



# Dance movement patterns recognition (part II)

Computer Science Final Project Report  
made by

Jesús Sánchez Morales

and directed by

Anton Nijholt and

Dennis Reidsma

Enschede, 02 of February of 2007



El sotasignat, Jordi Carrabina

Professor de l'Escola Tècnica Superior d'Enginyeria de la UAB,

**CERTIFICA:**

Que el treball a què correspon aquesta memòria ha estat realitzat sota la seva direcció per en Jesús Sánchez Morales

I per tal que consti firma la present.

Signat: .....

Bellaterra, 09 de Febrer de 2007



Who signs, Anton Nijholt

Electrical Engineering, Mathematics and Computer Science Department at the  
University of Twente teacher,

**CERTIFIES:**

That the work, explained in this report, has been done under his management  
by Jesús Sánchez Morales

And to certify it, he signs this document.

Signed: .....

Enschede, 09 of February of 2007

## Acknowledgments

There are many people whom I should say thanks to give me the opportunity to do this project here. Unfortunately I cannot quote everybody but I am going to say thanks whom have been for me the most important people during these months.

First I would like to mainly remember Jordi Carrabina and the exchange department of my university to help me with all the processes to come here.

In Enschede I would like to say thanks to the SMIT people who have taught me many important things about the life in Netherlands.

In the University everybody has been very kind, from the secretary to my tutor, but probably I would not have done this project without the help of Dennis Reidsma and Anton Nijholt who have guided me during these months and have shown me the way to finish this project successfully. Moreover I would like to say thanks to Herwin van Welbergen and Ronald Poppe who have not had any doubt to help me always that I have had any problem.

I also would like to remember my family and girlfriend to support me in the difficult moments during this time far from home, and I would not like to forget my Erasmus partner, and above all, friend Israel who has been living with me all this time.

Finally I want to remember my grandfather who unfortunately has left us during this project, but I will never forget him.

# Contents

<b>1. INTRODUCTION</b>	<b>7</b>
1.1. PRESENTATION	7
1.2. GOALS	8
1.3. REPORT ORGANIZATION	9
<b>2. PREVIOUS KNOWLEDGE</b>	<b>10</b>
2.1. PROGRAMMING LANGUAGE (JAVA)	10
2.2. HIDDEN MARKOV MODEL	11
2.2.1. GENERAL EXPLANATION ABOUT HMM	11
2.2.2. OUR HMM	13
2.2.3. HMM CLASS	18
2.3. THREADS	22
2.3.1. CREATED THREADS	22
2.3.2. COMMUNICATION BETWEEN THREADS	24
2.4. USED CLASSES	26
2.4.1. MODIFIED CLASSES	26
2.4.2. NEW CLASSES	26
<b>3. THE SYSTEM BUILT (AI)</b>	<b>28</b>
3.1. HOW WE SHOW THE RESULTS	28
3.2. RECOGNIZE A SIMPLE STEP	32
3.2.1. HORIZONTAL STEP RECOGNITION	32
3.2.2. VERTICAL STEP RECOGNITION	36
3.2.3. TWISTER RECOGNITION	42
3.3. RECOGNIZE A COMPLEX PATTERN	46
3.4. AUTO GENERATE A COMPLEX PATTERN GRAPH	48
3.4.1. GENERAL EXPLANATION	48
3.4.2. PROGRAMMING EXPLANATION	49
3.4.3. EXAMPLE	51
<b>4. THE FAILED RECOGNITION ATTEMPT</b>	<b>53</b>
4.1. HANDS MOVEMENT RECOGNITION	53
4.2. FORWARD AND BACKWARD STEP RECOGNITION	54

<b>5. EXTENDING THIS APPLICATION</b>	<b>57</b>
5.1. DEVELOPING A NEW SIMPLE STEP RECOGNITION	57
5.2. DEVELOPING A NEW COMPLEX PATTERN RECOGNITION	59
<b>6. TEST BENCH</b>	<b>60</b>
6.1. TEST SETUP	60
6.2. SIMPLE MOVEMENT TEST	62
6.2.1. HORIZONTAL TEST (LEFT AND RIGHT STEP)	62
6.2.2. VERTICAL TEST (UP, DOWN AND JUMP)	62
6.2.3. TWISTER TEST	63
6.3. COMPLEX PATTERN TEST	64
6.4. TEST CONCLUSIONS	65
<b>7. CONCLUSIONS</b>	<b>66</b>
7.1. REACHED AND FAILED GOALS	66
7.2. FOUND PROBLEMS	67
7.3. POSSIBLE IMPROVEMENTS	70
7.4. PERSONAL OPINION	72
<b>8. REFERENCES</b>	<b>73</b>

# 1. Introduction

## 1.1. *Presentation*

As we know, the goal of this project has been to recognize patterns in the user dance like for example "left + right + jump", this goal has needed some previous steps because this application did not recognize simple steps like "left step" or "right step". Thus the first goal has been to recognize simple steps, and once we have had these simple steps we have used it to recognize more complex patterns.

After some time trying to find out good models to recognize human movement patterns and trying to find similarities between these projects and our project, we realized that the speech recognition techniques could be very useful for us due to its similarities as to have very general information about the user, and the fact that this information was received in real time. If we look it carefully, the speech recognition and the user movement recognition have the same goal because in both cases we receive some continuous data, and our goal is in both cases to discover the meaning of this data. Obviously this data is very different, but the essence is the same.

The speech recognition mainly uses the Hidden Markov Model (and its variations) to recognize words and sentences, and using it, they have got a high rate of success.

We have used the essence of this model as to recognize simple steps as to recognize complex patterns, doing little changes between these two kinds of recognitions.

Other goal has been to try to do a dynamic application, which means, to update the database of recognized movements easily, for this, we have created some graphs that contains the main information of the movement to recognize, and we also have created tools to modify and create new graphs easily and quickly.

## **1.2. Goals**

Since the beginning our goals has been changing, but in this point our goals are quite limited, and at the end of this report will be able to analyze what of these goals has been gotten and in what case has not been possible.

1. To detect simple movements in the user dance

In this goal we want to detect the simplest user's movement, like left or right steps, only using the visual information received.

2. To detect complex patterns in the user dance.

In this goal we want to detect complex patterns, which means, combinations of simple movements done in a determined order and in a determined time.

3. To Generate pattern references automatically.

In this goal, we want to provide to the user a tool to generate a pattern reference, which later, we will be able to be used to search in a dance.

4. To search patterns in a dance without any reference.

In this goal, we want to find out dance patterns in the user dance automatically (without any reference pattern).



### ***1.3. Report organization***

The organization of this report has been done to understand as easily as it is possible the global way to run of the step recognition function implemented in the Virtual dancer application. Thus, this report has been divided in two main parts:

#### **1. Previous knowledge:**

In this part I am going to explain the basic knowledge required to understand the correct operation of this system.

Here will be explained the theoretical techniques and theoretical knowledge which will be used at the rest of the project like the Hidden Markov Model (used to build the Markov graphs) or how runs a thread in a system like this.

This part does not pretend to explain deeply these techniques, but only give the main idea to understand why we have used it and to explain how we have adapted these techniques to our project.

#### **2. The built system:**

In this part I am going to deeply explain all the developed code, using the techniques explained before, to build a step recognition application.

Here I will explain everything interesting or useful to understand the operation of this application. The goal of this report is that after reading (and understand) it, anybody will be able to continue developing this application or using this code for a future new applications.

The developed code is not inside this report, and I have decided to put it in an appendix part.

## 2. Previous knowledge

### 2.1. Programming language (Java)

To develop this application, we have used 2 programming languages, one for the computer vision part (c++) and other for the Artificial Intelligence and animation part (java). In this part of the project, we are centred in the Artificial Intelligence of our application, thus we only describe a little about java (here we receive the computer vision data directly).

Java is a high-level programming language developed by Sun Microsystems. It is an object-oriented language similar to C++, but simplified to eliminate language features that cause common programming errors. One of the best advantages of java is that his compiled code can run on most computers because Java interpreters and runtime environments exist for most operating systems (Unix, windows, MAC and so on).

Furthermore Java is a general purpose programming language with a number of features that make the language well suited for use on the World Wide Web, and this is one of the most important reasons why this language has become one of the most important programming languages.

For us, to use Java have some advantages:

- We have found classes already done by other people that have help us to develop our application.
- We had already worked with this language and have been easy to understand the already done code.
- There are very powerful tools to use this language and to do easier the work to the programmers.



## ***2.2. Hidden Markov model***

As we know, nowadays the hidden Markov models (HMM) are one of the most successful techniques used for the speech recognition, moreover, HMM is also used to other applications like handwriting, or musical recognition. The HMM give us some techniques to recognize patterns, but we have taken the general idea and we have adapted this model to extract useful information for us.

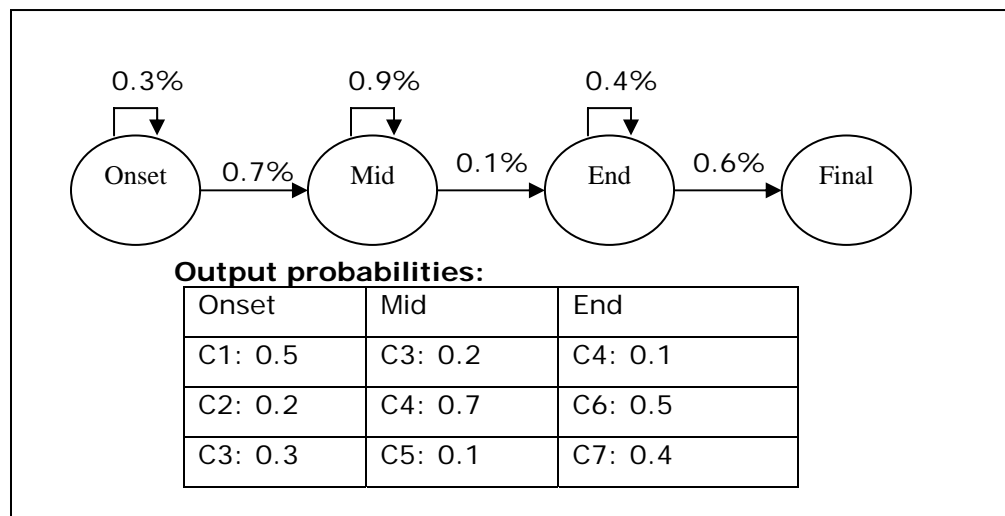
### **2.2.1. General explanation about HMM**

With a HMM we try to determine the value of the hidden parameters of a Markov process using observable parameters, and later, the extracted parameters will be used to do the pattern recognition. Basically a HMM is a very simple dynamic Bayesian network, but enough powerful for us.

The main difference between a regular Markov model and a HMM is that in a regular Markov model the state is directly visible to the observer while in a HMM the state transition probabilities are parameters. Thus, in a HMM the state is not directly visible, but variables influenced by the state are visible. Each state has a probability distribution that returns an output token. We use these output tokens to know information about the sequence of states that has taken place.

To approach the HMM we have used a probabilistic finite automata, this automata have some states (nodes) and transitions (arcs, edges) and as we have said before, each transition causes an output token. To make this automaton, we have to fill two kinds of probabilities; the probability to jump from one state to other (saved in table called Alpha) and the probability to have the observed variable in this state (saved in a table called Beta).

In the figure 2.1 we can see an example of a graph with 3 states (Onset, Mid and End) and the probabilities to jump between one state and another state. We also have the probabilities table (Beta), where we can see in each state the probability of have each variable. For example in the onset state, we can have the value C1, C2 and C3 with the probabilities that we can see in the table.



**Figure 2.1:** example of a HMM automaton

With this automaton and these probabilities we can know the probability of a sequence generated by this graph.

To extract information of this sequence there are basically 3 algorithms: backward-forward, Viterbi and Baum-Welch. Each algorithm solve us different questions about this sequence.

**Backward-forward:** what is the probability of the next sequence?

**Viterbi:** With a HMM and an observable sequence, what is the most probable sequence of states that have generated this observable sequence?

**Baum-Welch:** With a group of states and a collection of observables sequences, what is the most probably HMM that have generated the sequences?

## 2.2.2. Our HMM

Now we are going to explain how we have used the Hidden Markov model to be able to recognize movement patterns and how to interpret the results.

To recognize movement patterns, we have used two classes of graph, one to recognize simple movements (left, right, ...) and other to recognize complex patterns (left + right + jump + ...), both use the HMM but its graphs are different and we also interpret its results different.

### 2.2.2.1. Simple movement HMM graph

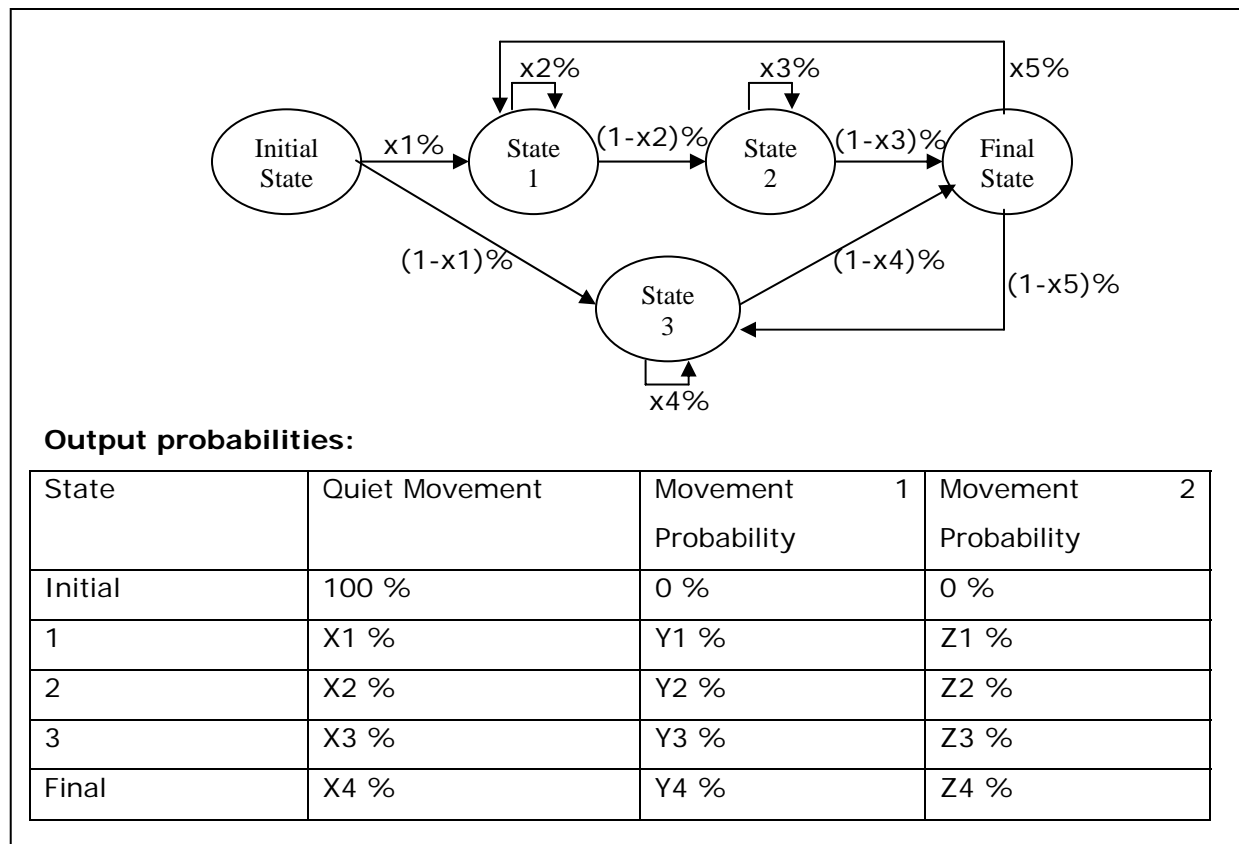
In this kind of graph (figure 2.2), we have an input sequence and we want to know what simple movement is (or are, because an input sequence can contain more than one simple movement).

Our simple movement graphs detect the movements in one dimension, thus we have one graph to detect the horizontal movements (left step and right step) and other for the vertical movements (down, up and jump), the twister recognition, although is a simple movement, follows the complex pattern recognition rules.

We have labelled each path in the graph and depending what path has taken our sequence, we know what movement has been realized, thus, if the sequence takes: **State1 -> State2** in the graph of the figure 2.2 we can translate it as some movement, and if the sequence takes the **State3** it is translated like a different movement.

#### Structure of the graph:

- Our graph have an unique initial state (state 0), that only can have the QUIET value, and therefore we force our sequence to begin with this value (as we see in the output probabilities graph).
- We cannot back to the initial state during the sequence.
- We have a final state that indicate us if we have a new step (for example if this sequence is left – right, we know that the left step has finished because we have been in the final state).
- Since the last step we can go to the beginning of all paths.



**Figure 2.2:** Our HMM graph

### What algorithm to use:

As we have told before, we have 3 possible algorithms to use (backward-forward, Viterbi and Baum-Welch) and each one gives us a different answer.

In our work, we have an input sequence (the information received from the Vision software) and we want to know what path would take this sequence in our graph. For this job the best algorithm is the Viterbi's algorithm because it returns us the sequence of states gone through by the input sequence and this is exactly what we need.

Once we have the sequence of states, we analyze this sequence and we exactly know what simple step has been done.

**Example:**

If we use the graph of the figure 2.2, and we label the first path (state1 + state2) as right step, and the second path (state3) as left step, if the graph receives the next input sequence:

*"Quiet movement + movement1 + movement1+ Quiet movement + movement2 + Quiet movement"*

We could obtain the next sequence of states:

*Initial State -> State1 -> State2 -> Final State -> State 3 -> Final State*

This sequence of states can be divided in two sub sequences (we erase the initial state because has been forced by us):

1. (State1 -> State2 -> Final state) = Right Step
2. (State3 -> Final state) = Left Step

We already know that the user has done a Right Step and a Left Step.

**2.2.2.2. Complex movement HMM graph**

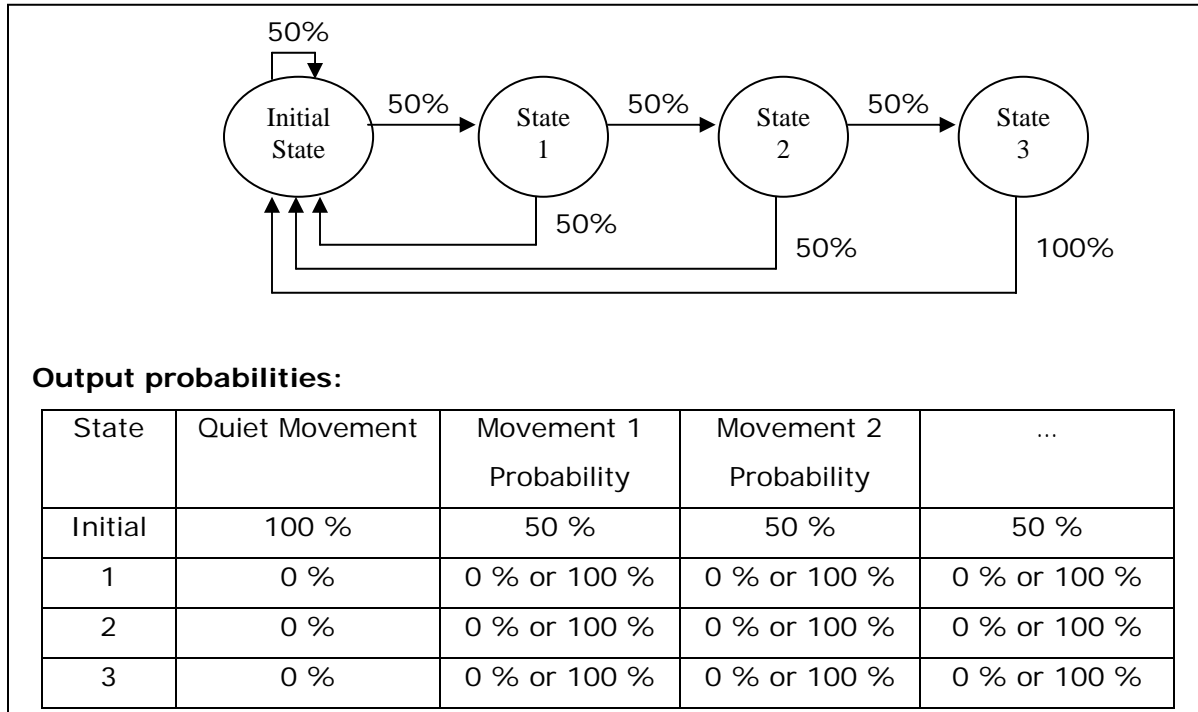
In this case, the graph (figure2.3) is a few different than the simple movement graph, although we have an input sequence to analyze (as well as in a simple movement graph), now; the input sequence is formed by the simple movements recognized by the simple movement graph. Moreover, this graph has a different structure and we also analyze the results in a different way.

Our goal in this kind of graphs is to go through all the states of the graph, and when it happens, means that we have already found the pattern.

**Structure of the graph:**

- The graph have a unique Initial state (state 0), this state can have any value but is the only that can have the Quiet value.
- All the states can back to the initial state.
- Our final state is the last state of the pattern to recognize, thus when we arrive here, we do not have to back to a before step because we have already found the pattern.
- The probabilities to jump are fixed (always 50%).

- Each state (except the Initial state) only can have one kind of variable, thus in the output probabilities table, for each state, one variable must have 100% of probability and the rest must have 0% of probability.



**Figure 2.3: Complex movement graph**

#### What algorithm to use:

To analyze this graph we need the same information than with the simple movement graph, because with the sequence of states gone through we can know if we have arrived to the final state. Thus, the algorithm used is exactly the same, the Viterby Algorithm.

When we have the result of this algorithm, we only have to look if we have arrived to the Last State (state3 in the example) and we know if this pattern has been recognized.



**Example:**

If we use the graph of the figure 2.3, we have to arrive to the state 3 to detect a movement pattern (for example "*movement1 + movement2 + movement2*"), if we receive the next input sequence:

*"Quiet movement + movement2 + movement1 + movement2 + movement2 + movement1"*

We could obtain the next sequence of states:

*Initial State -> Initial State -> State1 -> State2 -> State3 -> Initial State*

If we go through this sequence, we arrive at the last state (in this case State3) at the fifth step, and then we can say that we have detected the pattern and to stop to analyze the sequence (but we can follow analyzing to find out if this pattern is repeated).

*Initial State -> Initial State -> **State1** -> **State2** -> **State3** -> Initial State*

Now, we already know that the user has done this movement pattern.

### 2.2.3. HMM Class

To develop our application we have used a new class called HMM, which give us the necessary tools to create the HMM graphs and analyze its.

In this class we have a lot of useful methods and although we have not used all these methods, we have kept it in the code for possible future modifications.

#### 2.2.3.1. Most important methods contained in the HMM Class

##### 1. **public HMM(int numStates, int sigmaSize)**

This method has been de constructor used to create our graphs; the parameters requested are the number of states of the graph and the size of the output vocabulary.

Furthermore, to generate the graph completely we have to fill 3 structures:

- Pi: Array of 1 x "numStates" dimensions. Indicate what states can be an Initial state.
- A: Array of "numStates" x "numStates" dimensions. Indicate the possible jumps between states.
- B: Array of "numStates" x "sigmaSize" dimensions. Indicate the probability to generate each output in each state.

##### 2. **public HMM(int numStates, int sigmaSize, int dimensions)**

This method is a more powerful constructor, in addition to the seen in the previous method; this method offers us the possibility to choose the dimension of the output vocabulary.

The new structure in this method is "v" that is an Array of "sigmaSize" x "dimensions" dimensions.

##### 3. **public void baumWelch(int[] o, int steps)**

This method trains our graph using the Baum-Welch algorithm.

Basically, given an observation sequence "o", it will train this HMM using "o" to increase the probability of this HMM generating "o".

We have not used this method.

#### 4. **public double[][] forwardProc(int[] o)**

In this class, to do the Forward – Backward algorithm has been divided in two parts (forward and backward), and this is the first part (forward).

This method do the calculation of Forward variables " $f(i,t)$ " for state " $i$ " at time " $t$ " for sequence " $o$ " with the current HMM parameters. These calculations are returned in a 2 dimensions array.

#### 5. **public double[][] backwardProc(int[] o)**

This is the second part of the forward-backward algorithm.

This method do the calculation of Backward variables " $b(i,t)$ " for state " $i$ " at time " $t$ " for sequence " $o$ " with the current HMM parameters. These calculations are also returned in a 2 dimensions array.

After calculating the forward and backward variables we can do some operations (we can see this in the attached documentation of the project) to mix these results and obtain a good probability result.

During the project we have not had to do these calculations because the backward-forward algorithm has not been used.

#### 6. **public double[][] viterbi(int[] o)**

This has been one of the most important methods for us because the we have used the Viterbi algorithm to recognize patterns into a sequence.

With this method, given an observation sequence " $o$ ", we find the best possible state sequence for " $o$ " on this HMM, along with the state optimized probability. The result is returned in a 2 dimensions array.

**7. public static HMM kmeansLearner(String data, int N, int steps)**

The Kmeans is a training algorithms, using this algorithm we can improve the probability of success of our HMM.

This method, Given training data, will classify the data into a specified number of states, then using the final classification, calculate initial, transition, and bias probabilities.

The parameters of this method are:

data: The filename of training data file.

N: The number of states.

Steps: The number of iterations.

This method returns a new and improved HMM.

We have not used this method, but can be a very useful method for future improvements in the HMM graphs.

**8. public double hmmDistance(HMM hmm2)**

This method calculates the distance between two hidden Markov models using the Kullback-Leibler distance measure. The models should have the same size vocabulary.

The parameter "hmm2" is the HMM to compare with our HMM. The result is the symmetric distance between our HMM and "hmm2"

We have not used this method.

**9. public int[] observationGeneration(int length)**

This method provides us an observation sequence of a given length using this HMM.

The parameter "length" means the length of the observation sequence, and the returned value is a 1 dimension array with the observation sequence generated.

We have used this method to generate sequences and test our application with this sequences (to use this method is faster than create a sequence by hand).

#### **10.    public void print()**

With this method we can print all the parameters of this HMM including initial, transition, and output probabilities.

This method has been useful to see if our dynamic graphs were well generated.

The result is directly printed on the screen.

#### **11.    public static HMM load(String filename)**

With this method we can load an HMM from a file. A new HMM is created and the read values are stored as the HMM parameters. The returned value of this method is the new HMM.

The parameter of this method "filename", like its name says, is the name of the file to read the HMM.

We have used this method to be able to load all our pre-saved graphs (all our recognized movements have been saved in files).

#### **12.    public void write(String filename)**

With this method we can save a HMM graph in a file. The information written in the file includes the number of states, output vocabulary size, number of dimensions (-1 if not used), initials, transitions, biases, and, if it was used, the output vocabulary.

The unique parameter is the filename, that is, the name and the path of the file that we want create or update.

## 2.3. Threads

One of the most important features of our application is the fact that it works in a real time. For this reason we cannot use a very complex algorithm that slow down the process while it is working (we would leave to analyze a lot of frames).

To solve this problem, we have used threads. A *thread* can be defined as a separate stream of execution that takes place simultaneously with and independently of everything else that might be happening.

For us has been very useful because it has let us to continue analyzing the input data while a thread is looking for patterns in a sequence.

### 2.3.1. Created threads

The application already ran with some threads, but only we are going to explain the threads created to detect patterns and movements.

Basically we have created 4 kinds of threads (each thread has a different goal and it is launched different times and in different moments).

#### 1. Move Detector thread

This is the MoveDetector class, the goal of this thread is to detect the possible simple movements and send it to another thread to be analyzed.

This thread is launched in the beginning of the application, and it is always receiving the computer vision data (CVC). It is launched only one time and it is destroyed when the application is closed. In this thread we create the sequence that is sent to the HMM, thus here is where we read the data from CVC, we detect a possible simple movement, we create the sequence to send to the HMM to recognize this movement and we launch the HMM thread with this sequence.

The threads launched by the MoveDetector thread are HorizontalMoveDetector, VerticalMoveDetector and TwisterMoveDetector.

## 2. Simple Movement thread

This kind of thread is done to analyze sequences and to recognize simple steps (left, right, jump ...), thus here is where we use the Viterbi's algorithm and where we analyze the result to know if the simple movement has been done.

In this kind of thread we have included the HorizontalMoveDetector, VerticalMoveDetector and TwisterMoveDetector classes. These threads are different because each one has to recognize different data, but, all have the same input (a sequence of data) and have to recognize simple movements.

These threads are launched as times as we think that we have detected some movement, thus we can have the same thread running many times at the same time.

Now, the movements recognized by these classes are:

- HorizontalMoveDetector: Left and Right.
- VerticalMoveDetector: Down, Up, Jump.
- TwisterMoveDetector: Twister.

These simple movements are sent to the pattern detector thread to detect complex movements.

## 3. Receive simple movements thread

This thread is the ReciveStep class. It is launched only one time at the beginning of the application (at the same time than the move detector thread) and its function is to receive the simple steps, recognized by the simple movement threads, and to try to detect some complex pattern.

This thread does not recognize complex patterns, but it creates the sequence to be sent to the thread that is going to recognize the complex pattern.

#### **4. Complex Movement thread**

This thread is the PatternMoveDetector class. Here we recognize pre-saved patterns.

The function of this thread is very similar to the Simple Movement thread, because we receive a sequence of data and we have to apply the viterby's algorithm to analyze the result of this algorithm. When we launch this thread, we choose what pattern we want to recognize.

This is the last thread to be launched, therefore, this thread does not launch any other thread and do not send data.

### **2.3.2. Communication between threads**

In our application we have need to communicate the threads. We have used two ways to send information between threads.

The simplest way is to send this information is as parameter when the thread is created. For example, if we want to recognize a movement in a sequence, we create the recognizing thread and we send this sequence as a variable. This way has been very useful and all our threads receive some information as parameter.

Although the explained way is very simple and useful, it does not solve all our problems because some times we have to receive information while the thread is already running. For these cases we have used sockets.

We could define a socket as one endpoint of a two-way communication link between two programs running on the network, or in our case, between two different threads of the same application. A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent; in our case we have used the port 7004.

We have only used this technique to send the simple steps to the thread that looks for complex patterns, because we have one thread always waiting for new simple



steps and another threads sending the recognized simple steps. Thus, we have one thread listening to the port 7004 and many threads sending information to this port.

Classes which receive data from the socket:

- ReciveStep

Classes which send data to the socket:

- HorizontalMoveDetector
- VerticalMoveDetector
- TwisterMoveDetector

## ***2.4. Used classes***

To do this project, we have had to modify and add some new classes to the existing java project. Each new class have a different purpose; now will be showed these classes.

### **2.4.1. Modified classes**

#### **1. CVClient.java**

This class was already done by the before developers, and it was been used by other application functions.

Basically, the received objects from the CV part are of this class. This class has been modified to receive new information necessary to develop the new features of the application.

#### **2. CVInfoFrame.java**

This class has been modified to save the new information used in the application, its methods have also been modifying to show and take into account this new data.

#### **3. Test.java**

This is the executed class, and here is where we have launched our new recognition code.

### **2.4.2. New classes**

#### **1. HMM.java**

This class contains all the HMM algorithm and functions used to recognize movements (the most important features of this class has been seen in the HMM part).

#### **2. HorizontalMoveDetector**

This is a new class which receive the horizontal movement data, and look for a possible simple movement in this data.

#### **3. MoveDetector**

Here is where we receive all the computer vision data, here we also process this data and we send it to the class which is going to look for the simple movement.

#### **4. PatternGraphGenerator**

In this class we generate a reference pattern to be searched in the future.

#### **5. PatternMoveDetector**

This is the class which detects some pattern in a sequence of simple movements.

#### **6. ReciveStep**

Here we receive the simple steps already recognized, we process it, and we send it to the class which is going to look for complex patterns.

#### **7. UserStep**

This class has been created to save all the information of the simple steps done by the user.

#### **8. VerticalMoveDetector**

Here we receive information of the verticals user's movements and we look for possible vertical steps.

### 3. The built system (AI)

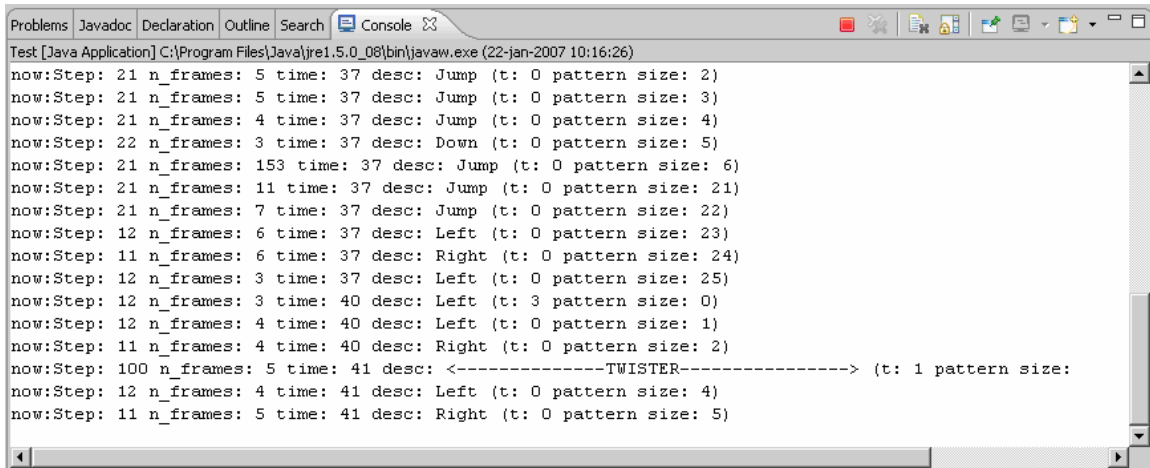
In this chapter we are going to deeply explain how has been done and how runs the recognizing code. If the your goal is only to create a new simple recognition or to create a new complex pattern to be recognized later to extend the application we suggest to only read the chapter 5, where we have exactly explained what part of the code we have to change to create new recognitions easily but without explain why we have to change this code.

To develop this project, we have used the test application of the global Virtual Dancer application. That means that to launch our application is necessary to launch the test application (test.java). Thus, in the test.java code, we have included the code to launch the simple move detector thread and the complex pattern detector thread.

#### ***3.1. How we show the results***

Our goal has been to recognize simple steps and complex patterns. In the future this information will probably be used to improve the dance skills of the virtual dancer, but, nowadays we only want to do a step in front of the webcam and see that this step has been well recognized.

For this reason, we have decided to show the recognized steps in the console window (doing a simple “printf” to show the result as we can see in the figure 3.1).



The screenshot shows a Java IDE window with a console tab active. The console displays the output of a Java application, showing a sequence of steps with associated data. The output is as follows:

```
Test [Java Application] C:\Program Files\Java\jre1.5.0_06\bin\javaw.exe (22-jan-2007 10:16:26)
now:Step: 21 n_frames: 5 time: 37 desc: Jump (t: 0 pattern size: 2)
now:Step: 21 n_frames: 5 time: 37 desc: Jump (t: 0 pattern size: 3)
now:Step: 21 n_frames: 4 time: 37 desc: Jump (t: 0 pattern size: 4)
now:Step: 22 n_frames: 3 time: 37 desc: Down (t: 0 pattern size: 5)
now:Step: 21 n_frames: 153 time: 37 desc: Jump (t: 0 pattern size: 6)
now:Step: 21 n_frames: 11 time: 37 desc: Jump (t: 0 pattern size: 21)
now:Step: 21 n_frames: 7 time: 37 desc: Jump (t: 0 pattern size: 22)
now:Step: 12 n_frames: 6 time: 37 desc: Left (t: 0 pattern size: 23)
now:Step: 11 n_frames: 6 time: 37 desc: Right (t: 0 pattern size: 24)
now:Step: 12 n_frames: 3 time: 37 desc: Left (t: 0 pattern size: 25)
now:Step: 12 n_frames: 3 time: 40 desc: Left (t: 3 pattern size: 0)
now:Step: 12 n_frames: 4 time: 40 desc: Left (t: 0 pattern size: 1)
now:Step: 11 n_frames: 4 time: 40 desc: Right (t: 0 pattern size: 2)
now:Step: 100 n_frames: 5 time: 41 desc: <-----TWISTER-----> (t: 1 pattern size:
now:Step: 12 n_frames: 4 time: 41 desc: Left (t: 0 pattern size: 4)
now:Step: 11 n_frames: 5 time: 41 desc: Right (t: 0 pattern size: 5)
```

**Figure 3.1:** Console Windows where the results are shown

During our tests, we began leaving the visual part in the background because we only have to watch the console window, but finally we decided to disable the Visual part (where we can see the virtual dancer dancing) because this function consumes a lot of resources.

When we show the information, it is different shown depending on the kind of recognition done:

- Simple steps:

"now: Step: **X** n\_frames: **Y** time: **Z** desc: **M** (t: **N** pattern size: **L**)"

**X**: Code of the recognized step.

**Y**: How long (in frames) has been the step.

**Z**: Seconds after the beginning of the application that the step has been recognized.

**M**: Name of the recognized step (left step, jump, twister, ...).

**N**: Seconds since the last recognized step.

**L**: Actual size of the sequence list built to detect a complex pattern.

To know the name of each step, we have labelled each possible step with a code (the recognizing application send us this code) and we have created structures to translate this code to the name of the step.

To do our work easily (and to let an easily update) we have labelled the steps according to its recognizing thread. After this, we have used a HashMap to translate this code (11, 12, 21, ...) to a sequential number (1, 2, 3, ...) which we will also use to detect patterns..

Finally we have an array of strings that in each position contains the name of the step with this code.

Horizontal steps:

11 -> 1 -> Right step

12 -> 2 -> Left step

Vertical Steps:

21 -> 3 -> Jump

22 -> 4 -> Down

23 -> 5 -> Up

Twister step:

100 -> 6 -> Twister

- Complex patterns:

"Thread T: <----- **X** ----->"

X: Name of the recognized pattern.

In this case, as name of the pattern we have used the name of the file that contains the pattern graph (these files has been called patternXX where XX is a sequential number).

### ***3.2. Recognize a simple step***

As we have seen before, the simple step recognition has been divided in three parts (horizontal steps, vertical steps, and twister). For each one of these recognitions we have used different input data, and obviously we have had to adapt this data in a different way.

Some of the things done in these recognitions have been explained in the *previous knowledge* part (for this reason, to understand this explanation is very important to read before the *previous knowledge* part).

#### **3.2.1. Horizontal step recognition**

In this kind of recognition we recognize Left Steps and Right Steps.

To do this recognition we only use the variation of the horizontal (x) position of the centre of mass and the user's radius. Transforming this information we can detect exactly what movement has been done. In the figure 3.2 we can see how the centre of mass position changes according to the step done.

The first step of this recognition is to translate this X variation to a useful data (X Variation is the number of frames between the position of the centre of mass in one frame and its position in the next frame).

We know that if the X Variation is positive ( $>0$ ) the user has done a right movement, while if the X Variation is negative ( $<0$ ) then the user has done a left movement (a right or left movement does not mean a right or left step because a user never is completely quiet and we could detect the body swing).

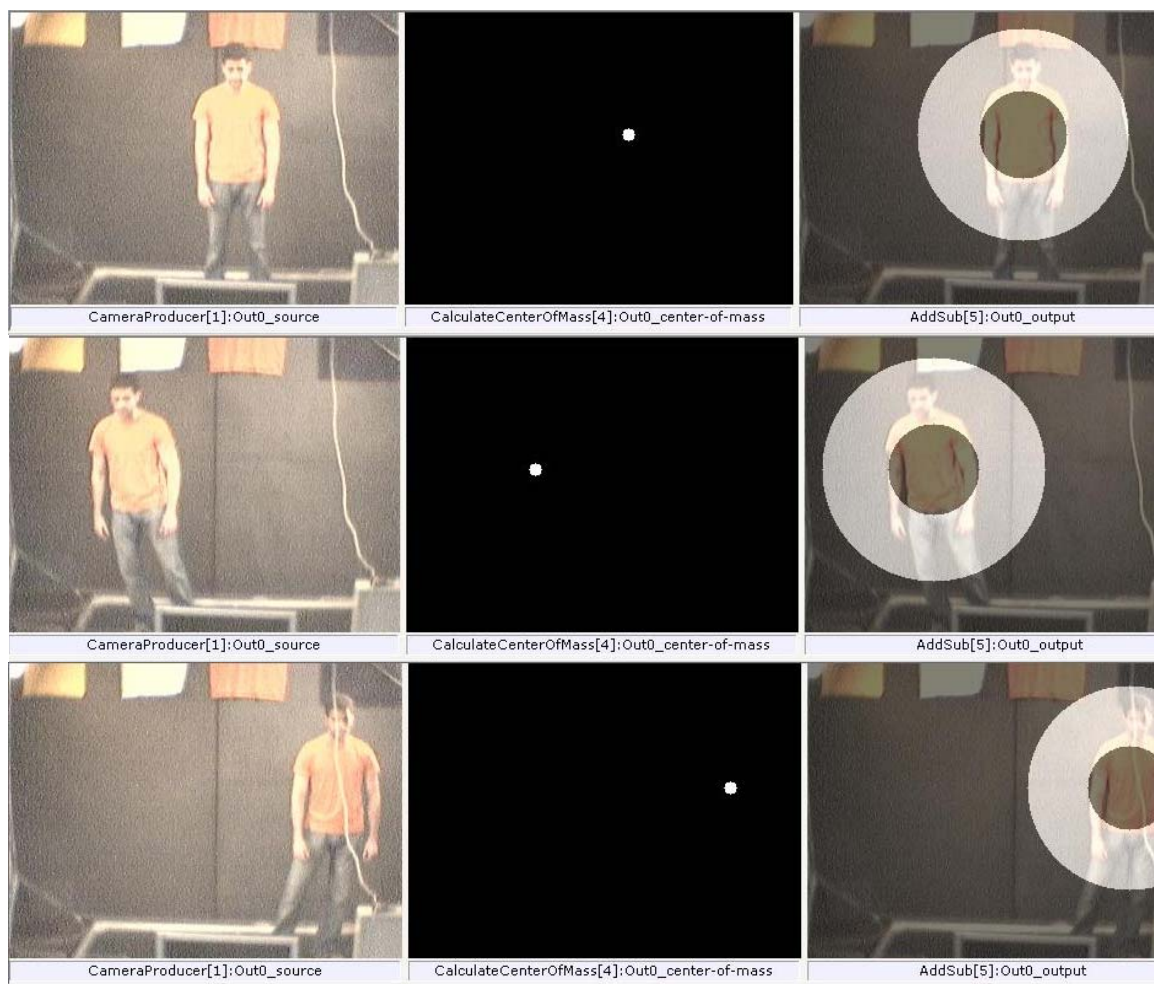
To eliminate the body swing we have decided that if this movement is lower than the 10% of the user's radius, it is discarded ( $= 0$ ).



When we have filtered the movement to eliminate the body swing, we translate this variation to 4 more useful states:

- 1: right
- 2: very right
- 3: left
- 4: very left

Now, as input information we have 4 possible values (1, 2, 3 and 4) and each value has a different meaning (right, very right, left and very left).



**Figure 3.2:** X Centre of mass variation

The second step is to create a sequence to be sent to the HMM graph which have to recognize the steps in the sequence. To create this sequence we have to detect when is possible to have some step. To do it, we have counted the frames with movement

(X variation  $\neq 0$ ) and the frames without movement (X variation = 0). The main idea is that if we have detected movement during more than 1 frame and after this we have had 5 frames without movement, in this frames we can have some step to recognize. Thus, we can send this sequence to analyze and to begin to the sequence again.

This sequence is a list of integers called horizontalO, in this list we add the new variations (if this variation is different to zero) and finally, when we send the sequence we clear it to add new values again.

Our HMM graph has already been built to run with these values (1, 2, 3 and 4).

Finally, before to send this sequence with possible steps we have to put at the beginning of the sequence a Quiet value (0) because our HMM graph force us to put this value (it has been explained in the HMM part).

The sent sequence must be an array of integers, then, as we already know the length of our sequence, we can create an array of integers of this size and we can fill it with the values of the horizontalO list (with the value 0 at the beginning).

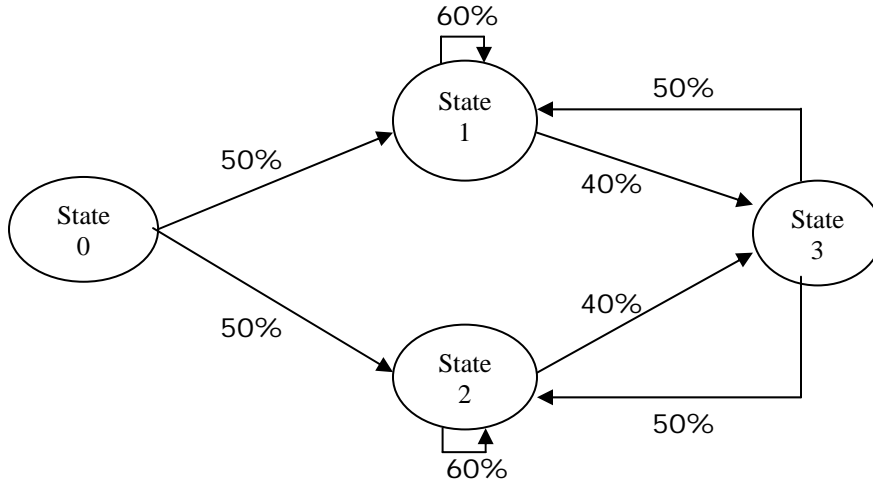
Now we can launch the HorizontalMoveDetector thread passing this sequence as parameter.

The third step is done by the HorizontalMoveDetector thread, which is working while the MoveDetector fills a new sequence. This thread is which loads the HMM graph from a file (this file is saved as "c:\\grafos\\horizontalMov.txt").

When we have loaded (and created) the graph, we can analyze the input sequence using the Viterbi's algorithm.

This algorithm returns us the list of states generated by this graph. Now we have to go through this list and detect what steps has generated (how to recognize the steps inside this list is also explained in the HMM part).

In the figure 3.3 we can see the horizontal HMM graph, this is a very simple graph because the variables only have 4 possible values, and we only have to recognize 2 different steps.



**Output probabilities:**

State	Quiet Movement	Right Movement	Very Right Movement	Left Movement	Very Left Movement
0	100 %	0 %	0 %	0 %	0 %
1	5 %	50 %	40 %	6 %	4 %
2	5 %	6 %	4 %	50 %	40 %
3	60 %	10 %	10 %	10 %	10 %

**Figure 3.3:** Horizontal HMM graph

### 3.2.2. Vertical step recognition

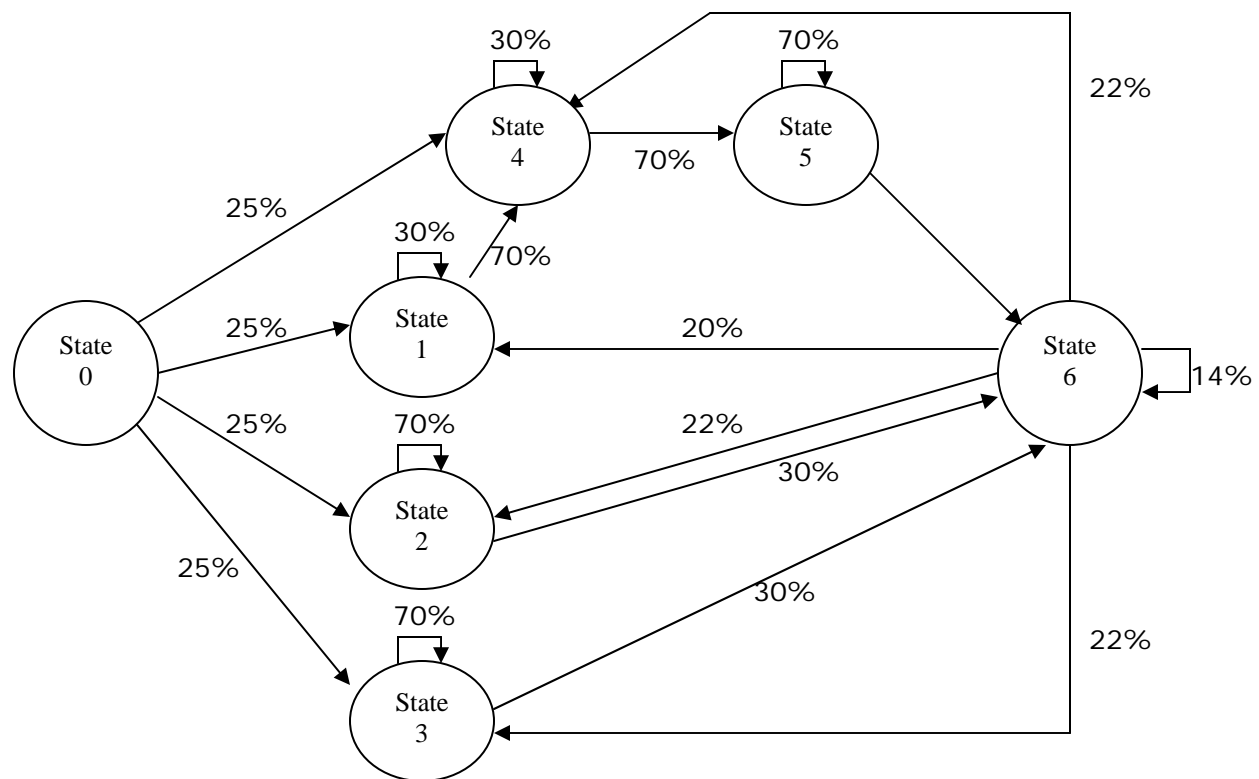
In this kind of recognition we recognize the jump, down and up step.

When we began to do this recognition we tried to use the same method than the horizontal recognition (to use the vertical (Y) variation of the centre of mass) and we did a first version using this technique, but after a lot of tests we realized that it was not enough reliable because the vertical movements are too fast for our application. For example, when the user jumps, he is very few time in the air, and our application some times does not detect this variation (we have a limited frame rate, and we cannot change it), usually we detect that the centre of mass go up or go down, but not both, and to recognize a jump our application must detect that the centre of mass has go up and go down in a certain period of time, in other case we cannot know if the user is jumping or downing (in booth cases the centre of mass go down in some moment).

Moreover this technique have another problem, in the horizontal recognition does not matter where is de user, if the centre of mass goes to the left, it is a left step and if the centre of mass goes to the right, it is a right step, but, in the vertical axes, is not the same to go up if the user is in a normal position (he would be jumping) than if the user is ducked (he would be going back to the normal position).

For these reasons we decided to look for other input data and other techniques.

Although this technique has been ruled out, we have done a previous version and we have built the vertical HMM graph for this technique (we can see it in the figure). We can see that this graph is very complicated because we tried to build an accurate graph to correct the Y variation problems.



**Output probabilities:**

State	Quiet Movement	Down Movement	Very Down Movement	up Movement	Very up Movement
0	100 %	0 %	0 %	0 %	0 %
1	25 %	55 %	30 %	6 %	4 %
2	25 %	80 %	85 %	6 %	4 %
3	30 %	6 %	4 %	40 %	45 %
4	40 %	6 %	4 %	40 %	45 %
5	35 %	60 %	25 %	15 %	4 %
6	60 %	10 %	10 %	10 %	10 %

**Figure 3.4:** Old version of the vertical HMM graph

After studying some other techniques, we decided to use the centre of mass position (the exactly position, not the variation).

The main idea of the new technique is to know the centre of mass position when de user is in a normal state (quiet), and compare this position with the actual position.

Now we can deduce that if the actual position is over the normal position, the user is jumping, and if the actual position is under the normal position, the user is ducking.

In the figure 3.5 we can see how the Y centre of mass changes if the user is quiet, jumping or ducking.

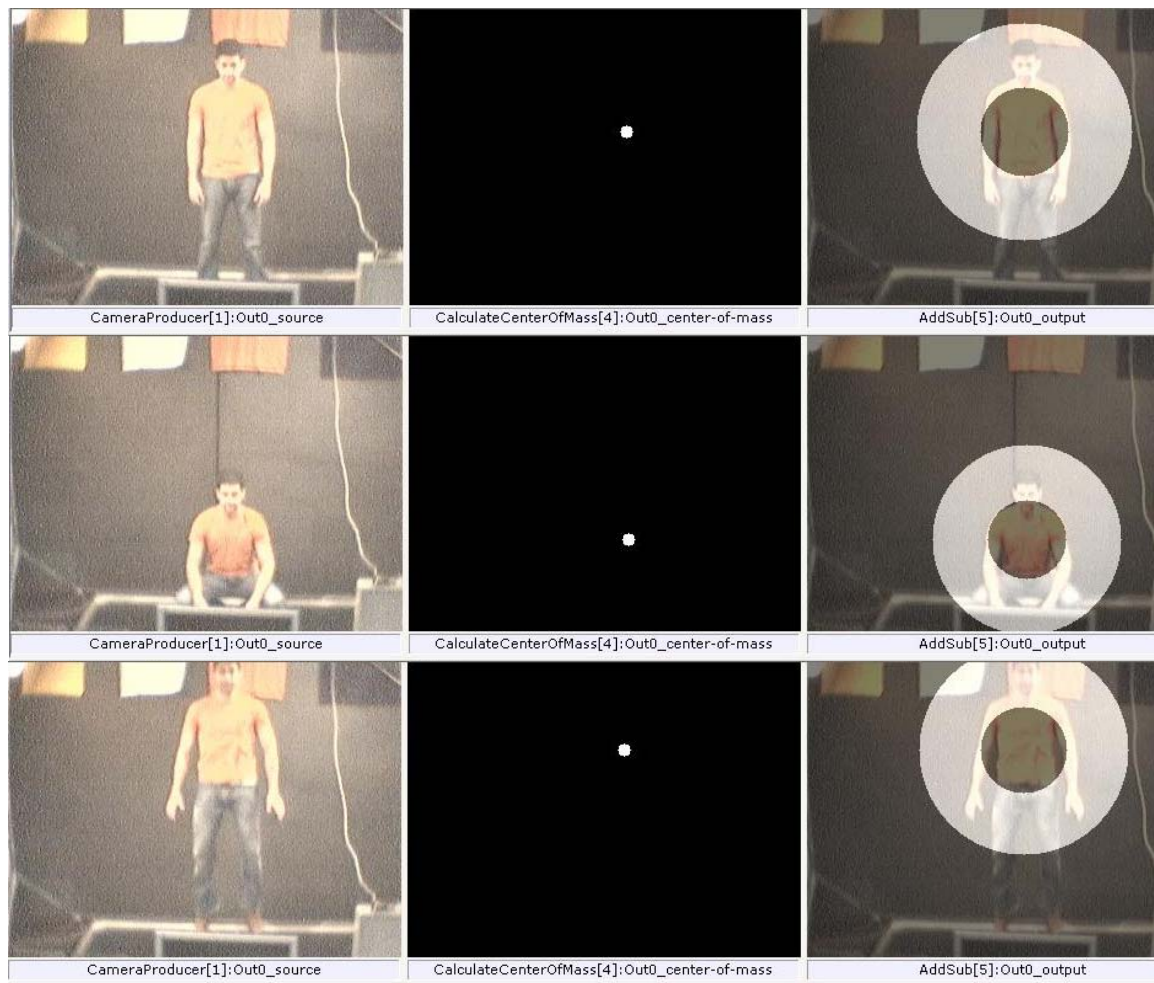
This new technique has one important problem that is to know what the normal user Y position coordinate is, because these coordinates can change during the dance (for example, if the user do a front step, this coordinate changes), for this reason we have created a function that calculates every time the normal centre of mass Y coordinate.

Then, the first step of the vertical recognition is to calculate (and update every frame) the normal user Y position coordinate.

To calculate this coordinate we save the last 50 Y positions (without null values) and we calculate the average of this positions. These 50 positions are saved in an Array of integers which is updated every frame (the oldest value is changed by the actual value).

With this method the Y normal position is quickly updated (the changes usually are very little) and gives us a quite good approximation of the real centre of mass of the user.

This technique implies that the application, when the user is detected by the web cam, needs a few seconds to calculate this centre off mass (we usually work with 15 frames per second, thus in 3 seconds we already have this position calculated).



**Figure 3.5:** Y variation during a vertical movement

The second step is to translate this information (the normal Y position and the actual Y position) in useful information. For this purpose we look if the actual Y position is over the normal Y position (it means a jump or an up) or if the actual Y position is under the normal Y position (it means a down).

To avoid the body swing we have also filtered this input, similar than in the horizontal recognition, but in this case, we only consider a vertical movement when the variation is larger than a 10% of the normal Y position.

When we have filtered the variation to eliminate the body swing, we translate this variation to 3 more useful states:

0: normal

1: down

2: up

The third step is to create the sequence that have to be sent to the HMM graph, which will recognize the possible movements.

This sequence is created with the same method than the horizontal sequence. This sequence is also a list of integers, in this case called verticalO, and here we are adding the new done movements (normal, down and up).

In this case, we have had to add a little code to improve the jump recognition. It is because when a user jumps, sometimes our Vision system does not capture the falling part of the action, then if we detect it, like we know that the user has been over the Y normal position, we automatically put the down value after the up value in the sequence, obviously this value is only added if the maximal Y position detected is over the maximal Y normal position during this step (in this way we avoid problems with the up step).

To decide when we must send this sequence, we also wait for a pause (like in the horizontal recognition), but in this case of more than 3 frames (more than 3 frames in normal state), and then we prepare to copy this sequence to an integer array and we launch the thread which is going to recognize the possible movements.

When we are copying the list into the Integer array, we also force to put a 0 value at the beginning of the sequence (this requirement is forced by the HMM graph).

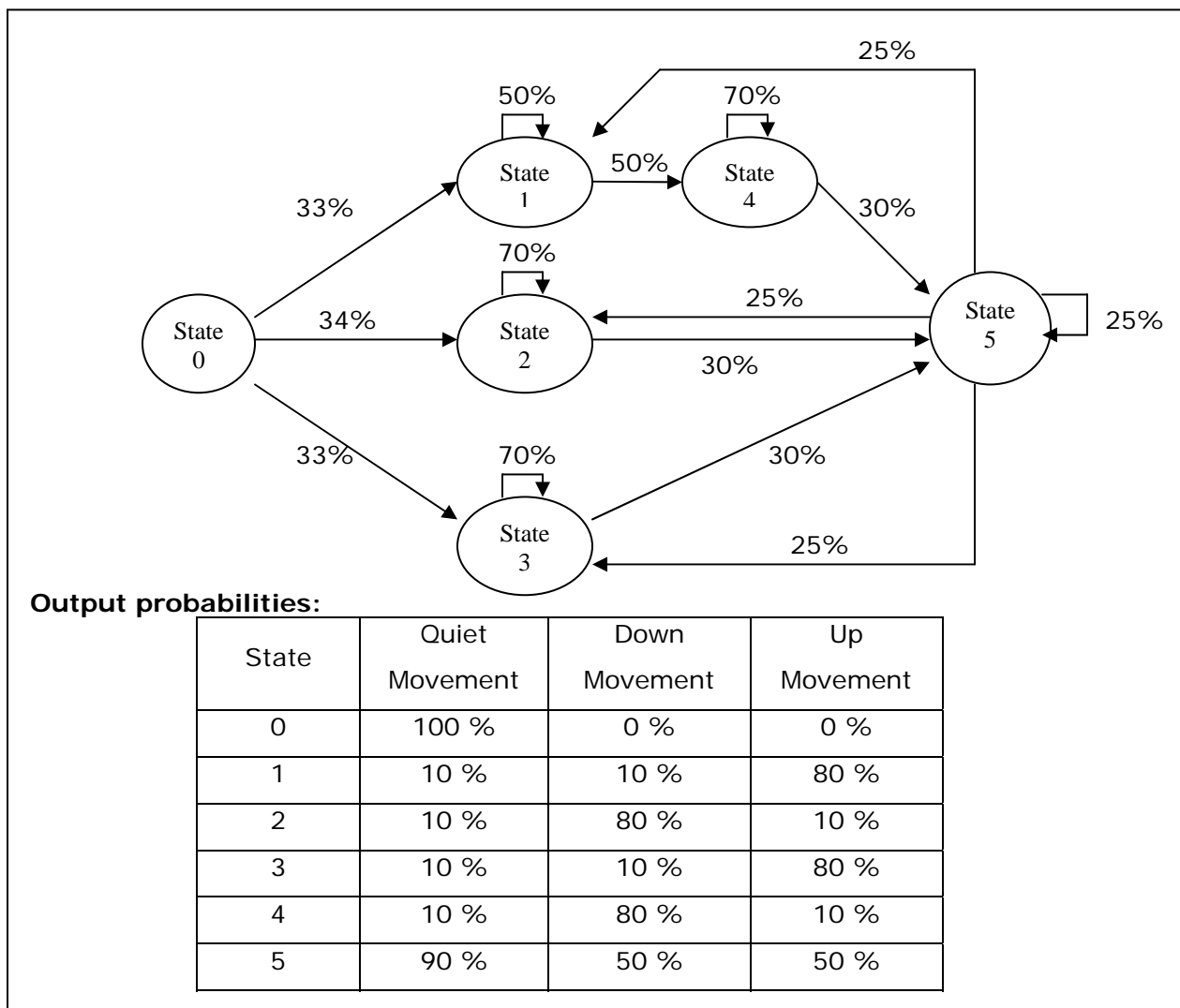
This sequence is passed as a parameter to the launched thread (verticalMoveDetector).

The forth step is done by the verticalMoveDetector thread. It loads the HMM graph (saved as "c:\grafos\verticalMovV2.txt") and analyze the input sequenced (received as a parameter) using the Viterbi's algorithm.



This returns us the list of states generated by this graph again. Now the process is the same than in the horizontal recognition.

In the figure 3.6 we can see the vertical HMM graph (the newest version) and we can see that is easier than the old version graph but more complex than the horizontal HMM because we have to recognize more steps.



**Figure 3.6:** Newest vertical HMM graph

### 3.2.3. Twister recognition

In this recognition, as its name says, we are going to recognize if the user has done a twister (a complete turn).

To do this recognition we have basically used the radius of the user, he have detected that when a user does a twister, his radius follow a determinate variation.

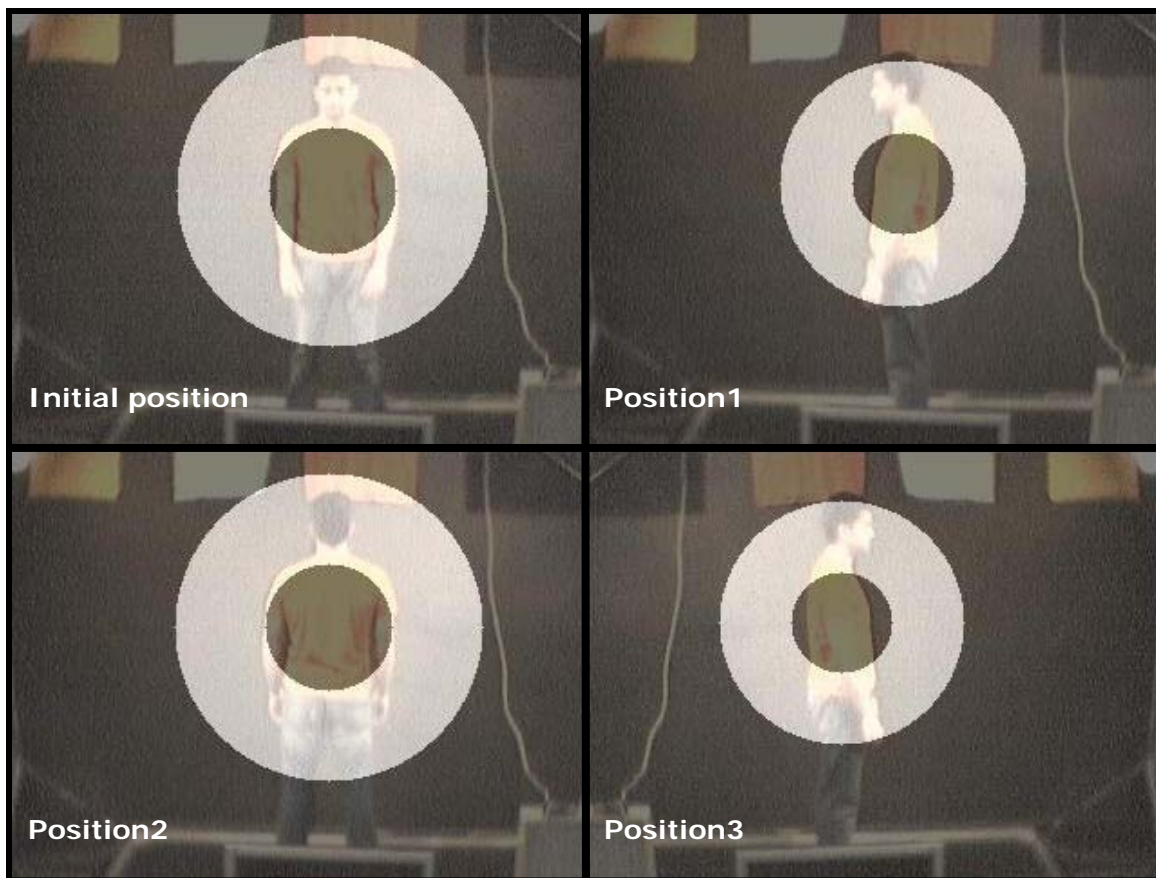
When the user is looking at the cam, the radius has the biggest size this radius is decreasing until the user has rotated  $45^\circ$  (position1), then the user's radius has the smallest size (position2), the user follows rotating until he is backward to the webcam and the radius is increasing until arrive to the biggest size another time (position3), now the user rotate until  $135^\circ$ , then the radius size is decreased and we get the smallest position another time (position4), and finally we back to the start position where we recover the biggest size (position5).

In this case, we have also calculated the average radius size, to compare the actual radius with this average radius, and then we can know if the radius size is increasing or decreasing.

The technique followed to calculate this average radius is the same used to calculate the normal Y position in the vertical recognition. We save the last radius sizes (in this case 100 last frames) and we calculate the average of this radius. This average is recalculated each frame.

Obviously at the beginning of the twister, the average radius and the actual radius will have the same size, but in the position1, the actual radius will be smaller than the average radius, when we go to the position2, the average radius size will be similar than the initial position and we will see that the actual radius is equal than the average radius (or a little bigger if the twister is done very slow), and this sequence will continue until the start position.

In the figure 3.7 we can see the radius size in each one of these positions and we can easily see the different size according to the user position.



**Figure 3.7:** Radius variation during the twister movement.

Then in each position we obtain the next result:

Initial Position	-> average radius = actual radius
Position1	-> average radius > actual radius
Position2	-> average radius = actual radius
Position3	-> average radius > actual radius
Position4	-> average radius = actual radius

Then the first step of the twister recognition is to calculate the average radius (normal radius for us). This implies that this recognition system also needs few seconds at the beginning to calculate this normal radius, while this seconds the results are not correct (in this case, working with 15 frames per second, we need 6 seconds to do this calculations).

The second step is to normalize these values and eliminate the body swing, for this, we have eliminated the variations lower than a 10% of the actual radius (in this case we consider that has not been any movement).

To detect the twister we only need to know if the actual radius has the same size, has a bigger size or has a smaller size of the normal radius. Then we transform the input data to 3 possible results:

0: Normal (Actual Radius = Normal radius)

1: Smaller (Actual Radius < Normal radius)

2: Bigger (Actual Radius > Normal radius)

The third step is to create the sequence that has to be sent to the HMM graph, which will recognize if a twister has been produced.

To build this sequence we have used the same method than in the before recognitions. We have used a list of integers called `twisterO` and we have been adding the new values until that we have detected a pause in the radius variation, and then, we have sent this sequence to the twister recognition thread.

In this case we have waited to a pause of 15 frames (the user's radius is not usually changing), because a twister can be done very fast or very slow. For this reason, the twister recognition is a bit slow than the recognitions explained before.

To improve the application we have had to restrict the twister recognition. We only recognize a twister when the user is stopped (the user does not have horizontal or vertical movements). This restriction is because when the user is walking and he rotates his body can change the radius size and, randomly, he can generate the twister radius variation sequence. We have preferred to do not detect twister in movement than detect also a false twisters.

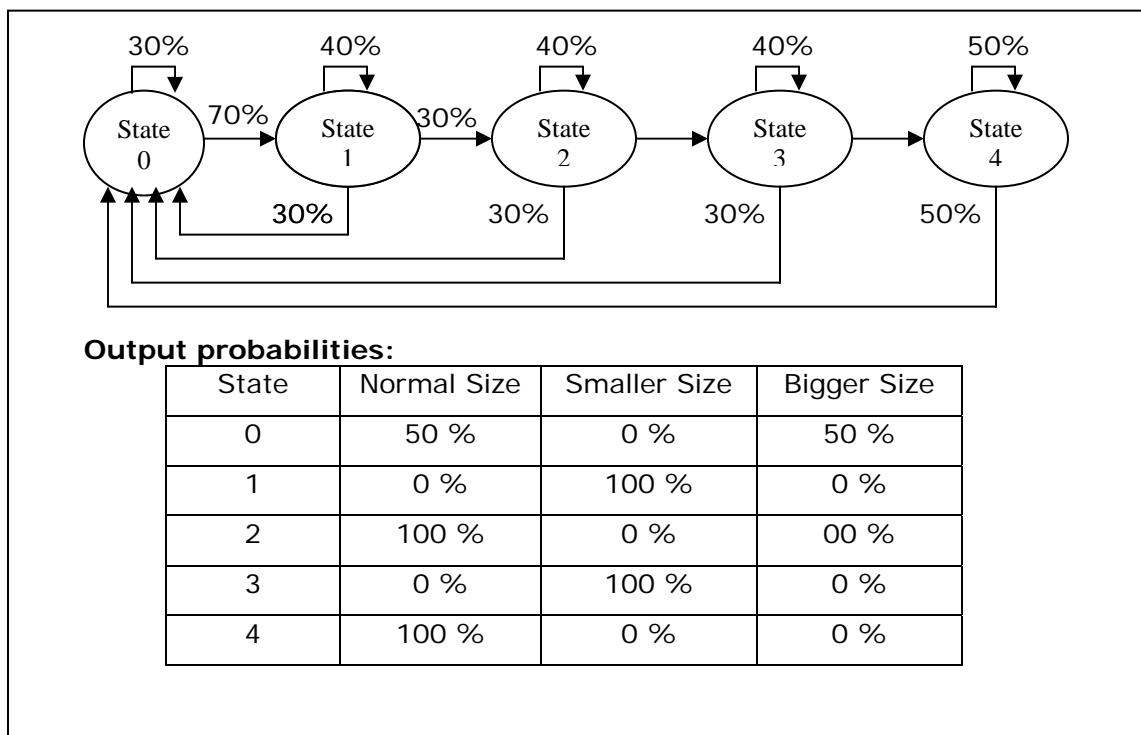
Now, we send this sequence to the Twister recognition thread.

The fourth step has place in the `twisterMoveDetector` thread. First it loads the HMM graph (saved as "`c:\grafos\twisterMove.txt`"), after that, it analyzes the

input sequence (like in the other threads, received as a parameter) using the viterbi's algorithm.

This graph is different than the other simple recognitions graphs, now we only have one path, and we have to go through all the states to recognize the twister, thus we only can say that a twister has been produced when we arrive to the final state.

In the next figure, we can see the HMM graph used to recognize the twister movement. This is the simplest graph because we only have to recognize one movement.



**Figure 3.8:** Twister HMM graph

### ***3.3. Recognize a complex pattern***

For us, a complex pattern is a sequence of simple steps (vertical and horizontal movements and twisters) in a determined order that forms a dance choreography. Our goal is to detect if the user has done these steps and to detect if these steps have been done in the correct order.

Now, that we already have the simple step recognized, as has been explained in the "Communication between threads" part, we receive all these recognized steps in our thread.

The class created to do this recognition is the "recvStep" class. This class has a similar function than the moveDetector class, thus here we are going to create a sequence of steps that will be sent to another thread to be analyzed by the Viterbi's algorithm.

In the recvStep class we are every time waiting for recognized simple steps, here we receive userStep objects, which contain the main information of this simple step.

The userStep class has been made by us, and its main purpose is to save easily a recognized simple step.

In a userStep object we have the next data:

1. Step: Code assigned to this step (each simple step has a different code).
2. n\_frames: How long is this step (in frames).
3. time: How many seconds after the beginning of the application has been done this step.
4. step\_description: Name of this step.

When we have received these simple steps, we treat each simple step separately.

The first step to do with each one of these simple steps is to show this step in the console window (here is where we show to the user the simple step recognized), to do it, we fill a string with the data to show (this data has been explained in the "How we show the results" part) and we print this string on the screen.

The second step is to find too large steps and divide it in some smaller steps. For example, if we receive a left step of 50 frames, it means that the user has been going to the left during 50 frames, but probably he has walked and he has done some steps, then we divide this large step in smaller steps (in small steps of 10 frames).

The third step is to put these simple steps (already divided if it was necessary) in the sequence list that will be sent to the HMM graph.

A sequence of steps is sent to the HMM graph when have been 2 seconds without any simple step. To do it, we have a socket waiting for new steps, but when it has been waiting during 2 seconds, it create a default userStep object, which have the number of frames as 0 to know that this step has been auto-generated, and when we receive this default step, we know that we have to send the actual sequence (obviously if this sequence is not empty).

The last step of this class is to launch a pattern recognition thread with this sequence and the pattern that we want to recognize.

If we want to recognize more than one pattern, we have to launch as threads as patterns we want to recognize, each thread with the same sequence and a different pattern to recognize.

Now, the patternMoveDetector thread (the thread launched) must load the HMM graph that contains the pattern (we have received the name of the pattern to recognize as a parameter and we load the file with this pattern information).

When we have loaded the graph, we have to analyze the also received sequence with the Viterbi's algorithm and detect (as has been explained in the HMM part) if this pattern is in this sequence.

In this case we cannot show a HMM graph because each pattern have his own HMM graph.

### ***3.4. Auto generate a complex pattern graph***

The last function done has been to be able to generate the graph of a complex pattern automatically.

Our goal has been to do some pattern (a sequence of simple movements) in front of the camera and save the graph which would recognize these movements automatically.

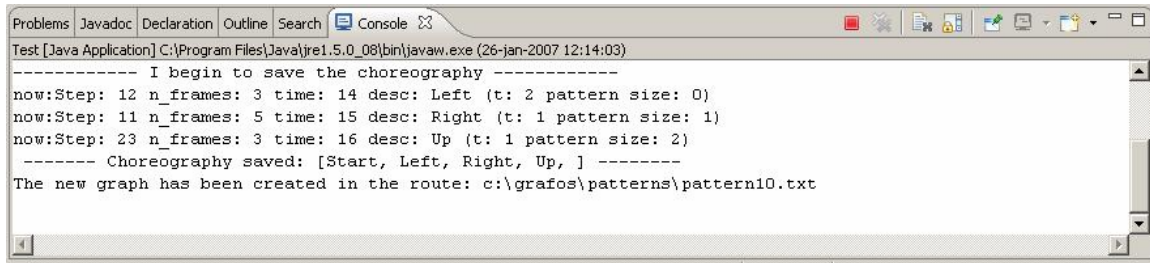
Due to a time problems, this function has been integrated in the step recognition function, thus basically when we start the application, the firsts movements are saved and a HMM graph is automatically generated (and saved).

#### **3.4.1. General explanation**

This function has been done in the ReciveStep class (the class created to recognize complex patterns) and it is running during some seconds. This function does not work exactly at the beginning of the application, because, as we have seen before, some simple steps needs some seconds after the user is in front of the camera to do calculations and until these calculations has not been done the recognition does not work well. For this reason, this function begins to work 40 seconds after the application, we have thought that it is enough for the user is able to go to the dance floor and to do the calculations.

This function is running until a pause of 4 seconds is produced, thus, to save a complex pattern, the user has to be in front of the webcam at least 34 seconds after the application is running (the duration of the calculations is more or less 6 seconds), in the second 40 the sentence "----- I begin to save the choreography -----  
---" will appear in the console window, then the user have to begin to do the pattern (as long as the user wants), and when the user does a pause of 4 seconds (without any simple movement) this pattern is saved in a file and the next sentence appears in the console window: " ----- Choreography saved: [X] -----", where **X** is the sequence of steps that will be recognized by this graph. After this, appears a line telling where has been saved this graph. We can see an example of this process in the figure 3.9.





```

Problems Javadoc Declaration Outline Search Console
Test [Java Application] C:\Program Files\Java\jre1.5.0_08\bin\javaw.exe (26-jan-2007 12:14:03)
----- I begin to save the choreography -----
now:Step: 12 n_frames: 3 time: 14 desc: Left (t: 2 pattern size: 0)
now:Step: 11 n_frames: 5 time: 15 desc: Right (t: 1 pattern size: 1)
now:Step: 23 n_frames: 3 time: 16 desc: Up (t: 1 pattern size: 2)
----- Choreography saved: [Start, Left, Right, Up, ] -----
The new graph has been created in the route: c:\grafos\patterns\pattern10.txt

```

**Figure 3.9:** Console window while we are saving a complex pattern

### 3.4.2. Programming explanation

When this function begins to run (40 seconds before the receivedStep thread has been launched), it begins to save the received simple movements in a list of integers called oTest2 (obviously we have used the done code in the recognition part).

For each frame we look if has been a default generated frame (when we have been 2 seconds without any received step) or if has been a real step done by the user; in this case, we add this step to the list where we are saving the complex pattern.

This function continues running until we receive two following default steps. As these default steps are generated each 2 seconds, we have been waiting 4 seconds. In this case, we stop to save simple steps and we copy these steps in an Array of integers which will be sent to a new thread that will built the HMM graph. The new array of integers must not have a quiet movement in the first position because this is automatically done by the graph generator.

When the array of integers (called oTest) has been filled we launch the PatternGraphGenerator thread with this array as a parameter.

The PatternGraphGenerator builds a generic pattern graph, with its probabilities and its transitions.

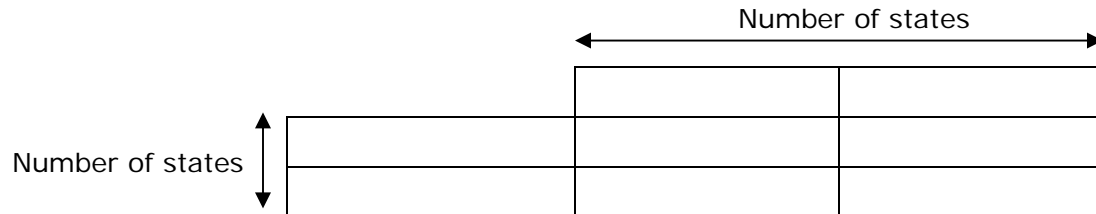
These graphs have the structure of a complex pattern HMM graph (explained in the HMM part).

The number of states of this graph is the number of simple steps done plus one, because we have to create an initial state, and we also have to create the Alpha a

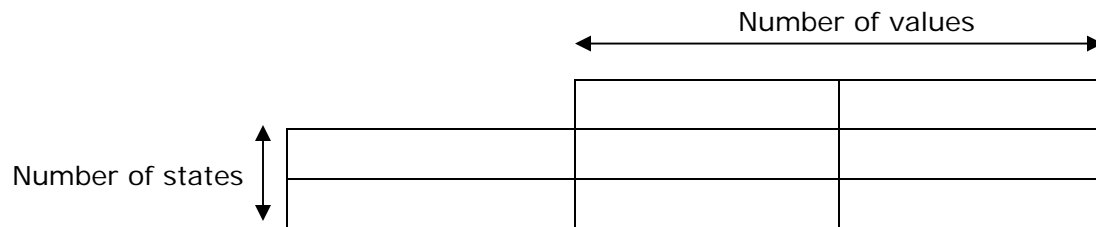
Beta tables with the probabilities of jump and the values. These percentages are fixed, but we have to create these arrays of the correct size.

The size of these arrays is:

Alpha probabilities (probabilities to jump to other state):



Beta probabilities (probabilities to have a value in a state):



In our case the number of values is always 7, because we have a fixed number of recognized steps. If this some new step is recognized, it will have to be changed.

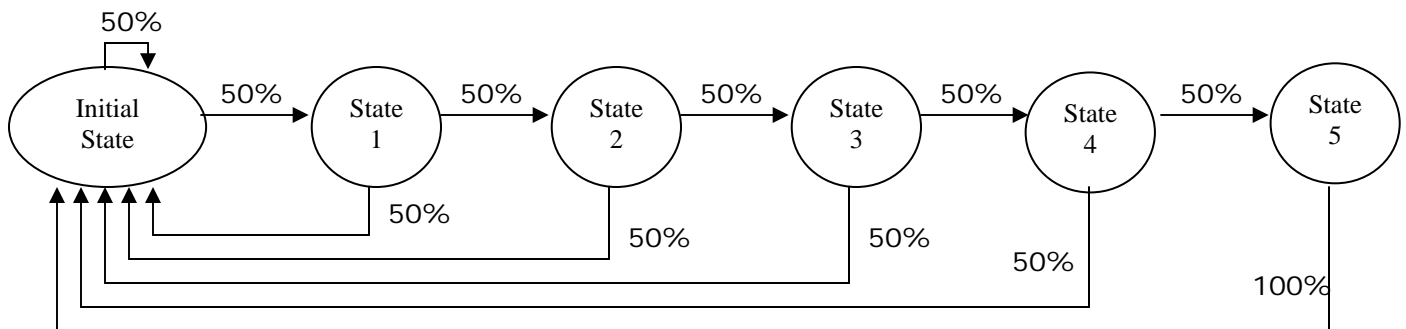
This graph will be saved in a txt file (that we will be able to load later) in the route "c:\grafos\patterns" (this route can be easily modified) with the name patternXX.txt, where XX is the next free number in a sequential order.

### 3.4.3. Example

For example, if the user has done, during the time that this function is saving, the next pattern (also called choreography):

Left Step + Right Step + Left Step + jump + twister

We have 5 steps to save, and then we have to create 6 states (initial state + 5 simple steps).



The transitions probabilities (Alpha probabilities) are always the same, which is 50% to go to the next step and 50% to go to the beginning, except the final step that only can back to the initial step.

**Alpha table:**

	Initial State	State 1	State 2	State 3	State 4	State 5
Initial State	50 %	50 %	0 %	0 %	0 %	0 %
State 1	50 %	0 %	50 %	0 %	0 %	0 %
State 2	50 %	0 %	0 %	50 %	0 %	0 %
State 3	50 %	0 %	0 %	0 %	50 %	0 %
State 4	50 %	0 %	0 %	0 %	0 %	50 %
State 5	100 %	0 %	0 %	0 %	0 %	0 %

**Beta table (output probabilities):**

Here we have to know that each state represents a different step of the pattern:

State1 = Left Step

State 2 = Right Step

State 3 = Left Step

State 4 = jump

State 5 = twister

And then in this state, we only can have this value.

	Quiet	Left	Right	Up	Down	Jump	Twister
Initial State	100%	50%	50%	50%	50%	50%	50%
State 1	0%	100%	0%	0%	0%	0%	0%
State 2	0%	0%	100%	0%	0%	0%	0%
State 3	0%	100%	0%	0%	0%	0%	0%
State 4	0%	0%	0%	0%	0%	100%	0%
State 5	0%	0%	0%	0%	0%	0%	100%

This graph will be saved in a text file in the next format:

- states, variables, kind of graph:

6 7 -1

- Initial states:

1.0 0.0 0.0 0.0 0.0 0.0 0.0

- Alpha table:

0.5 0.5 0.0 0.0 0.0 0.0 0.5 0.0 0.5 0.0 0.0 0.0 0.5 0.0 0.0 0.5 0.0 0.0 0.5 0.0  
0.0 0.0 0.5 0.0 0.5 0.0 0.0 0.0 0.0 0.5 1.0 0.0 0.0 0.0 0.0 0.0 0.0

- Beta table:

1.0 0.5 0.5 0.5 0.5 0.5 0.5 0.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0  
0.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0  
0.0 1.0

## 4. The failed recognition attempt

In the chapters before, we have explained the simple movements that have been well recognized, but during this time, we have tried to develop some others functions to recognize others movements, these functions have not obtain satisfactory results, and for this reason, we have not included these recognized movements in the previous chapters.

Although these recognitions have not been finally included we have done a previous study and we have done some test that will be explained following that can be very useful to follow extensions of the project.

### ***4.1. Hands movement recognition***

One interesting function that we have tried to do is to recognize what is doing the user with the hands; this function includes since recognizing a hand up until a clap.

With this function working we also could find some others useful application to improve the others recognitions, because a hands movement can give us a very important information about the movement done by the user, and also can avoid bad recognitions produced by hands movements.

To do this function we have had to use another input data not used until now, this new data is the hands position. Our first goal was to know where the hands were, for this, we created two detectors (one for each hand).

The received data is explained in the Vision part, but basically, we receive the quadrant number where the vision part thinks that the hand is. With this information we want to inform about when the user changes the position of his hands and we want to say what the new position is.

When the user has his arms stiffed this software runs quite well and we know where the hands are.

In the rest of the cases, we have had a lot of problems like we tell following:

- If the arm is over the body, the software does not find the arms, and can say that the arms are in the raised position or in the lower position, it depends of the user's clothe, of the background, and so on.
- If the arm is not completely over de body, this software detects few part of the arm and tells that the entire arm is in this position.
- There are some positions where the application does not know in what quadrant the arm is exactly, then in each frame says one or the other while the real arm can be at the same position.
- If the user has a T-shirt, the application could not recognize the arm.

In the MoveDetector class we can find the done code to recongnize the hands movements. Although this code works (it does not have programming problems), it has been disabled to do not disturb the rest of recognitions.

## ***4.2. Forward and backward step recognition***

Another important simple step that we would like to have recognized is the forward and backward step.

Our actual application, only recognize 2D movements (horizontal and vertical movements) but in a dance the user also do a backward and forward movements that now are not detected.

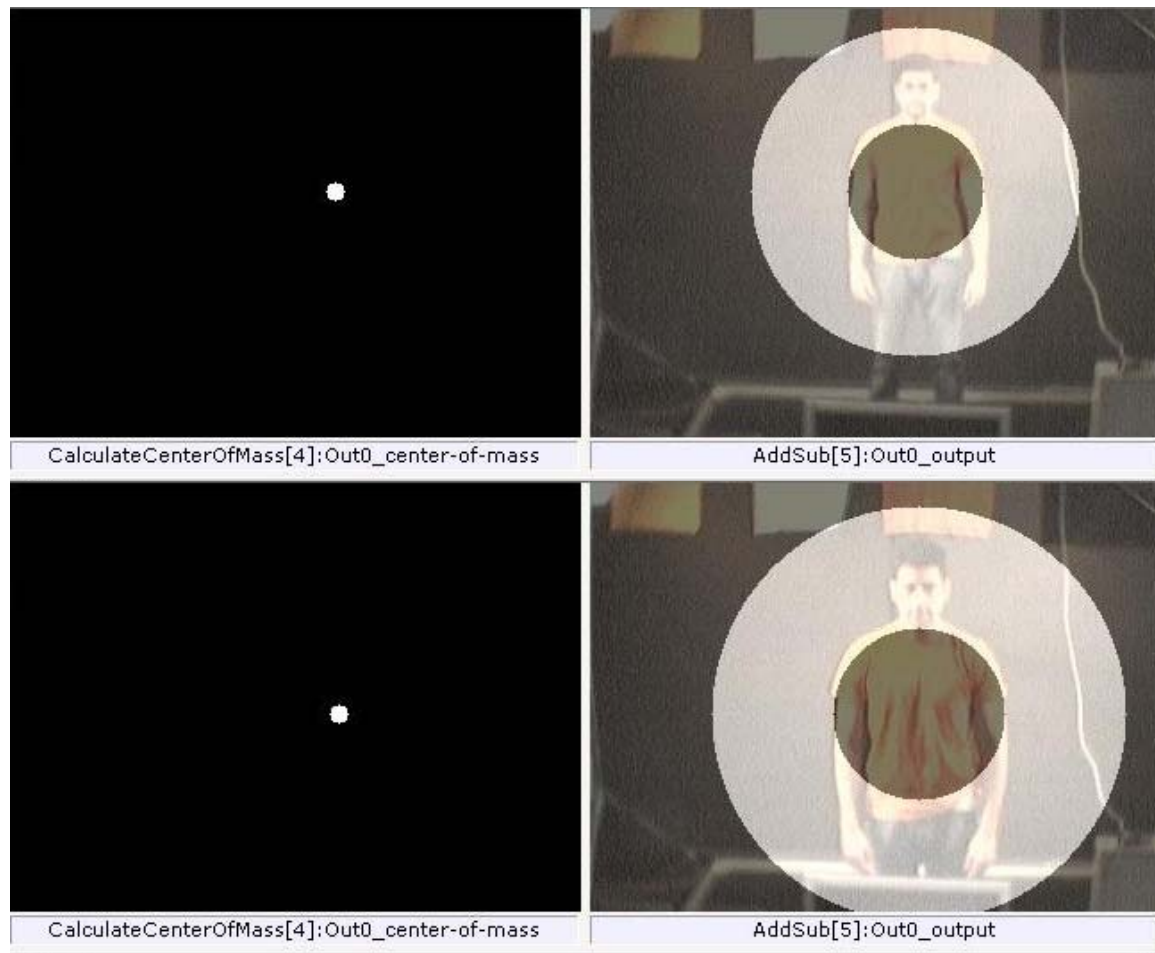
This option has been studied, and we began to do some test, but we have not had enough time to develop this recognition and the done test had too many errors.

We have tried to recognize the forward and backward movement we using 3 variables, the horizontal position of the centre of mass, the vertical position of the centre of mass and the radius of the user.

We have detected some patterns in these variables when the user does this kind of steps:

- Forward step:
  - o X position: There is not any change.
  - o Y position: This value is increased because the user seems taller.
  - o Radius: This value is decreased because the user is closer to the camera
- Backward step:
  - o X position: There is not any change.
  - o Y position: This value is decreased because the user seems shorter.
  - o Radius: This value is increased because the user is farer to the camera

In the next figure we can see these values in a real image, and we can admire that when the user is in the normal position, the radius is smaller and the centre of mass is in a higher position, while when the user does a forward step, the radius is larger and the centre of mass is in a lower position.



**Figure 4.1:** forward and backward step features

Theoretically, with these values, we can know if the user is doing a forward or backward step, but our test have not been successful and we have not had time to continue developing this function.



## 5. Extending this application

Nowadays, this application can recognize seven simple steps and one complex pattern, but, is quite easy to recognize new simple steps or complex patterns without learn very deeply how is running this software.

In this chapter we are going to explain how to extend the simple and complex recognitions easily and quickly without a hard study of the done code. This is a good option if we only want to create some new movement, but if we really want to know how this application works and why this application recognizes the movements we hardly recommend to read and understand the chapter 3.

### *5.1. Developing a new simple step recognition*

To recognize a complete new movement is very difficult, for this propose the most recommended option is to read and understand the code because you should deeply modify some classes and probably, to create some new classes. This is a not recommended option if you do not want to dedicate a lot of time to the application.

In other case, if you want to take some of the existing simple recognitions (horizontal and vertical, twister runs different) and to recognize some new movement, there are an easy process to follow that only implies to understand the HMM graph.

To recognize a new step, first we have to study the data received by the HMM graph, for example, in the horizontal recognition case, the received values are left, very left, right and very right. Combining these values we have to recognize the new step.

The next step is to modify the HMM graph to detect the new step without to change the existing paths (it can entail many problems), thus the best option is to create a new path in the graph.

When you have created a new path (and you have filled the probabilities tables), you have to number each state, and it is important to remember that the last state always has to have the last state number.

This graph must be saved with the same name and in the same place than the original file.

The second step is to label this new path. For this we have some arrays to update, the first is in the thread which will analyze this graph (horizontalMoveDetector or verticalMoveDetector), here we have an array called stateNames which size is the number of the states of the graph, we have to modify this size with the new size, and put the name of the new states (the last states always have as a name: "final state"). If the path has more than one state, we only have to label the first state of the path (the rest does not matter).

The sent code of this new step is the number of its first step plus 10 if it is a horizontal movement or plus 20 if it is a vertical movement.

The third step is to recognize this simple movement in the complex recognition part. For this, we have to go to the receiveStep thread. Here we have 2 structures, a hashMap and an Array. In the HashMap we have to translate this received code to a sequential code (here, each simple step have a sequential code), now the application recognizes 7 different simple steps (codes from 0 to 6), therefore, the new step would have the code 7.

Finally we have to label this step in the array of the strings also called stateNames, which contains the name of all the recognized simple steps.

In conclusion, only changing a HMM graph and updating two arrays and one hashMap we can recognize new Horizontal or Vertical movements.

The twister recognition does not let us to create other step recognition in the same graph due to its way to work.

## ***5.2. Developing a new complex pattern recognition***

To develop a new complex pattern is the easiest part thanks to the graph generator function and the done code.

Obviously is also possible to make an own graph studying the features of this kind of graph, but we should take the advantage of use the developed function to create these graphs automatically.

When we already have created the new graph, we have to tell to the application that the pattern that we want to recognize is the just saved created graph, for this we have to go to the reciveStep class, there we will find the next line:

```
"patternMoveDetector = new PatternMoveDetector(o, 1, time, "XXX")"
```

where XXX is the name of the file that contains the pattern to recognize. Thus we only would have to change this name for the name of the file that contains our graph info (without the file extension).

Now, the recognition pattern will be our new graph.

With this option, we only recognize one kind of pattern. Maybe we could be interested in search the new pattern besides the other pattern.

To do this, we must do a little change in our code. We have to take the next code:

```
patternMoveDetector = new PatternMoveDetector(o, 1, time, "XXX");  
patternMoveDetector.start();
```

and repeat it just under the existing code, but changing the file name to load (XXX).

Now our application will recognize both complex patterns.

## 6. Test bench

Now, that we consider that our code is finished, we have to do some tests to know the success rate of our application. These tests are necessary to decide if this technique can be used in the future.

Moreover, all these kind of applications need to demonstrate that they are valid to do this job, and the only way to demonstrate it is to do some tests and to analyze its results.

In this chapter we are going to show the done tests and we are going to explain the reason of these results

### ***6.1. Test setup***

These tests have been done in our laboratory with a good light and a dark background. The user wore a clear T-shirt to have a good contrast with the background, thus we can say that the test has been done with good external conditions.

Due to problems with the recorded video processing we have had to do these tests in real time (with a real user dancing while we are studying the results), this difficult us the test because we cannot repeat the movements to verify the results, but we think that it is a small problem because this application is going to run in real time and not with a recorded video.

We have divided these tests according to the thread which has recognized the step, this is because the detection done by each one of these threads is completely independent and then, the result of one thread cannot modify the result of the other.

Moreover, we have done each kind of movements in 3 different ways to know our rate success depending of the way to move of the user. These ways to test the thread are explained following:

- Individual step: The user is stopped and does one unique movement.
- Various steps (slow): The user does a sequence of movements, but doing a little pause between these movements.
- Various steps (fast): The user does a sequence of movements without any pause between these movements.

This results has been obtained after doing (and analyze) a lot of movements, the optimum test would be doing the same steps for each thread, but as each thread runs with a different data and recognize a different number of steps, it has not been possible. For this reason the number of movements done in each test is different.

Number of movements analyzed in each thread:

Horizontal test: 92 movements.

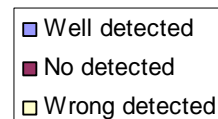
Vertical test: 80 movements.

Twister test: 38 movements.

We can see that the twister test have less movements analyzed, but it only recognize one movement. For this reason the number of combinations of the rest of threads is larger.

Finally in our graphs we have 3 kinds of results:

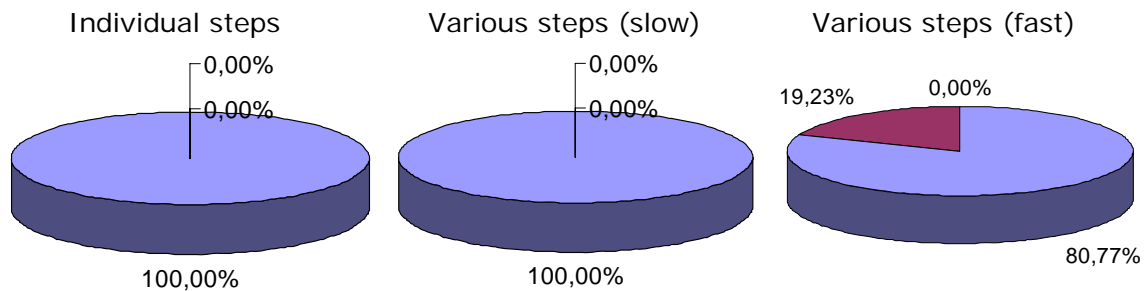
- The step has been well recognized.
- The step has been wrong recognized.
- The step has not been recognized.



As we can see, we have 2 kinds of wrong results, we consider that a movement is not detected when we do a movement and our application does not detect any movement, and we consider a wrong detection when we do a movement and our application says that we have done another different movement.

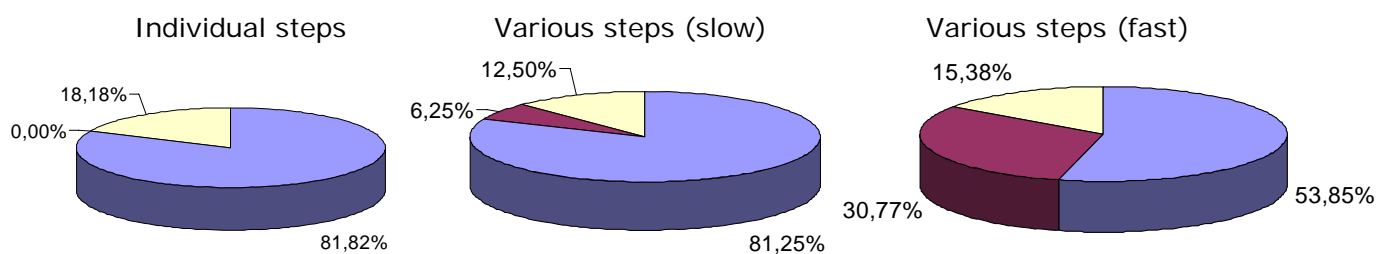
## 6.2. Simple movement test

### 6.2.1. Horizontal test (left and right step)



As we can see, the horizontal recognition is the best successful recognition done. With individual steps and slow steps we have not find any problems during our tests. We only begin to have some problem when we begin to dance very fast, in this case we can have some problems, but these problems are because our application only detect one step when de user has done the same step some following times (the application cannot detect when finishes one and when begins the other).

### 6.2.2. Vertical test (up, down and jump)



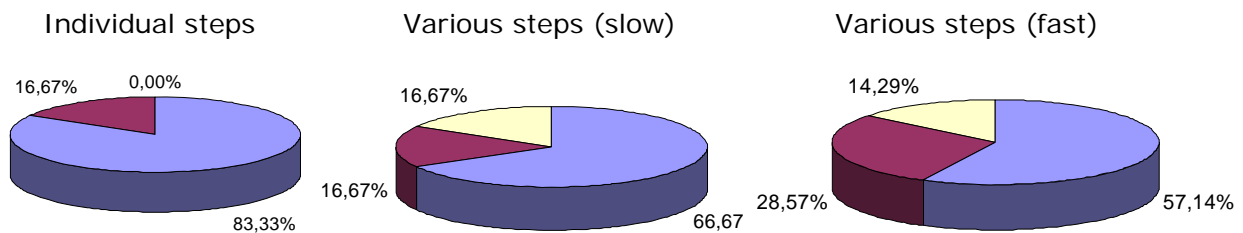
The vertical recognition also works quite well when we do individual steps or we do various steps not very fast.

When we do individual steps the recognition is quite good, and the only error detected is that some jumps are recognized like ups. The rest of test has been successful.

With various steps we continue having the same problem with the jump and up, but now appear a little problem when the user do the steps very slow, the problem is that we are calculating the average centre of mass every time, but if we stay a lot of time in a lower position, this average centre of mass is completely recalculated and some times can fail the next step.

Finally when we try to do vertical steps very fast, the recognition gets worse quickly. Here the most important problem is the not detected steps. This is because the frame rate does not let us to receive all the frames, thus we lose some frames and we cannot completely recognize some steps.

### 6.2.3. Twister test



The twister recognition is the most easily disturbed by a bad vision recognition. If the user radius is not well recognized, it begins to increase and decrease constantly and it produces a lot of errors.

In this case, we have a quite good recognition, and when we do individual twisters they are also recognized quite well. The most of the times that are not detected is because the user has done some other step that has invalidated the twister recognition and some other because in some moment the radius has not been well detected

When we begin to do more than one twister consecutively, we find that we continue without detect the twister in the same times than in the individual tests, but, in this case we also have some error detecting some twister when it has not been produced. These cases are when we have begun some twister and we have not finalized it, then

we can do some movement that the application things that is the final part of the twister.

Finally, testing with fast twisters we have seen that we have more not detected twister. This is because in fast twister we can loose some frame and then the application join some consecutive twisters.

### ***6.3. Complex pattern test***

The other part of our work has been to recognize complex patterns. In this case we have to receive the simple steps recognized by the threads tested before and detect if its follow a determined pattern.

To do this test we have a very important problem, this problem is that we do not have any fixed complex pattern to be tested. We can try to recognize any complex pattern, and the user can want to recognize some pattern that he has done.

Obviously the success rate will change depending of the pattern's size and the kind of the steps inside the pattern.

Moreover, we have tried to send some complex pattern many times, and it function always recognize this complex pattern. Then, can we say that this thread have a 100% of success? The answer is not.

This kind of recognition has a 100% of success only if the received data is correct, but it is not true because here we depend of the simple recognitions.

For example if the user wants to recognize the complex pattern "left step + jump + right step" and the simple recognition threads recognize "left step + up + right step", then this complex pattern will not be recognized, but this is not a fault of the complex pattern thread.

We could try to calculate this success rate using the percentages of the simple movement test, but it always depends of the length of the sequence and the movements included in this sequence.



## ***6.4. Test Conclusions***

After seeing these results we can say that our applications recognize very well the horizontal movements and we only have problems with some forgotten step.

The vertical and twister recognition is quite worse because we have as not recognized movements as wrong recognized movements. In some cases like the fast vertical movements our success rate is quite bad and we only recognize few more than the half of the done steps.

In conclusion we can say that our application runs quite well with horizontal movements and slow movements, but when we begin to do a lot of movements and without pauses between these movements, our success rate decrease.

## 7. Conclusions

### ***7.1. Reached and failed goals***

After this report we can say what goals have been reached and what goals will continue been available for a future studies.

#### 1. To detect simple movements in the user dance

We consider that this goal has been successfully reached because we have gotten to recognize the most important simple movements (left step, right step, down, up and jump). Moreover, we have gotten to recognize the twister movement which enrich our final work.

Due to time problems, other simple movements like forward/backward steps or hand movements have not been able to be reached, but we have done a previous study and we have given some ideas of how could be done.

#### 2. To detect complex patterns in the user dance.

We also consider that this goal has been reached. Now we can recognize a pattern (previously saved) in a dance if it has been done. Moreover, the pre-saved database is easily updatable, and we hope that in the future, a lot of new pattern will be saved.

#### 3. To Generate automatically pattern references.

We consider this goal almost reached, because we can save automatically a user dance pattern, and this saved pattern can be easily used in future dance recognitions. Due to time problems we have available this function only one time during the song, but in a future it can be easily improved to let to the user save many patterns in a same song.

#### 4. To search patterns in a dance without any reference.

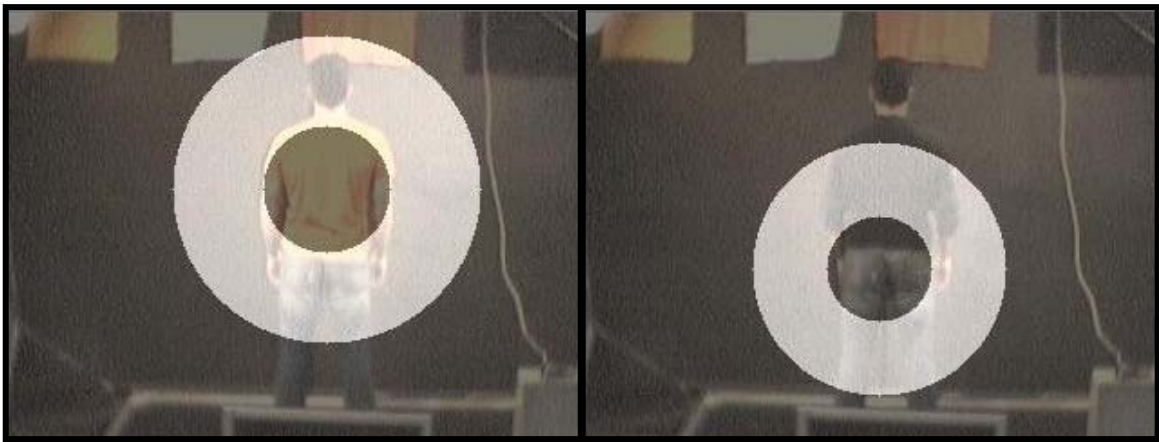
This goal has not been reached, we have looked for some ways to find out this kind of patterns, but it is a very difficult job that probably could be another entire project.

## 7.2. Found problems

Although we have got a quite good success rate, nowadays, our applications have some problems that we have to know.

- The visual part is easily influenced by the light, user clothe, webcam resolution, and so on, thus, the received information is not very well, and we have to do some software corrections to improve this information.

In the next figure we can see the difference that we can have wearing light or dark clothe.



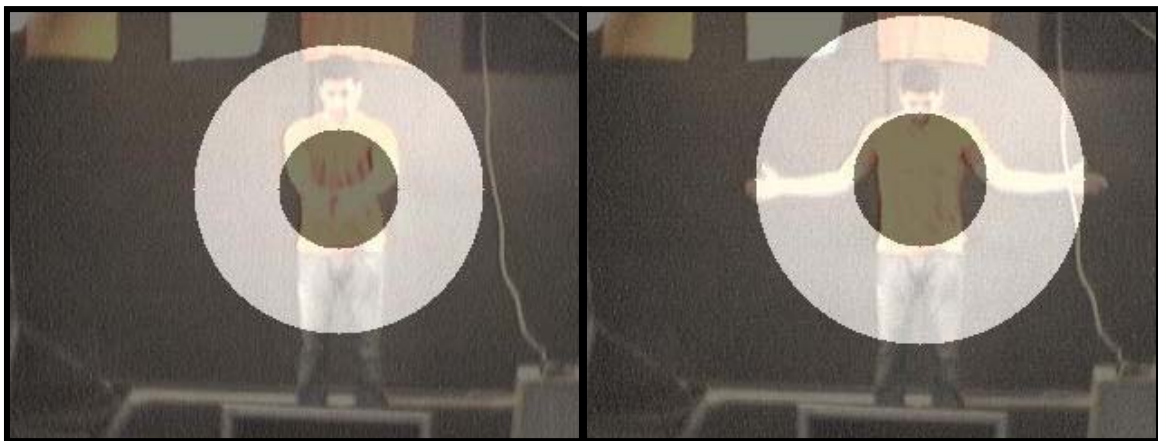
**Figure 7.1:** Differences in the visual part according to the user clothe.

- The visual part send the information with a certain frame rate, our detection would be very much precise if we would know exactly how many frames we receive in each moment, because the frame rate changes dynamically. We have not been able to receive dynamically this frame rate, thus we have fixed this value to 15 frames / s (but it should be changed if the real frame rate is very different). In our test computers this value is very similar to the real frame rate, but sometimes we can forget some frame or take a frame two times.

- The horizontal recognition is very accurate, but it is not perfect, the main errors are detected when the user does many steps to the same direction, then the number of steps done by the user and the number of steps recognized can not be the same.

- The body swing can produce the detection of some step, when actually, the user is stopped. It has been almost solved using a percentage of his radius as a threshold.
- The movement of the user's hands can cause errors because it can change the centre of mass of the user and his radius. It has been tried to solve with the hand movement recognition, but finally has not been applied.

In the next image we can see an example of the changes in the user radius depending to the position of his hands.



**Figure 7.2:** User radius according to the hands position

- The vertical recognition is very difficult to do, because is difficult to differentiate the some movements, like jump or stand up since a lower position. To solve this problem we have calculate the normal Y position of the centre of mass, but in some cases it can be bad calculated (for example if the user is a lot of time in a lower position).
- The twister recognition, as we have explained before, is done following the patterns of size of the user's radius. Here we have basically 2 problems.  
The first is that to avoid bad recognitions we only detect the twisters when the user is stopped, then if while the user is doing the twister, he does some step, this twister is automatically discarded.  
The second problem is that a user doing random movements (for example, hands movements) could generate the changing radius pattern, and then, we could have false twister recognition.

- To do a step recognition, we must send a sequence to be analyzed by the HMM graph. The main problem is, when we stop to fill this sentence to send it, because if we wait too much time, the step will be recognized too late (and we want to recognize steps in a real time), while if we stop to built the sequence too soon, we can cut a steep, and then it will not be detected (or will be detected twice).

To solve this problem we have decided to not send a step until we detect a short pause. The problem is reduced, but appears the problem that if the user does a lot of steps without any pause, the sequence is not send (until the next pause), and when we recognize the steps is too late.

This problem looks like very important, but at last, is difficult to find this situation because the user always do stops.

- The steps are recognized by threads (different threads depending of the kind of recognition), as the threads runs at the same time, we can do two different steps, each one be sent to a different thread, and be recognized in an incorrect order.

- The thread, which receive the simple recognized steps and build the sequence to be send to analyze, inherit the simple step recognition problems, therefore, it can receive the steps in an incorrect order (then the complex pattern will not be recognized) or receive the steps too late and to not recognize this complex pattern (because the sequence can be already sent).

### ***7.3. Possible improvements***

This project has a lot of possible improvements, now we are going to explain the 11 most interesting improvements (under our point of view).

1. To improve the visual detection, because the results have an important variation depending the light and the user's clothe.
2. To include another camera in the vision system, because now with only one camera we only have a 2D perspective (especially to improve the twister recognition).
3. To use the data detected by the dance dance revolution pad to improve the actual recognitions.
4. To do more robust the movement recognition (specially the vertical recognition thread)
5. To detect twister movement at the same time than other movement.
6. To detect forward and backward steps (this function has not been finished due to time problems).
7. To detect the hands movements (this function have had the same problem than the prior possible improvement).
8. To find a better way to decide when we send the sequence to the graph (to wait to a pause of the user probably is not the best way to decide it).
9. To develop a learning algorithm (like k-means or something similar) to improve the graph probabilities.
10. To modify the receiveSteps thread to correct a possible incorrect order reception of the simple steps.

11. To develop a new recognition thread which can detect pattern in a user's dance and create automatically de HMM graph for this patterns.
12. To try to relate the beat detector data with the complex pattern recognition, in order to know in what moment of the song this choreography has been done.
13. To do a new system to make easier to use the recording pattern function (now is implemented at the beginning of the application, but it should be possible during all the song).
14. To let to the user to do pauses in a complex pattern.

## ***7.4. Personal opinion***

After working during five months in this project we have a global idea of how to recognize some user movements in real time using only a webcam.

Nowadays, this area of study is still growing, that means that we do not have a determined path to follow, there are many techniques trying to get the same goal. We have chosen one and we have realized of the difficulty of these projects that, we think, will be very important in the future.

Our first conclusion is that the movement recognition is really difficult because there are many possible variables involved in this recognition (light, user, computer, webcam...) and all these variables can modify our results easily. In this technique we have tried to do an application that works with all the users, but probably to improve the global performance we should train the machine for each user (similar to the speech recognition) and limit some of these variables.

Using Hidden Markov Models graphs we have been able to recognize simple different movements and complex patterns with very few information, and we think that this technique can be improved until get excellent results, but now, have some important limitations and for these reasons the results are not as good as we would like.

In our opinion this part of the virtual dancer project is only the beginning of new and amazing functionalities in this kind of applications, like recognize very complex choreographies and dance some of these choreographies with the user in a more believable way or learn movements to apply later, that will be present in the society very soon.



## 8. References

### Books

1. Radford M. Neal: Probabilistic Inference Using Markov Chain Monte Carlo Methods (1993).
2. Saeed V. Vaseghi: Advanced Digital Signal Processing and Noise Reduction (2000).
3. A. S. Poznyak & k. Najim & E. Gómez-Ramírez: Self-Learning control of Finite Markov Chains (2000).
4. Wilhelm Huisinga & Eike Meerbach: Markov Chains for Everybody (2005).
5. Troy L. McDaniel: Java HMMPak v1.2 User Manual (1996).
6. Stuart J. Russell & Peter Norvig: Artificial Intelligence A Modern Approach (1995)

### Articles

1. Sean R. Eddy: Hidden Markov Models (1996).
2. Lawrence R. Rabiner: A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition (1989).
3. L. R. Rabiner & B. H. Juang: An Introduction to Hidden Markov Models.
4. Terran Lane: Hidden Markov Models for Human/Computer Interface Modeling.
5. Edgar Noda & Alex A. Freitas & Heitor S. Lopes: Discovering Interesting Prediction Rules with a Genetic Algorithm
6. Asuncion Gomez Perez: Introducción a la representación de conocimientos
7. David Diaz: Inteligencia Artificial, Robótica, Neurocomputación, Programación Neuronal Y Otras Hierbas (2001)
8. Michaelmas Term: Markov chains (2003)
9. Geoff K. Nicholls: Bayesian Inference and Markov Chain Monte Carlo by Example (2004)
10. N. Davey & S. P. Hunt & R. J. Frank: Time Series Prediction and Neural Networks.
11. Jonathon Shlens: Time Series Prediction with Artificial Neural Networks (1999)
12. Jafar Adibi & Wei-Min Shen & Eaman Noorbakhsh: Self-Similarity for Data Mining and Predictive Modeling A Case Study for Network Data.
13. Jose Oncina: Aprendizaje Computacional y Extracción de Información (2006).
14. Alfons Juan & Enrique Vidal & Roberto Paredes: Aprendizaje y Percepción (2005).
15. Takaaki Shiratori & Atsushi Nakazawa & Katsushi Ikeuchi: Dancing-to-Music Character Animation (2006).

16. D. Milone: Modelos ocultos de Markov para el reconocimiento automático del habla (2004).

**URLs**

1. <http://www.cs.brown.edu/research/ai/dynamics/tutorial/Documents/HiddenMarkovModels.html>
2. [http://en.wikipedia.org/wiki/Hidden\\_Markov\\_model](http://en.wikipedia.org/wiki/Hidden_Markov_model)
3. <http://www.informatik.uni-bremen.de/agki/www/ik98/prog/kursunterlagen/t2/node4.html>
4. [http://www.ici.ro/ici/revista/sic2006\\_1/art02.html](http://www.ici.ro/ici/revista/sic2006_1/art02.html)
5. <http://java.sun.com/developer/technicalArticles/ALT/sockets/>



Project done by Jesús Sánchez Morales

Student of Escola Tècnica Superior d'Enginyeria at the Universitat Autònoma de Barcelona.

Signed: .....

Enschede, 09 of February of 2007

## Resum

Durant aquest projecte s'han intentat detectar moviments de l'usuari utilitzant la informació rebuda d'una càmera web. Aquesta funció ha estat implementada dins de l'aplicació de la ballarina virtual.

En aquesta part del projecte rebem informació ja tractada per la part visual de l'aplicació, la qual només ens dona la informació més important de l'usuari, com podria ser la posició del seu centre de massa o el radi al voltant d'ell.

Amb aquesta informació s'ha buscat una manera de traduir-la a un moviment reconegut (pas a la dreta, pas a la esquerra, salt,...).

El mètode utilitzat ha estat el Model Ocult de Markov. Amb aquest mètode hem creat alguns grafs amb les característiques de cadascun dels moviments a reconèixer, i analitzant una seqüència d'entrada sobre aquest graf, utilitzant l'algorisme de Viterbi, podem reconèixer els moviments realitzats per l'usuari amb uns resultats força bons.

Després d'això, hem intentat millorar el mètode per també detectar patrons de moviment complexes (pas dret + twister + salt + ...). Per fer aquesta nova funcionalitat hem adaptat la funció de reconeixement de moviments simples per rebre els passos simples ja reconeguts i detectar algun tipus de patró en aquests moviments.

Finalment hem realitzat una funció per generar el grafs HMM utilitzats per reconèixer els patrons complexes de moviments.

## Resumen

Durante este proyecto, se han intentado detectar movimientos del usuario utilizando la información recibida de una cámara Web. Esta información ha sido implementada dentro de la aplicación de la bailarina virtual.

En esta parte del proyecto recibimos información ya tratada por la parte visual de la aplicación, la cual solo nos da la información más importante del usuario como podría ser la posición de su centro de masa o el radio a su alrededor.

Con esta información se ha buscado la manera de traducirla a un movimiento reconocido (paso derecho, paso izquierdo, salto,...).

El método utilizado ha sido el Modelo Oculto de Harkov. Con este método hemos creado algunos grafos con las características de cada uno de los movimientos a reconocer, y analizando una secuencia de entrada sobre este grafo, utilizando el algoritmo de Viterbi, podemos reconocer los movimientos realizados por el usuario con unos resultados bastante buenos.

Tras esto, hemos intentado mejorar este método para detectar también patrones de movimiento complejos (paso derecho + twister + salto +...). Para realizar esta nueva funcionalidad hemos adaptado la función de reconocimiento de movimientos simples para recibir los pasos simples ya reconocidos y detectar algún tipo de patrón en estos movimientos.

Finalmente hemos realizado una función para generar los grafos HMM usados para reconocer los patrones complejos de movimientos.

Corresponder

## Summary

During this project has been tried to detect user movements using the received data of a webcam. This function has been implemented inside the virtual dancer application.

In this part of the project we receive treated data by the visual part of this application which gives us only some important data about the user like could be his centre of mass position or the radius round him.

With this data has been searched a way to translate this data to a recognized movement (left step, right step, jump ...).

The method used has been the Markov Hidden Model, with this method we have created some graphs with the features of each movement to recognize, and analyzing an input data sequence over this graph, using the Viterbi's Algorithm, we can recognize movements done by the user with quite good results.

After this, we have tried to improve the method to also detect complex patterns (left step+twister+jump...). To do this new function, we have adapted the simple movement recognition function, to receive the already recognized steps and detect some kind of patterns in these recognized movements.

Finally we have done a function to generate the HMM graphs used to recognize complex patterns.