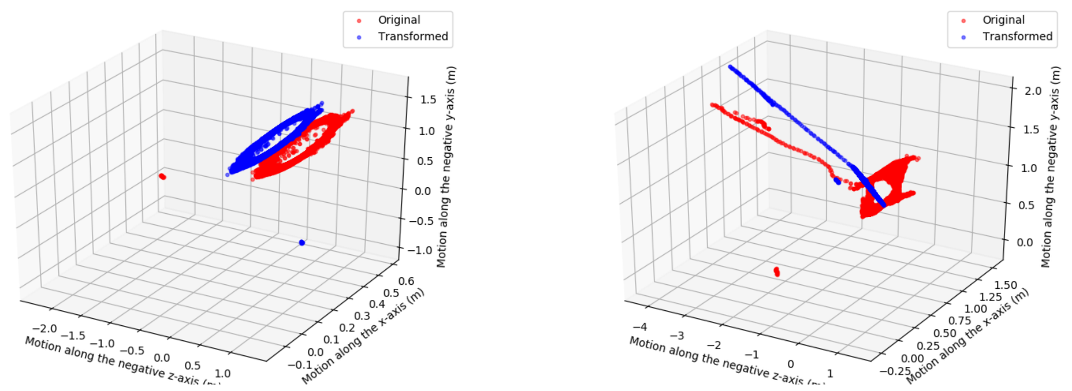
In the two motions that we manually corrected, the need for correction can also be seen from the PCA data. Indeed, the following is what PCA gives without correction. The third eigenvalue is about the same magnitude as the first two (especially close in motion 23), which means that PCA does not detect a good plane to fit. Thus, if we use the planarized motion without the manual correction, our motion will be projected to a plane that is sort of random, which means we will not be able to recover the intended motion at all.

motion	1st eigenvalue	2nd eigenvalue	3rd eigenvalue
23 (unchanged)	0.46482232	0.40469471	0.13048297
25 (unchanged)	0.38185545	0.31970822	0.29843633

Table 4: The three eigenvalues obtained from PCA, for motions 23 and 25 without correction.

Observe that the first two are always similar (with difference between 25% and 2% depending on the motion) but the third one is always significantly much lower, as corresponds to a motion that is almost planar. The third one is no longer an order of magnitude smaller than the first two. The effect is larger in 25, where the last two eigenvalues are similar, and about 20% smaller than the first one. This, together with the plot in Figure 16, shows that PCA has taken as first principal component that of the error in the motion.

This is also reflected in the plots, see Figure 16. In In 23 we still recognize the motion as circular. That means that even the error in the file was not big enough to affect the fitted plane significantly, although it affected the size of the third eigenvalue. But in motion 25 the deviation from the intended plane is bigger. In this case the number of points that are outside the "correct" path is over 120 (see section 4.1), and those are enough points and far away enough that the direction of this deviation has been taken as the first principal component, with the result that the projected motion is along a line (the diagonal of the square), with the deviation taking preference.



Motion number 23 Motion number 25

Figure 16: PCA projection of motions 23 and 25 without the manual correction).

4.3 Computation of periods

As mentioned in Section 3.4 , we compute the period in two phases: first, we compute how many "intervals" of our motion cross a small ball around a certain frame that is chosen as reference. For the computation we choose as reference frame the middle one in the whole motion and take as radius d = 0.1. The total number of frames divided by the total number of intervals found is taken as a first approximation of the period.

In a second phase we look at the gaps between the central points of the intervals computed in the first phase. Some gaps are discarded because they are too short (implying that two different intervals correspond to the same cycle) or about twice as long as they should be (implying that one of the cycles was not captured by the ball we chose). The average of the surviving gaps is taken as the period. Table 5 shows the results of both phases:

motion	Nbr. of frames	Intervals detected	1st approx. period	Outlier gaps	Gaps kept	Final esti-mated period
H11	20571	67	302.576	2	64	**302.484**
H17	39001	127	302.397	0	126	**302.397**
H20	34991	79	437.679	0	78	**437.679**
H21	25141	75	437.595	0	74	**437.595**
H23B	31411	116	267.913	13	102	**299.618**
H24	33471	111	296.091	2	108	**301.056**
H25B	50221	114	432.823	2	111	**435.153**
H26	42441	113	367.321	18	94	**432.777**

Table 5: The obtained periods and number of cycles for the individual motions. In some motions (17, 20, and 21) the final period coincides with the first approximation, because no gaps are removed. In 11, 24 and 25B a couple of gaps are removed, and in 23B and 26 13 and 18 are removed.

Let us analyze a bit more what happened in the two phases in the motions where the biggest number of gaps were outliers, namely motions 23B and 26.

In motion 23B, the initial list of 115 gaps between the 116 points selected in the first phase is:

```
[303, 305, 304, 301, 303, 302, 303, 256, 16, 2, 289, 291, 314, 304, 300, 303,
303, 302, 302, 294, 26, 286, 302, 306, 298, 301, 304, 292, 23, 292, 301, 302,
302, 286, 25, 292, 305, 301, 301, 305, 303, 305, 299, 302, 302, 304, 301, 302,
303, 304, 301, 302, 303, 302, 303, 302, 304, 302, 302, 303, 300, 303, 305, 296,
20, 2, 5, 6, 274, 303, 302, 305, 301, 302, 300, 304, 301, 276, 30, 300, 303,
302, 303, 301, 303, 302, 302, 302, 30, 279, 298, 304, 301, 302, 302, 303, 302,
304, 302, 277, 26, 301, 302, 302, 303, 304, 263, 38, 304, 301, 302, 301, 305,
301, 304]
```

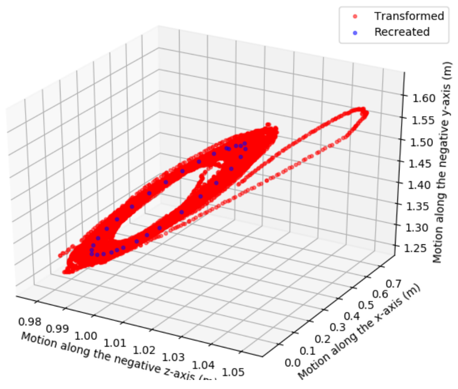
The 112 gaps between the 113 selected frames in motion 26 are:

```
[439, 439, 438, 437, 438, 436, 441, 438, 435, 437, 452, 423, 438, 438, 434, 442,
436, 439, 438, 438, 437, 431, 34, 407, 36, 398, 35, 6, 405, 34, 402, 438, 434,
33, 2, 406, 439, 436, 445, 430, 438, 437, 438, 437, 440, 435, 31, 409, 435, 32,
2, 406, 436, 436, 441, 437, 32, 406, 437, 439, 434, 440, 437, 438, 437, 438,
435, 442, 437, 436, 451, 425, 451, 45, 379, 444, 430, 440, 436, 440, 439, 435,
431, 32, 412, 439, 437, 436, 31, 2, 406, 437, 436, 29, 8, 403, 438, 434, 440,
436, 35, 404, 440, 435, 439, 436, 438, 440, 438, 439, 438, 434]
```

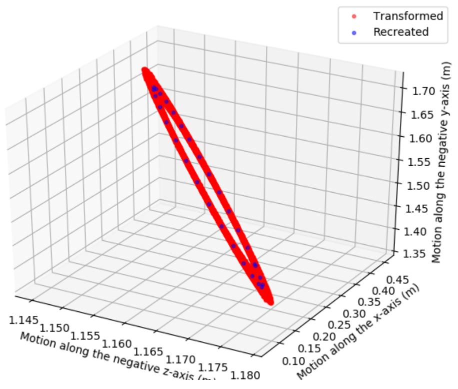
Outliers are marked in red in both lists. In all cases the error has been that we have very short gaps, meaning that the motion gets in and out of the reference ball too often. That is, we did not miss any cycle, but we overcounted cycles in the first phase.

4.4 Selection of sample points and averaging

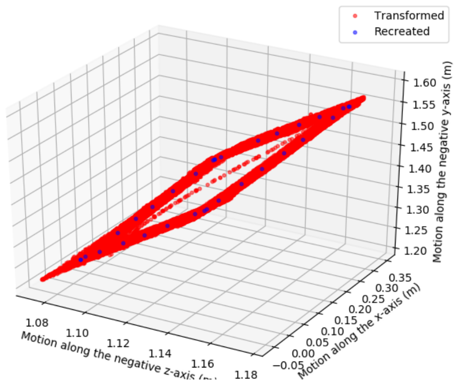
Once we have a flat motion generated by the PCA transformation, we can obtain a sample of points that we will use as a basis from which to generate the final trajectory. As we explained in section 3.5 we use the function *create_cycle_sample*, which provides us with a set number of points along a path that represents an average of the cycles. We have chosen to take 30 points along the cycles, representing a 90% to 95% reduction from the number of frames in the original periods. On Figure 17we see the results compared to the flattened motion:



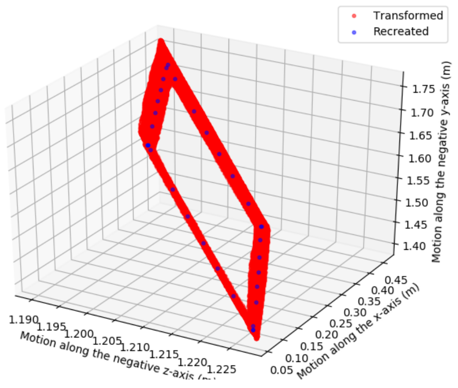
Motion number 11



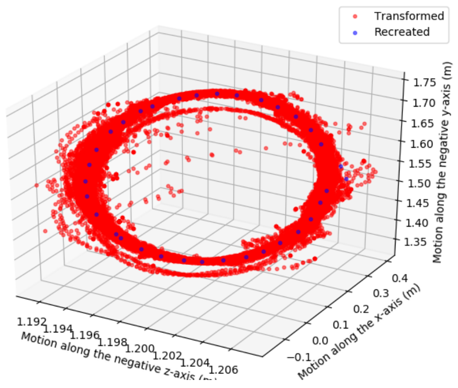
Motion number 17



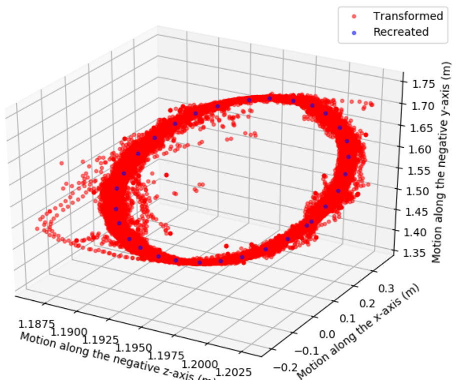
Motion number 20



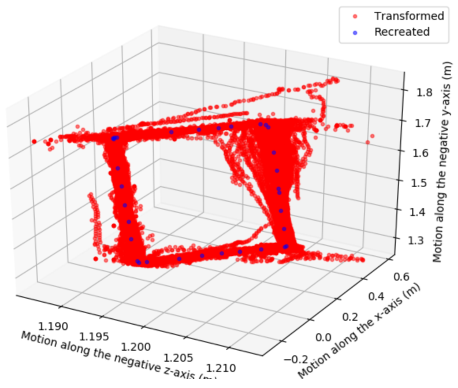
Motion number 21



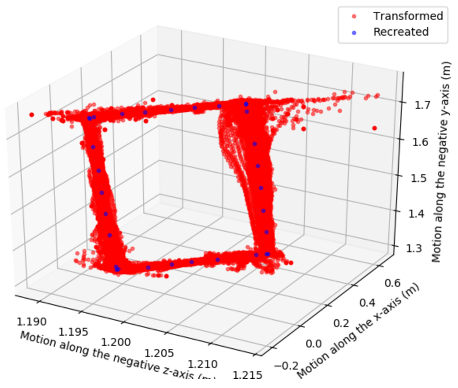
Motion number 23 (*)



Motion number 24



Motion number 25 (*)

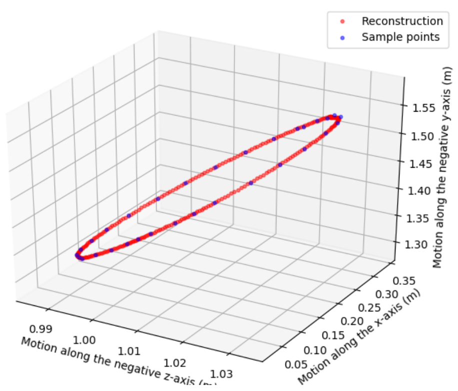


Motion number 26

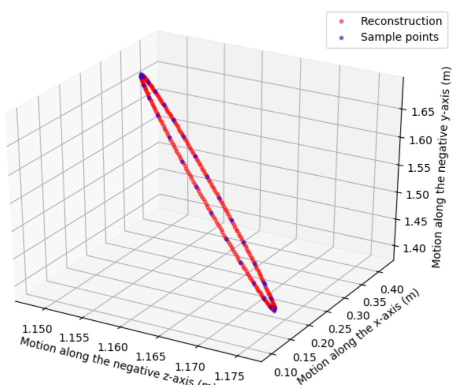
Figure 17: Human motion flattened by the PCA transformation (red) and the 30 points selected as an average cycle (blue). All of the blue points at are constant intervals of time, the points that are closer together represent a slower motion, the ones that are further apart are slower.

4.5 Reconstructing trajectory via splines

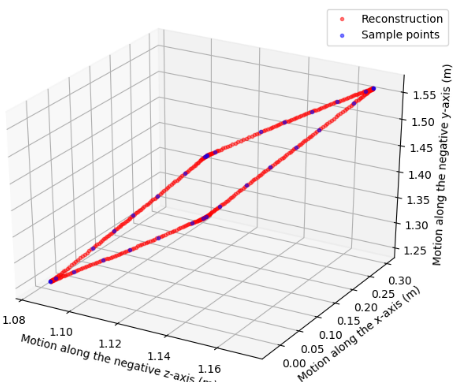
Once we have a set of points, we use splines to bridge the gaps between them, creating a full, smooth motion cycle.



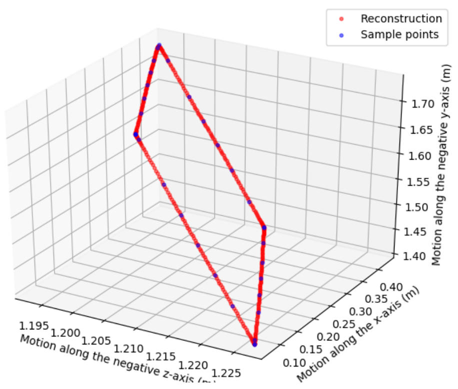
Motion number 11



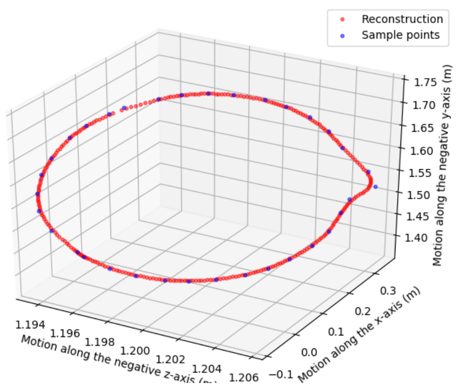
Motion number 17



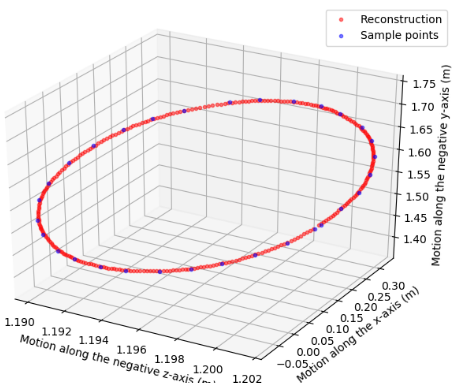
Motion number 20



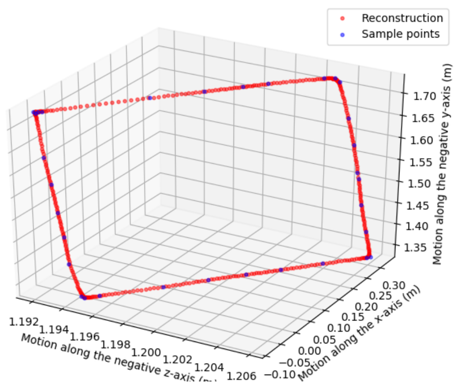
Motion number 21



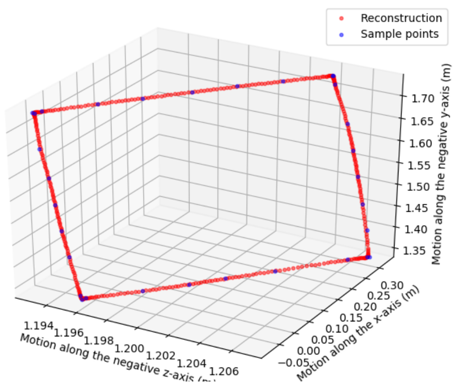
Motion number 23 (*)



Motion number 24



Motion number 25 (*)



Motion number 26

Figure 18: The sample points as seen in section 4.4 (blue) and the reconstructed motion (red). The new motion is as long as the average period (see section 3.4) with one point per frame.

We can see in some cases there are inconsistencies in the spacing of the points. In the circular motions, the velocity at all points should be the same. In the square motions it should be slower as it nears the corners and faster around the middle of each side of the square. However, we can see in some cases this does not happen, and there are also deviations from the expected trajectory. Particularly on the graph for motion 23 we can see on the right side (positive x axis) that there is a bump caused by one of the sample points being far out of the circle. On the figure for motion 25 we see instead that the velocity does not always match the expectation; for instance, there are two points on the right side that are too close, meaning that the motion will slow considerably when it reaches that area, and on the left side of the upper segment the opposite happens; the points are very separated, meaning an increase in velocity.

4.6 Comparison with original robot motion

So far, we haven't used the robot motion at all. We show it in some of the figures in the previous sections for illustration purposes, but all the computations so far use only the data in the human captured motion.

As said in Section 3.7 we are going to compare the reconstructed motion to the robot motion in three aspects.

Angle of projection

The direction of the third principal component in the PCA computation equals the normal vector of the plane that PCA selects as best fit. So, it makes sense to compare those directions for the human and the robot. First, let us show for comparison the eigenvalues obtained for the robot motion.

Robot motion nr.	1st eigenvalue	2nd eigenvalue	3rd eigenvalue
11	0. 532	0.468	0.0000617
17	0. 505	0. 495	0.000285
20	0. 512	0. 488	0.0000508
21	0. 51302	0. 481	0.00637
23	0. 513	0. 487	0.0000548
24	0. 514	0. 486	0.0000480
25	0. 506	0. 493	0.0000718
26	0. 505	0. 494	0.0000693

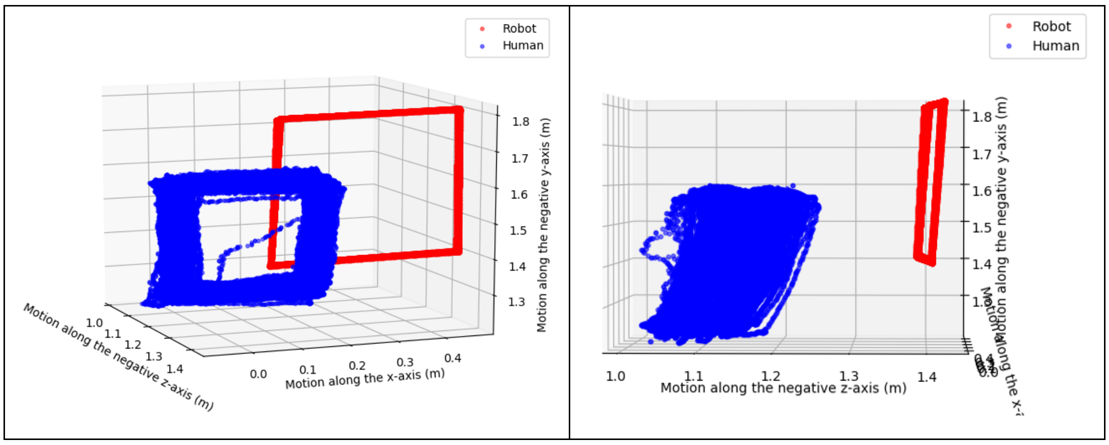
What we see is that (as expected) the robot motion is much more precise than the humans. In most of the motions the third eigenvalue is less than 10^{-4}, meaning that the deviation in the third direction is less than 10^{-2} = 1%. The two exceptions are motion 17 where it is between 1% and 2% and motion 21 where the (relative) variance in the third direction is 0.0064, so that its square root is about 0.08. This is more than 10% of the square roots of the other two components. Looking at Figure 13 we understand the reasons. In Motion 17 the robot trajectory (red oval) indeed appears thicker than in the rest of motions. In motion 21 the robot motion is very precise (the red trajectory appears quite thin) but it is not planar. It seems like the robot is describing a "spherical square", that is, it tries to draw a square, but it is constrained to lie in a sphere, perhaps due to a programming error.

We now show the angle deviation. In the following table we repeat the values of the 3rd eigenvalue for both the robot and the human, on order to check whether there is any correlation between the deviation from planarity in the two motions and the difference in angle.

motion	3rd eigenvalue human	3rd eigenvalue robot	Angular difference
11	0. 05771091	0.0000617	4.63º
17	0. 03075393	0.000285	2.92º
20	0. 03545014	0.0000508	11.68º
21	0. 02992609	0.00637	3.19º
23B	0. 00074734	0.0000548	1.40º
24	0. 00067107	0.0000480	1.30º
25B	0. 00117796	0.0000718	0.78º
26	0. 00096834	0.0000693	1.104º

Table 6: The third PCA eigenvalues for both human and robot, together with the angle difference. The first four motions, in which the human has a significantly larger third eigenvalue, (0.3-0.6) also have a larger angular deviation (3º to 11.68º). The last four motions, where the third eigenvalue of the human is in the order of 10^{-3}, have angular deviations in the order of 1º.

As seen in the table, there is indeed some correlation with the 3rd eigenvalue of the human motion. The first four motions have eigenvalue in the order of 0.3-06 and angular error of at least 3º, the last four have eigenvalue in the order of 0.001 or less, and angular error of 1.4º or less. One of the motions, number 20, has a quite big angular error, of 11.68º. This can be appreciated in the plot of these motions. In the left part of the following figure we repeat the plot from Figure 13, and in the right part we show a different perspective, that highlights the angular difference.



Periods

We now compute the period of the robot motion in exactly the same manner we did for the human. The result, and their comparison with what we got for the human, is in the next table. For the robot the final estimated period is exactly the same as the "1st approximation". That is, the second phase that we have in the function *find_period* produces no change in the outcome, because there are no "outlier gaps" at all.

	Robot		Human			
motion	Intervals detected	Period	Intervals detected	Outlier gaps	Final estimated period	**Difference**
11	67	302.394	67	2	302.484	**0.0299%**
17	127	302.389	127	0	302.397	**0.0026%**
20	79	437.628	79	0	437.679	**0.0117%**
21	55*	448.333	75*	0	437.595	**2.454%**
23	103	302.00	116	13	299.618	**0.795%**
24	109	301.565	111	2	301.056	**0.169%**
25	113	436.687	114	2	435.153	**0.353%**
26	95	437.660	113	18	432.777	**1.128%**

Table 7: Comparison of the period obtained for the human and the robot. In all cases the difference is much less than 1%, with two exceptions: motion 21 and motion 26. Motion 21 is the same one where the number of frames that we got in the BVH files for the human and robot were very different. We believe that perhaps in this motion the file we got for the human and the robot were not captured simultaneously, which could explain the big difference in the periods. As for motion 26, there the error is probably coming from our computation. This motion is the one where our second phase correction was bigger, with 18 outlier gaps. Even if the period that we got in the second phase is much better than the one from the first phase (which was 367, more than 15% the real one) the fact that the first phase result was not very good is still reflected in the final result.

Pointwise deviation

The following table contains the pointwise deviation between our reconstructed motion and the original motion of the robot, after translating them so that they have the same barycenter (but not trying to correct the angular difference or any other parameter).

motion	Maximum deviation (m.)	Average deviation (m.)	PCA angular difference
11	0.0554	0.0338	4.63º
17	0.0440	0.0311	2.92º
20	0.0704	0.0461	11.68º
21	0.0655	0.0431	3.19º
23B	0.0200	0.0055	1.40º
24	0.0046	0.0022	1.30º
25B	0.0057	0.0014	0.78º
26	0.0129	0.0070	1.104º

Table 8: Pointwise distance between the reconstructed motion and the original robot's motion (for each point in the reconstructed trajectory the closest point of the robot trajectory is computed). Since part of the error comes from the error in the PCA plane, in the last column we repeat that error (taken from Table 6) for comparison.

It is worth noting that in the cases where kinematics was used in the generation of the bvh files (motions 11 to 21, see section 3.1) all three measurements have much larger deviations than the reconstructions generated from files 23B to 26, which were recorded with a single tracker. We should also keep in mind that for motion 21 we are creating a flat motion as for all of the others, but we are comparing it to a robot motion that is not flat (see Figure 19) and this should increase the deviation.

Finally, the following figure shoes the reconstructed motion (red) versus the original robot motion (blue).

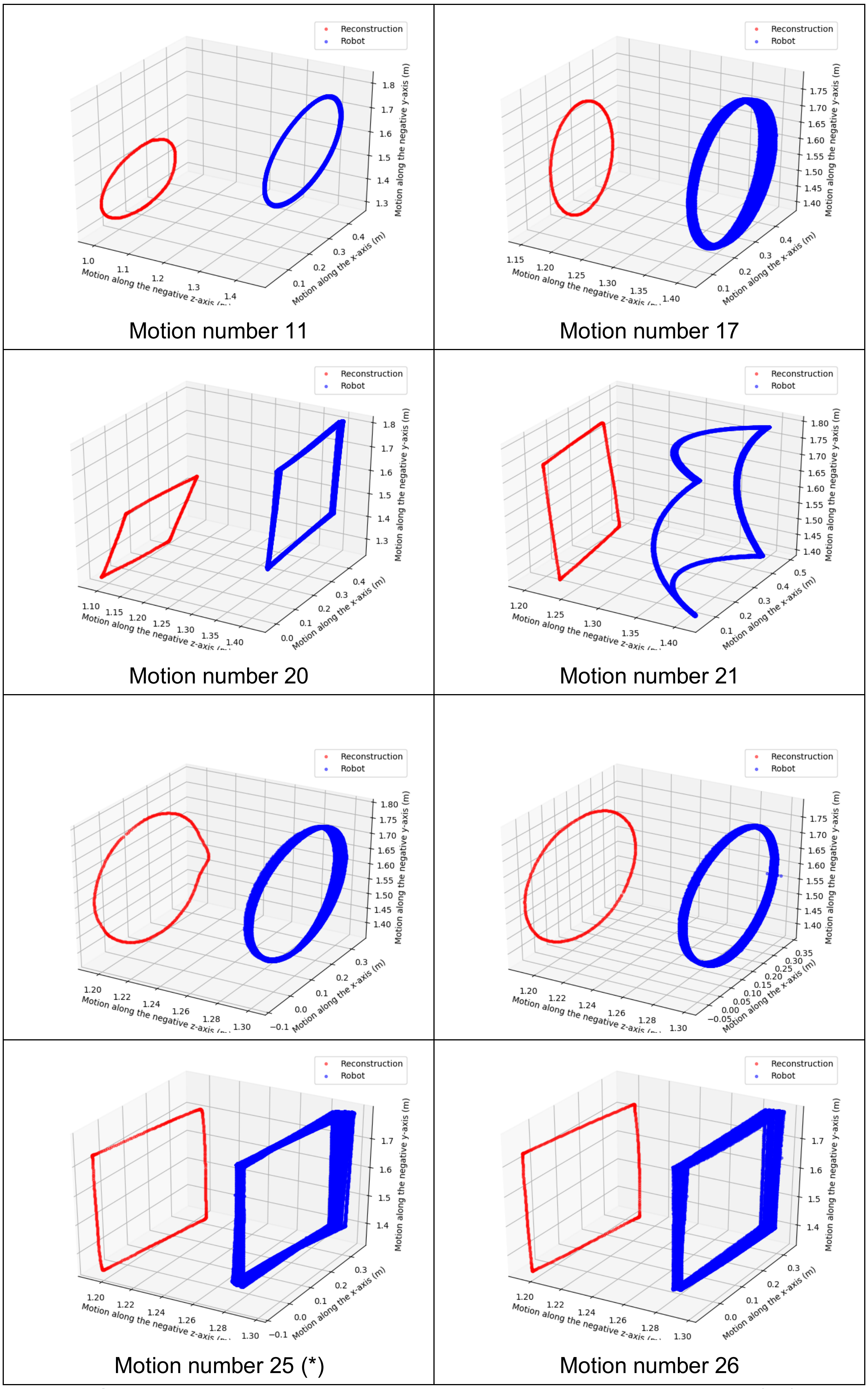


Figure 19: Comparison between the new motion obtained with our proposed methodology (red) and the original motion of the robot as captured during the recording phase (blue).

4.7 Error in the original motion

In order to determine the efficacy of our method at reducing the error generated by the recorded human motion, we determine the variation on the original motion. We use the *error_calculation* function again, this time comparing the original robot motion to the reconstructed motion we have created with the splines. Since this motion is meant to be an average of all the cycles, we assume it is a good centre with which to calculate the variance. We obtain the following results:

Motion	Maximum deviation (m.)	Average deviation (m.)	% of max. deviation reduced by our method	% of av. Deviation reduced by our method
11	0.4502	0.0393	87.7	14.2
17	0.1195	0.0238	63.2	-30.6
20	0.1602	0.0350	56.1	-31.7
21	0.1065	0.0334	38.5	-29.0
23B	0.1535	0.0091	87.0	39.6
24	0.1515	0.0068	97.0	67.6
25B	0.2657	0.0101	97.9	86.2
26	0.3798	0.0126	96.6	44.4

Table 9: Deviation of the human motion from the reconstruction, which we use here as a centre from which to draw the deviation. On the rightmost columns, the percentage of maximum and average deviation eliminated by our method, comparing the error of the human motion to the error of our reconstruction as seen in Table 8.

As we see, the maximum error is greatly reduced by our method. This is to be expected, as the human motion has large “spikes” in the deviation. The average deviation, however, is not, and we can see a clear difference between the recordings performed with a single tracker and those using kinematics. In motions 11 to 21 (recorded with forward or inverse kinematics, see section 3.1) the average error actually increases after the procedure (except in the case of motion 11, but the reduction of error is still significantly smaller than in motions 23 to 26.

Chapter 5 Conclusions

Generally, the proposed method could be considered to generate a robot motion from a human motion capture. However, we must note several issues:

It seems evident in the last two tables that the procedure works best when recording the movement of the human operator using a single tracker, without using forward or inverse kinematics. We must also keep in mind that this approach considers that the motion we are capturing is cyclical. Additionally, we must consider that its accuracy is dependent on two things:

- **Reliability of the measurement:** We are taking as a basis for the motion a recorded movement, so the reliability of this will influence how reliable our results are.
- **Reliability of the operator:** The reconstruction will be a motion that is an average of what the operator initially did in the different cycles. This initial motion by a human needs to be as accurate as possible. Humans are inherently worse than robots at cyclically repeating a movement, so utilizing a human to generate the motion might be underutilizing some of the biggest advantages of robots.

Partially to combat these issues is why we use many cycles of the motion and obtain an average. Performing many repetitions can alleviate the errors, since we assume that the error in the measurements is small in value and random in direction. While this is probably the case for the measurement error, it might not be so for the human error; the longer an operator is required to repeat a movement, the more bored and tired he will become, which can affect his performance and lower his accuracy. In the end, performing too many repetitions could become counterproductive. Looking at the eigenvectors and eigenvalues provided by the PCA, it seems that when making the circular motions the operator tends to make them wider than they are tall. This is reflected in the first eigenvector given by the PCA, which is nearly horizontal for all of the circular motions (11,17, 21 and 23B; see output in the appendix). This could be because it is easier for a human to move their hand horizontally than vertically, and making accurate circles freehand is difficult.

In the square movements this doesn't seem evident when looking at the values. However, looking at the figures (for instance Figure 15) it seems that the operator overextends on the horizontal sides of the square, creating an "overhang" that extends the sides of the square further than they should. Indeed, the reasons for this apparent horizontal bias are an interesting topic for further study but fall out of the scope of this project.

However, our method also has advantages; it is easy and intuitive, as the operator need only wear the motion capture equipment and perform the desired movements, and the resulting motion files can be processed quickly. For the motions recorded with a single tracker it has achieved deviation reductions ranging from 40% to 85%, with the maximum deviation measured in the reconstruction being 1.3 cm.

All in all, the methodology seems most useful for tasks or motions that do not require a very precise repetition but are complicated to program using more common methods. Because of how the original data is obtained, it is helpful if the motion and timing is made very clear to the operator, or if he or she has an intuitive knowledge of it. For instance, drawing or painting complicated shapes or writing words in a specific style, or even signatures, which require a flat trajectory to be followed at a determined speed (the designs would have to be large, so a deviation would not be critical).

In terms of my personal development, I want to mention several components of this project that have been new to me:

It is the first time that I have to write a structured program of this magnitude from beginning to end and dealing with files of data in a complex format like bvh. In terms of Python programming, my very first experience was in the course Software Engineering, which I took the semester previous to the realization of this project.

The mathematical techniques, mainly Principal Component Analysis and B-splines were also a novelty for me.

Bibliography

Ayusawa, K. & Yoshida, E., 2017. Motion Retargeting for Humanoid Robots Based on Simultaneous Morphing Parameter Identification and Motion Optimization. *IEEE Transactions on Robotics,* Volume 33, pp. 1-15.

Bartholomew, D. J., 2010. Principal Components Analysis. In: *International Encyclopedia of Education (Third Edition) .* s.l.:Elsevier, pp. 374-377.

British Automation and Robot Association, 2020. *Robot Programming Methods.* [Online]
Available at: https://www.ppma.co.uk/bara/expert-advice/robots/robot-programming-methods.html
[Accessed 2020].

Daniel Rakita, B. M. M. G., 2017. A Motion Retargeting Method for Effective Mimicry-based Teleoperation of Robot Arms. In: *HRI '17: Proceedings of the 2017 ACM/IEEE International Conference on Human-Robot Interaction.* Vienna, Austria: Association for Computing MachineryNew YorkNYUnited States, pp. 361-370.

Du, H. et al., 2016. Scaled Functional Principal Component Analysis for Human Motion Synthesis. In: *Proceedigs of MIG'16.* s.l.:Association for Computing Machinery, pp. 139-144.

FileInfo Team, ret. 2020. *.BVH File Extension.* [Online]
Available at: https://fileinfo.com/extension/bvh

Ford, L. R., 2008. On the Runge Example. *Mathematical Association of America,* 23 September.

Gray, A., 2014. *A Brief History of Motion-Capture in the Movies.* [Online]
Available at: https://www.ign.com/articles/2014/07/11/a-brief-history-of-motion-capture-in-the-movies
[Accessed 2020].

Hamilton, H. J., 2019. *Uniform B-Spline Curve Derivation.* [Online]
Available at:
http://www2.cs.uregina.ca/~anima/408/Notes/Interpolation/UniformBSpline.htm

IMDb, n.d. *Internet Movie Database.* [Online]
Available at: https://www.imdb.com/title/tt0120915/trivia?ref_=tt_trv_trv
[Accessed 2020].

Kubota, A., Iqbal, T., Shah, J. & Riek, L., 2019. Activity recognition in manufacturing: The roles of motion capture and sEMG+inertial wearables in detecting fine vs. gross motion. In: *2019 International Conference on Robotics and Automation (ICRA).* Montreal: s.n., pp. 6533-6539.

Patriakalis, T. & Maekawa, N. M., 2010. *Shape interrogation for computer aided design and manufacturing.* s.l.:Springer- Verlag.

Pedregosa, F. et al., 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research,* 12(85), pp. 2825-2830.

RoboDK INC, n.d. *RoboDK.* [Online]
Available at: https://robodk.com

Sewell, M., 2007. *Principal Component Analisys.* [Online]
Available at: http://www.stats.org.uk/pca/pca.pdf

Tuli, T. B. & Manns, M., 2020. Real-Time Motion Tracking for Humans and Robots in a Collaborative Assembly Task. In: *Proceedings, 2020, ECSA-6 2019.* s.l.:MDPI.

Universal Robots A/S, n.d. *https://www.universal-robots.com.* [Online]
Available at: https://www.universal-robots.com/media/50588/ur5_en.pdf
[Accessed 2020].

General description of the code and files

The code we have written comes in three files:

- *BVHtoMatrices.py*
  The script that reads the BVH files (after passing them through blender), extracts from them the information for the joint we are interested in, and writes that information into a file via the package *pickle*.

- *main.py*
  The main script used for analysing each motion. Towards the beginning of the file the user has to change the variables *noH* and *noR* to one of the labels of the files in the folder *files_matrix*, but after that the script needs no other interventio fro the user.

- *CustomFunctions.py*
  This contains the functions and class definitions used by the previous two.

Apart of these three files, the data we submit contains the following files and folders:

- File *bvhplot.py:* the script given to us, which reads BVH files and extracts the desired infromation from them.

- Folder *files_native (635 MB approx)*
  These are the sixteen BVH files, after preprocessing them via Blender. Each file is named *Rnn_native.bvh* or *Hnn_native.bvh* where *nn* is one of 11, 17, 20, 21, 23, 24, 25, 26.

- Folder *files_matrix (16 MB approx)*
  These are the eighteen *pickle* files containing the data we need fort he rest. Sixteen of them are created automatically by *BVHtoMatrices.py* (the folder includes the file *output.txt* containing the messages that *BVHtoMatrices.py* prints out while working, which includes information of the time spent for each part). The files are nammed *Rnn_pickled* or *Hnn_pickled*, where *nn* is as before, except fort he addition of *H23B* and *H25B* for the manually created files.

A. Python scripts

A.1 BVHtoMatrices.py

This script automatically goes through the sixteen input BVH files **_native.bvh*, extracts the information we want via the functions in the *bvhplot* package, and creates the corresponding sixteen *pickle* files **_pickle.bvh*.

The package *pickle* writes any Python object into a binary file that can then be read back by Python. It has the disadvantage that the internal format of *pickle* files is not stable, open, or standard: it is not readable by any program other than Python, and it even may not be readable by different versions of Python than the one that created the files. In our case this is not serious since we can consider the *pickle* files that we use as „temporary files“. If at some point in the future we want to use these files and we cannot, they can be generated from the BVH file again via this script. The reason we we use the pickle files is that processing the bvh file takes about 1 to 2 minutes per motion, and we do not want to do it over and over every time we change from studying one of the eight motions to another one. Using it instead of more standard (and open) ways of writing our data in ASCII format use allows us not to need to care about the format of the files.

The parts of this script that read the BVH files and extract the joint information for it is based in code given tu us by Mr. Tadele Belay Tuli, except we have automatized the process.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Sun Sep 13 20:19:34 2020

@author: santosgranero
"""

import pickle
from bvhplot import *
from CustomFunctions import *
from time import time

print("######################################################")
print("#                                                    #")
print("#           MOTION RETARGETING IN ROBOTICS           #")
print("# CONVERSION OF BVH FILES TO MATRIX (pickled files)  #")
print("#                                                    #")
print("######################################################")


filenumbers=[11,17,20,21,23,24,25,26]

for no in filenumbers:
    print()
    print()
    print("Starting motion number",no)
    time0=time()

    DEFAULT_BVHDIR = "files_native/"
    DEFAULT_BVHFILE = DEFAULT_BVHDIR + "R"+str(no)+"_native.bvh"
    print()
    print("Reading file", DEFAULT_BVHFILE)
    with open(DEFAULT_BVHFILE) as bvh_file:
        motion = BVHMotion(bvh_file)
    joint_nameR = "tool0"
    motion_arrayR = BVHToMatrix(motion, joint_nameR)
    del motion
    print(len(motion_arrayR.cartesian_frame),"frames read")
    print("Ellapsed time",time()-time0)

    DEFAULT_BVHFILE = DEFAULT_BVHDIR + "H"+str(no)+"_native.bvh"
    print()
    print("Reading file", DEFAULT_BVHFILE)
    with open(DEFAULT_BVHFILE) as bvh_file:
        motion = BVHMotion(bvh_file)
    joint_nameH = "hand_r"  # for 11, 17, 20, 21
    if no > 22:
        joint_nameH = "Tracker_Robot"  # for 23, 24, 25, 26
    motion_arrayH = BVHToMatrix(motion, joint_nameH)
    del motion
    print(len(motion_arrayH.cartesian_frame),"frames read")
    print("Ellapsed time",time()-time0)

    originalR = motion_arrayR.cartesian_frame
    originalH = motion_arrayH.cartesian_frame

    print()
    PICKLEDFILE ="files_matrix/R"+str(no)+"_pickled"
    print("Writing pickled file",PICKLEDFILE)
    outfile = open(PICKLEDFILE,'wb')
    pickle.dump(originalR,outfile)
    outfile.close()

    PICKLEDFILE ="files_matrix/H"+str(no)+"_pickled"
    print("Writing pickled file",PICKLEDFILE)
    outfile = open(PICKLEDFILE,'wb')
    pickle.dump(originalH,outfile)
    outfile.close()
    print("Ellapsed time",time()-time0)
```

A.2 main.py

This script takes as input two pickle files each containing an *n* x 3 matrix representing the positions of a joint at each frame. The names of the files are created automatically by the script except for the part *noH* and *noR*, each of which is a string of two or three characters corresponding to the sixteen *pickle* files created by *BVHtoMatrices.py* plus the two that we created manually for the reasons explained in Section 3.2. More precisely, these variables can be given the following values:

noH: „11“, „17“, „20“, „21“, „23“, „23B“, „24“, „25“, „25B“, „26“.

noR: „11“, „17“, „20“, „21“, „23“, „24“, „25“, „26“.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
```

```
Created on Sun Sep 13 20:19:34 2020

@author: santosgranero
"""


from CustomFunctions import *
from time import time
import pickle
from collections import OrderedDict
import numpy
import matplotlib
#matplotlib.use('GTK3Cairo')
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d.axes3d import Axes3D
from matplotlib.animation import FuncAnimation
from transformations import angles2vec4

print('#####################################################')
print('#                                                   #')
print('#          MOTION RETARGETING IN ROBOTICS           #')
print('#                                                   #')
print('#####################################################')

time0=time()
# set to print 5 decimal digits of numerical variables:
numpy.set_printoptions(precision=5, floatmode='fixed')

# loading files

noH="H20"
noR="R20"

print()
print('Reading files',noR,'and',noH)
print("------------------------")
PICKLEDFILE_R ="files_matrix/"+noR+"_pickled"
infile = open(PICKLEDFILE_R,'rb')
originalR = pickle.load(infile)
infile.close()

PICKLEDFILE_H ="files_matrix/"+noH+"_pickled"
infile = open(PICKLEDFILE_H,'rb')
originalH = pickle.load(infile)
infile.close()

loopR = range(len(originalR))
loopH = range(len(originalH))

#%%
# To print robot and human together
plot2_from_points(originalR, originalH, 'Robot', 'Human')
```

```
print('Read',len(originalR),'robot frames and',len(originalH),'hu-
man frames')
print('Plotting robot and human motion side to side')
print('Ellapsed time',time()-time0)


#%%
# Applying PCA

print()
print('Performing PCA')
PCA3ComponentsH, PCA3VarianceH, data_transH = pca_from_array(origi-
nalH, loopH)
print('Principal components (eigenvectors of the covariance mat-
rix):')
print(PCA3ComponentsH)
print()
print('Variance of the principal directions (eigenvalues):')
print(PCA3VarianceH)

plot2_from_points(originalH, data_transH+[0, 0, 0.4], 'Original',
'Transformed')

errorH = (data_transH - originalH)
print('Ellapsed time',time()-time0)

#%%
#Finding period

print()
print('Computing period')
p = find_period(data_transH)
print('Final computed period is',p)
nOfCycles = len(originalH)/p
print(' N of cycles:', nOfCycles)
print('Ellapsed time',time()-time0)

#%%
# Outliers

#%%
# def point_select(self, f)


n=30
print()
print('Computing',n,'sample points in average cycle')


[cycle, frames] = create_cycle_sample(n, data_transH, p)

print('Plotting flattened human motion vs sample points and origi-
nal robot motion vs sample points')
```

```
plot2_from_points(data_transH, cycle, 'Transformed', 'Recreated')
plot2_from_points(originalR, cycle, 'Robot', 'Recreated')

#%%

print()
print('Reconstructing spline curve from sample points')
#point = motion_arrayH.create_spline_points(10, p, cycle)

#%%
# getting the whole motion reconstructed
# inputs: time t, Period p, sample points cycle, sample frames fra-
mes
reconstruction = create_spline(cycle, p)
print('Ellapsed time',time()-time0)

#%%
print('Plotting reconstructed curve vs sample points')
plot2_from_points(reconstruction, cycle, 'Reconstruction', 'Sample
points')

#%%
print('Plotting reconstructed curve vs robot motion')
plot2_from_points(reconstruction, originalR, 'Reconstruction', 'Ro-
bot')
print('Ellapsed time',time()-time0)

#%%
print()
print('Comparisons between human motion and robot motion')
print('----------------------------------------------------')
# Computing error in angle of PCA projection
PCA3ComponentsR, PCA3VarianceR, data_transR = pca_from_array(origi-
nalR, loopR)
normalH=PCA3ComponentsH[2]
normalR=PCA3ComponentsR[2]
# print(normalR, normalH)
pi = numpy.arccos(-1)
angle = numpy.arccos(numpy.dot(normalH,normalR))
angle_abs=min(angle,pi-angle)
print('Angle between normal to fitted planes in human and ro-
bot:',angle_abs*180/pi,'degrees')
print()
# Computing difference in period estimated from human and robot
print('Computing robot period')
pR = find_period(originalR)
print('Computed robot period =',pR)
print('Difference from human:', abs(pR/p -1)*100,'%')
print()

# Computing pointwise error between reconstructed curve and origi-
nal
```

```
print('Computing distance between each reconstructed point and the
original robot motion')
[errorvector, errormod, maxerror, averror] =error_calcula-
tion(reconstruction, originalR)
print('maximum error =', maxerror,';  average error =', averror)
print('Ellapsed time',time()-time0)
print()
```

A.2 CustomFunctions.py

This file defines the classes and functions used by the previous two scripts. After importing the several packages that we use, the first part of the file defines the class *BVHToMatrix* used by the script *BVHtoMatrices.py*. This class contains no functions since this script uses the functions contained in *bvhplot.*

```
class BVHToMatrix(object):
    """
    Requires bvhplot.py to be run.

    Represents motion of a specific joint in the bvh file in a
    three column array with each column having, in order, x,y and z
coordinates
    """

    def __init__(self, motion, joint_name):
        self.cartesian_frame = []

        cartesian_frame_our_joint = []
        for i in range(len(motion.frames)):
            # Use get_cartesian_joint which requires the frame and
joint
            # and gives the cartesian position
            temp_vector = motion.get_cartesian_joint_frame(i,
joint_name)
            cartesian_frame_our_joint.append(temp_vector)

        self.cartesian_frame = numpy.asarray(carte-
sian_frame_our_joint)
```

The rest of the script defines the several functions in it used for our program. The individual functions have ben described in the different the different sections of Chapter 3.

```
 def aprox_frame_select(curve, f, d = 0.1):
    """
    Obtains points from the motion array which correspond to same
points in
    the motion cycle.

    These points are the middle point in each "interval" close
    (closer then the parameter d) to a certain frame f.
    "Interval" means a sequence of consecutive points in the mo-
tion.

    Returns the frames at which these points are.

    Inputs:
        f is the target frame
        d is the allowed distance from this frame
    Output:
        matrix of points, columns have x, y, and z coordinates.
    Error increases with the variation between loops
    Used to calculate period

    """

    # define range
    x_target = curve[f,0]
    y_target = curve[f,1]
    z_target = curve[f,2]

    x = curve[range(len(curve)), 0]
    y = curve[range(len(curve)), 1]
    z = curve[range(len(curve)), 2]

    frames_in_range = numpy.where(numpy.sqrt((x - x_target)**2 + (y
- y_target)**2 + (z - z_target)**2) <= d)

    pointstart = []
    pointend = []
    for t in range(1, len(frames_in_range[0])):
        if (frames_in_range[0][t] != frames_in_range[0][t-1]+1):
            pointstart.append(frames_in_range[0][t])
            pointend.append(frames_in_range[0][t-1])

    point = []
    for t in range(len(pointstart)-1):
        point.append(math.floor((pointstart[t]+pointend[t+1])/2))

    return(point)


def find_period(curve, f = 0, d = 0.1):
    """
    Finds the period of the motion cycle.
    Works best on R because they are more regular

    Inputs:
        f is the target frame
        d is the maximum distance from this frame
    Output:
        period
    """
    if f==0:
        f = math.floor(len(curve)/2)
    # First approximation: compute intervals within range and take
midpoint of each
    selected_frames = aprox_frame_select(curve, f, d)
    dif = selected_frames[len(selected_frames)-1] - selected_fra-
mes[0]
    temp_av_period = dif / (len(selected_frames)-1)

    print('First aproximation:',len(selected_frames),'cycles, esti-
mated period =',temp_av_period)
    """
    Second approximation of period is to compute the average gap
    between two selected points, but outliers need to be not coun-
ted:
    If the gap between two selected frames is too short, it means
    two "intervals" correspond to the same period (the curve goes 
out
    of the ball of radius d and back). If the gap between two sel-
ected
    frames is too long, it means one of the periods deviated too 
much
    and has not been detected.
    """
    dif_collect = []
    for i in range(1,len(selected_frames)):
        if (abs(selected_frames[i] - selected_frames[i-1] - 
temp_av_period) < 0.25 * temp_av_period):
            dif_collect.append(selected_frames[i] - selected_fra-
mes[i-1])
    av_period = sum(dif_collect) / len(dif_collect)
    print('Number of gaps kept to compute period is',len(dif_coll-
ect))
    #print("Their gaps are",dif_collect)
    return(av_period)


def _remove_outliers(curve, f, d = 0.05):
    """
    takes a collection of frames and removes ones whose points are 
too far
    from the centre (average)
    Inputs:
        curve: the matrix containing the motion we want to study
        f:  list of frames where we want to check the curve
        d:  maximum distance a point can be from the average,
        others will be removed
    """
    points = []
    i = 0
    for a in f:
        points.append(curve[a,:])
    points = numpy.asarray(points)
    target = points.mean(0)

    f2 = numpy.where(numpy.sqrt((points[:,0] - target[0])**2 + 
(points[:,1] - target[1])**2 + (points[:,2] - target[2])**2) <= d)

    frames_in_range = []
    for a in f2[0]:
        frames_in_range.append(f[a])

    return(frames_in_range)


def frame_select(f, curve, period = 0, d = 0.05):
    """
    Selecting points that correspond to the same point in a cycle.
    Better than aprox version, but requires period.
    Input:
        f:  reference frame
        curve: the matrix containing the motion we want to study
        p:  period of the motion. If none is given it will call the
        function that calculates it
        d:  fed to remove outliers

    NOT USED
    Problem: a small error in the calculation of the period leads
to very
    large error in the result of this function, since the errors
get
    added over cycles
    """
    if period == 0:
        period = find_period(curve)
    points = []
    i = f
    while (i <= len(curve)):
        points.append(i)
        i = i + period

    points = [math.floor(frame) for frame in points]
    points = _remove_outliers(curve, points, d)
    return(points)


def create_cycle_sample(n, curve, period = 0, d = 0.05):
    """
    Generates n points of an ideal cycle
    Input:
        n: the number op points in the cycle
        curve: the matrix containing the motion we want to study
        period: the period of the movement
    """

    if period == 0:
        period = find_period(curve)
    t = math.floor(period / n)
    i = 0
    addedtime = math.floor(len(curve)/2)
    cycle = []
    frames = []

    f = aprox_frame_select(curve, i+addedtime, d)
    point = curve[f, :]
    point = point.mean(0)
    cycle = [point]
    frames.insert(0, i)
    i = i+t

    while i < period:
        f = aprox_frame_select(curve, i+addedtime, d)
        point = curve[f,:]
        point = point.mean(0)
        cycle = numpy.r_[cycle, [point]]
        frames.insert(0, i)
        i = i+t

    return [cycle, frames]


def pca_from_array(curve, frames = 0):
    """
    Generate matrix containing eigenvectors and a vector with ei-
genvalues
    after applying PCA to a set of 3d points.
    Input:
        curve: the matrix containing the motion we want to study
        frames: vector with the frames we want to analyze

    Software by:
    Scikit-learn: Machine Learning in Python, Pedregosa et al.,
JMLR 12, pp. 2825-2830, 2011
    """

    if frames == 0:
        frames = range(len(curve))

    PCAData = curve[frames, :] # (numpy array of shape (n_samples,
n_points, 3)
    pca3 = PCA(n_components=3)
    transformed3 = pca3.fit_transform(PCAData)
    Data_new3 = pca3.inverse_transform(transformed3)



    pca = PCA(n_components=2)
    transformed = pca.fit_transform(PCAData)
    Data_new = pca.inverse_transform(transformed)

    return(pca3.components_, pca3.explained_variance_ratio_,
Data_new)


def create_spline_points(t, p, cycle):
    """
    Calculates a single point in the reconstructed trajectory at a
time t by
    using the formula of order 4 B-splines.
    Called by create_spline
    Inputs:
        t: time
        p: period
        cycle: sample points
    """
    n = len(cycle)
    p = math.floor(p)
    i = math.floor(t * n / p)
    im1 = i - 1
    ip1 = i + 1
    ip2 = i + 2
    if im1 < 0:
        im1 = im1 + n
    if ip1 >= n:
        ip1 = ip1 - n
    if ip2 >= n:
        ip2 = ip2 - n
    u = (t*n/p - i)
    pim1 = cycle[im1,:]
    pi   = cycle[i,:]
    pip1 = cycle[ip1,:]
    pip2 = cycle[ip2,:]
    S = (1/6*(1-u)**3) * pim1 + (2/3 - 1/2*(2-u)*u**2) * pi + (2/3
- 1/2*(1+u)*(1-u)**2) * pip1 + 1/6*u**3 * pip2

    return(S)


def create_spline(cycle, period = 0, d = 0.1):
    """
    Generates a single cycle of the reconstructed motion
    Inputs:
        t: time
        p: period
        cycle: sample points
    """
    if period == 0:
        period = find_period()
    cycle=cycle[0:len(cycle)-1]
    reconstruction = []
    for f in range(math.floor(period)-1):
        point = create_spline_points(f, period, cycle)
        reconstruction.append(point.tolist())
    reconstruction = numpy.asarray(reconstruction)

    return(reconstruction)

#_________________________________________________________________
________

def error_calculation(curve, reference):
    """
    Calculates the error between every point in a curve and the
closest
    point in the reference curve. Generates the error vectors and
the
    distance for each point, as well as the maimum and average er-
rors
    Input:
        curve: Motion we want to check
        reference: what we check against
    """
    displacement = curve.mean(0) - reference.mean(0)
    # reference = reference[range(10000,11000)] # temp, in order to
save time in tests
    reference = reference # temp, in order to save time in tests
    reference = reference + displacement
    errorvector = []
    errormod = []
    for i in range(len(curve)):
        e1 = 1000
        evec = []
        for j in range(len(reference)):
            evector = curve[i] - reference[j]
            emod = math.sqrt(evector[0]**2 + evector[1]**2 + evec-
tor[2]**2)
            if emod < e1:
                evec = evector
                e1 = emod
        errorvector.append(evec)
        errormod.append(e1)

        maxerror = max(errormod)
        averror = sum(errormod)/len(errormod)

    return(errorvector, errormod, maxerror, averror)


#_________________________________________________________________
________
```

```
def plot_from_frames(curve, frames):
    """
    Plots the position of the joint in the frames selected by the
    input vector frames.

    Adapted from a program sent by Tadele Belay Tuli on the
30.06.2020
    """

    t_axis = frames
    x_axis = curve[t_axis, 0]
    y_axis = curve[t_axis, 1]
    z_axis = curve[t_axis, 2]

    fig = plt.figure()
    ax = Axes3D(fig)
    ax.plot3D(z_axis, x_axis, y_axis, '.r', label = 'right wrist',
alpha=0.5)

    plt.xlabel('Motion along the z-axis (mm)')
    plt.ylabel('Motion along the x-axis (mm)')
    legend = ax.legend(loc='upper right')
    plt.show()


def plot2_from_frames(curve, frames1, frames2, label1 = '', label2
= ''):
    """
    Plots the position of the joint in the 2 sets of frames selec-
ted by the
    input vectors in red and blue.
    """
    fig = plt.figure()
    ax = Axes3D(fig)
    ax.plot3D(curve[frames1, 2], curve[frames1, 0], curve[frames1,
1], '.r', label = label1,  alpha=0.5)
    ax.plot3D(curve[frames2, 2], curve[frames2, 0], curve[frames2,
1], '.b', label = label2,  alpha=0.1)
    plt.xlabel('Motion along the z-axis (mm)')
    plt.ylabel('Motion along the x-axis (mm)')
    legend = ax.legend(loc='upper right')
    ax.auto_scale_xyz
    plt.show()

def plot_from_points(points1, label1 = ''):
    """
    Plots the positions given by a set of data selected by the
    input vector.
    """
    fig = plt.figure()
    ax = Axes3D(fig)
    ax.plot3D(-points1[:, 2], points1[:, 0], -points1[:, 1], '.r',
label = label1,  alpha=0.5)

    ax.set_xlabel('Motion along the negative z-axis (m)')
    ax.set_ylabel('Motion along the x-axis (m)')
    ax.set_zlabel('Motion along the negative y-axis (m)')

    legend = ax.legend(loc='upper right')
    ax.auto_scale_xyz
    plt.show()


def plot2_from_points(points1, points2, label1 = '', label2 = ''):
    """
    Plots the positions given by 2 sets of data selected by the
    input vectors in red and blue.
    """
    fig = plt.figure()
    ax = Axes3D(fig)
    ax.plot3D(-points1[:, 2], points1[:, 0], -points1[:, 1], '.r',
label = label1,  alpha=0.5)
    ax.plot3D(-points2[:, 2], points2[:, 0], -points2[:, 1], '.b',
label = label2,  alpha=0.5)

    ax.set_xlabel('Motion along the negative z-axis (m)')
    ax.set_ylabel('Motion along the x-axis (m)')
    ax.set_zlabel('Motion along the negative y-axis (m)')

    legend = ax.legend(loc='upper right')
    ax.auto_scale_xyz
    plt.show()
```

B. Output printouts

B.1 BVHtoMatrices.py output

This output simply tells us which file is the script processing, and how much time it took to process it. Time is set to zero after each pair of H and R files. The time taken by the eight (pairs of) files are as follows.

Motion no.	Time (Secs)	Motion no.	Time (Secs)
11	61.4	23	115.4
17	108.9	24	91.3
20	56.2	25	55.8
21	86.2	26	73.6

The total time is 649 seconds, that is, almost 11 minutes.

```
#######################################################
#                                                     #
#            MOTION RETARGETING IN ROBOTICS           #
# CONVERSION OF BVH FILES TO MATRIX (pickled files)   #
#                                                     #
#######################################################


Starting motion number 11

Reading file files_native/R11_native.bvh
20571 frames read
Ellapsed time 28.681632041931152

Reading file files_native/H11_native.bvh
20571 frames read
Ellapsed time 61.40838122367859

Writing pickled file files_matrix/R11_pickled
Writing pickled file files_matrix/H11_pickled
Ellapsed time 61.41848301887512


Starting motion number 17

Reading file files_native/R17_native.bvh
39001 frames read
Ellapsed time 53.39752697944641

Reading file files_native/H17_native.bvh
39001 frames read
Ellapsed time 115.33069086074829

Writing pickled file files_matrix/R17_pickled
Writing pickled file files_matrix/H17_pickled
Ellapsed time 115.35421586036682
```

```
Starting motion number 20

Reading file files_native/R20_native.bvh
34991 frames read
Ellapsed time 53.83756899833679

Reading file files_native/H20_native.bvh
34991 frames read
Ellapsed time 108.93157291412354

Writing pickled file files_matrix/R20_pickled
Writing pickled file files_matrix/H20_pickled
Ellapsed time 108.94269394874573


Starting motion number 21

Reading file files_native/R21_native.bvh
25141 frames read
Ellapsed time 34.3460590839386

Reading file files_native/H21_native.bvh
33251 frames read
Ellapsed time 91.27654886245728

Writing pickled file files_matrix/R21_pickled
Writing pickled file files_matrix/H21_pickled
Ellapsed time 91.29086303710938


Starting motion number 23

Reading file files_native/R23_native.bvh
31411 frames read
Ellapsed time 48.12377119064331

Reading file files_native/H23_native.bvh
31411 frames read
Ellapsed time 56.224478006362915

Writing pickled file files_matrix/R23_pickled
Writing pickled file files_matrix/H23_pickled
Ellapsed time 56.238057136535645


Starting motion number 24

Reading file files_native/R24_native.bvh
33471 frames read
Ellapsed time 47.073304653167725
```

```
Reading file files_native/H24_native.bvh
33471 frames read
Ellapsed time 55.75749468803406

Writing pickled file files_matrix/R24_pickled
Writing pickled file files_matrix/H24_pickled
Ellapsed time 55.769695520401


Starting motion number 25

Reading file files_native/R25_native.bvh
50221 frames read
Ellapsed time 69.7612657546997

Reading file files_native/H25_native.bvh
50221 frames read
Ellapsed time 86.15089893341064

Writing pickled file files_matrix/R25_pickled
Writing pickled file files_matrix/H25_pickled
Ellapsed time 86.17826581001282


Starting motion number 26

Reading file files_native/R26_native.bvh
42441 frames read
Ellapsed time 59.365992307662964

Reading file files_native/H26_native.bvh
42441 frames read
Ellapsed time 73.5438163280487

Writing pickled file files_matrix/R26_pickled
Writing pickled file files_matrix/H26_pickled
Ellapsed time 73.56251692771912
```

B.2 main.py output

We give the output for each pair of files. In case of motions 23 and 25 we give the output both fort he original files H23 and H25 and fort he manually modified ones H23B and H23B.

noH="11", noR="11"

```
######################################################
#                                                    #
#           MOTION RETARGETING IN ROBOTICS           #
#                                                    #
######################################################

Reading files R11 and H11
-------------------------
```

```
Read 20571 robot frames and 20571 human frames
Plotting robot and human motion side to side
Ellapsed time 0.22752594947814941

Performing PCA
Principal components (eigenvectors of the covariance matrix):
[[ 0.94940  0.31394  0.00884]
 [ 0.31168 -0.93835 -0.14950]
 [ 0.03864 -0.14469  0.98872]]

Variance of the principal directions (eigenvalues):
[0.52215 0.42014 0.05771]
Ellapsed time 0.6266419887542725

Computing period
First aproximation: 67 cycles, estimated period =
302.57575757575756
Number of gaps kept to compute period is 64
Final computed period is 302.484375
 N of cycles: 68.00681853401518
Ellapsed time 0.6419761180877686

Computing 30 sample points in average cycle
Plotting flattened human motion vs sample points and original robot
motion vs sample points

Reconstructing spline curve from sample points
Ellapsed time 1.1118168830871582
Plotting reconstructed curve vs sample points
Plotting reconstructed curve vs robot motion
Ellapsed time 1.2344708442687988

Comparisons between human motion and robot motion
-------------------------------------------------
Angle between normal to fitted planes in human and robot:
4.630344827985865 degrees

Computing robot period
First aproximation: 67 cycles, estimated period = 302.3939393939394
Number of gaps kept to compute period is 66
Computed robot period = 302.3939393939394
Difference from human: 0.02989761241737332 %

Computing distance between each reconstructed point and the origi-
nal robot motion
maximum error = 0.05542693071837361 ;  average error =
0.033772292095372455
Ellapsed time 22.054478883743286
```

noH="17", noR="17"

```
#######################################################
#                                                     #
#            MOTION RETARGETING IN ROBOTICS           #
#                                                     #
#######################################################

Reading files R17 and H17
-------------------------
Read 39001 robot frames and 39001 human frames
Plotting robot and human motion side to side
Ellapsed time 0.15599584579467773

Performing PCA
Principal components (eigenvectors of the covariance matrix):
[[ 0.98494  0.16149  0.06171]
 [-0.15780  0.98561 -0.06066]
 [ 0.07062 -0.05000 -0.99625]]

Variance of the principal directions (eigenvalues):
[0.49375 0.47550 0.03075]
Ellapsed time 0.35452890396118164

Computing period
First aproximation: 127 cycles, estimated period =
302.3968253968254
Number of gaps kept to compute period is 126
Final computed period is 302.3968253968254
 N of cycles: 128.97291480762163
Ellapsed time 0.3855600357055664

Computing 30 sample points in average cycle
Plotting flattened human motion vs sample points and original robot
motion vs sample points

Reconstructing spline curve from sample points
Ellapsed time 1.4299588203430176
Plotting reconstructed curve vs sample points
Plotting reconstructed curve vs robot motion
Ellapsed time 1.9965288639068604

Comparisons between human motion and robot motion
-------------------------------------------------
Angle between normal to fitted planes in human and robot:
2.9225840693828125 degrees

Computing robot period
First aproximation: 127 cycles, estimated period =
302.3888888888889
Number of gaps kept to compute period is 126
Computed robot period = 302.3888888888889
Difference from human: 0.0026245341451902604 %

Computing distance between each reconstructed point and the origi-
nal robot motion
maximum error = 0.04396559977803379 ;  average error =
0.03109959929526549
```

```
Ellapsed time 44.060827016830444
```

noH="20", noR="20"

```
######################################################
#                                                    #
#           MOTION RETARGETING IN ROBOTICS           #
#                                                    #
######################################################

Reading files R20 and H20
-------------------------
Read 34991 robot frames and 34991 human frames
Plotting robot and human motion side to side
Ellapsed time 0.22249913215637207

Performing PCA
Principal components (eigenvectors of the covariance matrix):
[[ 0.24794  0.94949  0.19235]
 [-0.96698  0.23048  0.10871]
 [ 0.05889 -0.21295  0.97529]]

Variance of the principal directions (eigenvalues):
[0.54768 0.41687 0.03545]
Ellapsed time 0.4819040298461914

Computing period
First aproximation: 79 cycles, estimated period = 437.6794871794872
Number of gaps kept to compute period is 78
Final computed period is 437.6794871794872
 N of cycles: 79.94662995401154
Ellapsed time 0.5062730312347412

Computing 30 sample points in average cycle
Plotting flattened human motion vs sample points and original robot
motion vs sample points

Reconstructing spline curve from sample points
Ellapsed time 1.9423890113830566
Plotting reconstructed curve vs sample points
Plotting reconstructed curve vs robot motion
Ellapsed time 2.8479361534118652

Comparisons between human motion and robot motion
-------------------------------------------------
Angle between normal to fitted planes in human and robot:
11.697499140432289 degrees

Computing robot period
First aproximation: 79 cycles, estimated period =
437.62820512820514
Number of gaps kept to compute period is 78
Computed robot period = 437.62820512820514
```

```
Difference from human: 0.011716804827321958 %

Computing distance between each reconstructed point and the origi-
nal robot motion
maximum error = 0.0703674464730722 ;  average error =
0.04609964846400692
Ellapsed time 57.47177600860596
```

noH="21", noR="21"

```
######################################################
#                                                    #
#           MOTION RETARGETING IN ROBOTICS           #
#                                                    #
######################################################

Reading files R21 and H21
-------------------------
Read 25141 robot frames and 33251 human frames
Plotting robot and human motion side to side
Ellapsed time 0.07516598701477051

Performing PCA
Principal components (eigenvectors of the covariance matrix):
[[ 0.99878  0.01828  0.04580]
 [ 0.01586 -0.99849  0.05261]
 [-0.04669  0.05182  0.99756]]

Variance of the principal directions (eigenvalues):
[0.49846 0.47161 0.02993]
Ellapsed time 0.765510082244873

Computing period
First aproximation: 75 cycles, estimated period = 437.5945945945946
Number of gaps kept to compute period is 74
Final computed period is 437.5945945945946
 N of cycles: 75.98585633994195
Ellapsed time 0.7895631790161133

Computing 30 sample points in average cycle
Plotting flattened human motion vs sample points and original robot
motion vs sample points

Reconstructing spline curve from sample points
Ellapsed time 1.742537021636963
Plotting reconstructed curve vs sample points
Plotting reconstructed curve vs robot motion
Ellapsed time 1.8875000476837158

Comparisons between human motion and robot motion
-------------------------------------------------
Angle between normal to fitted planes in human and robot:
3.185450765878103 degrees
```

```
Computing robot period
First aproximation: 55 cycles, estimated period = 448.3333333333333
Number of gaps kept to compute period is 54
Computed robot period = 448.3333333333333
Difference from human: 2.4540382517036274 %

Computing distance between each reconstructed point and the origi-
nal robot motion
maximum error = 0.06547573279998524 ;  average error =
0.043057795541597776
Ellapsed time 44.92396903038025
```

noH="23", noR="23"

```
######################################################
#                                                    #
#           MOTION RETARGETING IN ROBOTICS           #
#                                                    #
######################################################

Reading files R23 and H23
-------------------------
Read 31411 robot frames and 31411 human frames
Plotting robot and human motion side to side
Ellapsed time 0.07972908020019531

Performing PCA
Principal components (eigenvectors of the covariance matrix):
[[ 0.14186  0.95521  0.25972]
 [-0.98974  0.14146  0.02034]
 [-0.01731 -0.25994  0.96547]]

Variance of the principal directions (eigenvalues):
[0.46482 0.40469 0.13048]
Ellapsed time 0.5252242088317871

Computing period
First aproximation: 114 cycles, estimated period =
272.65486725663715
Number of gaps kept to compute period is 102
Final computed period is 299.8921568627451
 N of cycles: 104.74098532152081
Ellapsed time 0.549199104309082

Computing 30 sample points in average cycle
Plotting flattened human motion vs sample points and original robot
motion vs sample points

Reconstructing spline curve from sample points
Ellapsed time 1.3428001403808594
Plotting reconstructed curve vs sample points
Plotting reconstructed curve vs robot motion
Ellapsed time 1.5396342277526855
```

```
Comparisons between human motion and robot motion
-------------------------------------------------
Angle between normal to fitted planes in human and robot:
14.279254888675219 degrees

Computing robot period
First aproximation: 103 cycles, estimated period = 302.0
Number of gaps kept to compute period is 102
Computed robot period = 302.0
Difference from human: 0.7028670437085216 %

Computing distance between each reconstructed point and the origi-
nal robot motion
maximum error = 0.049396570496112284 ;  average error =
0.030142077831302586
Ellapsed time 32.12533402442932
```

noH="23B", noR="23"

```
######################################################
#                                                    #
#           MOTION RETARGETING IN ROBOTICS           #
#                                                    #
######################################################

Reading files R23 and H23B
-------------------------
Read 31411 robot frames and 31411 human frames
Plotting robot and human motion side to side
Ellapsed time 0.06279706954956055

Performing PCA
Principal components (eigenvectors of the covariance matrix):
[[-0.96559  0.25881  0.02535]
 [ 0.25859  0.96592 -0.01169]
 [-0.02751 -0.00473 -0.99961]]

Variance of the principal directions (eigenvalues):
[0.50777 0.49148 0.00075]
Ellapsed time 0.49095988273620605

Computing period
First aproximation: 116 cycles, estimated period =
267.9130434782609
Number of gaps kept to compute period is 102
Final computed period is 299.61764705882354
 N of cycles: 104.83694905271425
Ellapsed time 0.5138599872589111

Computing 30 sample points in average cycle
Plotting flattened human motion vs sample points and original robot
motion vs sample points

Reconstructing spline curve from sample points
```

```
Ellapsed time 1.3402070999145508
Plotting reconstructed curve vs sample points
Plotting reconstructed curve vs robot motion
Ellapsed time 1.4781827926635742

Comparisons between human motion and robot motion
-------------------------------------------------
Angle between normal to fitted planes in human and robot:
1.4031911334678666 degrees

Computing robot period
First aproximation: 103 cycles, estimated period = 302.0
Number of gaps kept to compute period is 102
Computed robot period = 302.0
Difference from human: 0.7951310493766517 %

Computing distance between each reconstructed point and the origi-
nal robot motion
maximum error = 0.020030610666652304 ;  average error =
0.0054501496453678005
Ellapsed time 33.58343291282654
```

noH="24", noR="24"

```
######################################################
#                                                    #
#           MOTION RETARGETING IN ROBOTICS           #
#                                                    #
######################################################

Reading files R24 and H24
-------------------------
Read 33471 robot frames and 33471 human frames
Plotting robot and human motion side to side
Ellapsed time 0.15832805633544922

Performing PCA
Principal components (eigenvectors of the covariance matrix):
[[-0.95013  0.31056  0.02826]
 [ 0.31061  0.95053 -0.00284]
 [-0.02775  0.00608 -0.99960]]

Variance of the principal directions (eigenvalues):
[0.50505 0.49427 0.00067]
Ellapsed time 0.36424899101257324

Computing period
First aproximation: 111 cycles, estimated period =
296.09090909090907
Number of gaps kept to compute period is 108
Final computed period is 301.05555555555554
 N of cycles: 111.17881527957188
Ellapsed time 0.39951491355895996
```

```
Computing 30 sample points in average cycle
Plotting flattened human motion vs sample points and original robot
motion vs sample points

Reconstructing spline curve from sample points
Ellapsed time 1.6300380229949951
Plotting reconstructed curve vs sample points
Plotting reconstructed curve vs robot motion
Ellapsed time 2.132643938064575

Comparisons between human motion and robot motion
-------------------------------------------------
Angle between normal to fitted planes in human and robot:
1.302006351945102 degrees

Computing robot period
First aproximation: 109 cycles, estimated period =
301.56481481481484
Number of gaps kept to compute period is 108
Computed robot period = 301.56481481481484
Difference from human: 0.16915790121179164 %

Computing distance between each reconstructed point and the origi-
nal robot motion
maximum error = 0.0046224361651706096 ;  average error =
0.0022187397599984517
Ellapsed time 37.91122508049011
```

noH="25", noR="25"

```
#####################################################
#                                                   #
#           MOTION RETARGETING IN ROBOTICS          #
#                                                   #
#####################################################

Reading files R25 and H25
-------------------------
Read 50221 robot frames and 50221 human frames
Plotting robot and human motion side to side
Ellapsed time 0.24767327308654785

Performing PCA
Principal components (eigenvectors of the covariance matrix):
[[ 0.23903  0.03017  0.97054]
 [ 0.48831  0.86020 -0.14700]
 [-0.83929  0.50907  0.19088]]

Variance of the principal directions (eigenvalues):
[0.38186 0.31971 0.29844]
Ellapsed time 0.5414032936096191

Computing period
```

```
First aproximation: 297 cycles, estimated period =
167.78716216216216
Number of gaps kept to compute period is 6
Final computed period is 132.5
 N of cycles: 379.0264150943396
Ellapsed time 0.581373929977417

Computing 30 sample points in average cycle
Plotting flattened human motion vs sample points and original robot
motion vs sample points

Reconstructing spline curve from sample points
Ellapsed time 2.289809226989746
Plotting reconstructed curve vs sample points
Plotting reconstructed curve vs robot motion
Ellapsed time 2.926636219024658

Comparisons between human motion and robot motion
-------------------------------------------------
Angle between normal to fitted planes in human and robot:
80.29034022746978 degrees

Computing robot period
First aproximation: 113 cycles, estimated period = 436.6875
Number of gaps kept to compute period is 112
Computed robot period = 436.6875
Difference from human: 229.57547169811318 %

Computing distance between each reconstructed point and the origi-
nal robot motion
maximum error = 0.193981239007396 ;  average error =
0.16058692409274658
Ellapsed time 26.450074195861816
```

noH="25B", noR="25"

```
######################################################
#                                                    #
#           MOTION RETARGETING IN ROBOTICS           #
#                                                    #
######################################################

Reading files R25 and H25B
-------------------------
Read 50221 robot frames and 50221 human frames
Plotting robot and human motion side to side
Ellapsed time 0.3994019031524658

Performing PCA
Principal components (eigenvectors of the covariance matrix):
[[ 0.83985  0.54212 -0.02738]
 [ 0.54206 -0.84028 -0.01016]
 [-0.02852 -0.00631 -0.99957]]
```

```
Variance of the principal directions (eigenvalues):
[0.51167 0.48715 0.00118]
Ellapsed time 0.8280019760131836

Computing period
First aproximation: 114 cycles, estimated period =
432.82300884955754
Number of gaps kept to compute period is 111
Final computed period is 435.15315315315314
 N of cycles: 115.40994161732434
Ellapsed time 0.862036943435669

Computing 30 sample points in average cycle
Plotting flattened human motion vs sample points and original robot
motion vs sample points

Reconstructing spline curve from sample points
Ellapsed time 2.7526841163635254
Plotting reconstructed curve vs sample points
Plotting reconstructed curve vs robot motion
Ellapsed time 3.910551071166992

Comparisons between human motion and robot motion
-------------------------------------------------
Angle between normal to fitted planes in human and robot:
0.7838800835691515 degrees

Computing robot period
First aproximation: 113 cycles, estimated period = 436.6875
Number of gaps kept to compute period is 112
Computed robot period = 436.6875
Difference from human: 0.35259927125170787 %

Computing distance between each reconstructed point and the origi-
nal robot motion
maximum error = 0.00572791939314946 ;  average error =
0.0014234237515080353
Ellapsed time 79.49503993988037
```

noH="26", noR="26"

```
######################################################
#                                                    #
#           MOTION RETARGETING IN ROBOTICS           #
#                                                    #
######################################################

Reading files R26 and H26
-------------------------
Read 42441 robot frames and 42441 human frames
Plotting robot and human motion side to side
Ellapsed time 0.5135722160339355

Performing PCA
```

```
Principal components (eigenvectors of the covariance matrix):
[[ 0.88354  0.46750 -0.02824]
 [ 0.46749 -0.88397 -0.00750]
 [-0.02847 -0.00657 -0.99957]]

Variance of the principal directions (eigenvalues):
[0.50767 0.49136 0.00097]
Ellapsed time 1.3029119968414307

Computing period
First aproximation: 113 cycles, estimated period =
367.32142857142856
Number of gaps kept to compute period is 94
Final computed period is 432.77659574468083
 N of cycles: 98.06676335389986
Ellapsed time 1.3448741436004639

Computing 30 sample points in average cycle
Plotting flattened human motion vs sample points and original robot
motion vs sample points

Reconstructing spline curve from sample points
Ellapsed time 3.350341320037842
Plotting reconstructed curve vs sample points
Plotting reconstructed curve vs robot motion
Ellapsed time 4.9632251262664795

Comparisons between human motion and robot motion
-------------------------------------------------
Angle between normal to fitted planes in human and robot:
1.1040758740624441 degrees

Computing robot period
First aproximation: 95 cycles, estimated period = 437.6595744680851
Number of gaps kept to compute period is 94
Computed robot period = 437.6595744680851
Difference from human: 1.128290848307567 %

Computing distance between each reconstructed point and the origi-
nal robot motion
maximum error = 0.012942634310439136 ;  average error =
0.006985410423384942
Ellapsed time 69.83267831802368
```