Classification of Joints and Joint Structure Using Stereovision

Andrew Spielberg
Department of Computer Science
Cornell University
Ithaca, NY 11554
aes89@cornell.edu

Shahrukh Mallick
Department of Computer Science
Cornell University
Ithaca, NY 11554
skm47@cornell.edu

Abstract

Autonomous understanding of the kinematic structure of objects in an agent's environment has important applications in the fields of robotics and computational physics. In this paper, we present a novel means for joint detection and classification using stereoscopic data of the agent's world. Our algorithm consists of three phases. The first is an unsupervised link detection phase, wherein we determine the set of links that compose the kinematic body. Here, we employ 2-D tracking algorithms coupled with 3-D spectral clustering methods. The second is a joint location detection step, which uses assumptions about the locality of jointed links along with a k-nearest neighbor algorithm for correcting noise in the link segmentation. The third step is a supervised classification step using a multiclass support vector machine. The joint detection was 55% accurate, able to find all the joints on ten out of the eighteen images. The joint classification had a LOOCV accuracy of 48.57% and a test accuracy of 40% on a small test size of 5 samples. These results are promising, and show that this algorithm provides a novel way to detect the kinematic structure of objects.

1 Introduction

The task of identifying objects and understanding how to utilize these objects is a growing area of interest in the fields of machine learning and robotics. Specifically understanding the kinematic model of an object will illuminate the functionality and uses of the object (e.g. a robot can learn how scissors move without any prior knowledge about the structure of the scissors, and can use this knowledge to cut paper). This extra layer of info will provide more valuable information to a computer. This is a very simple idea, and this type of learning can be applied to a wide scope of tasks and objectives.

In this paper, we will specifically focus on identifying the location of joints and classifying the type of joint (revolute, prismatic, etc.) on an object. Previous work by Brock and Katz [1] suggests a method for tracking motion of jointed objects over time and determining the joint structure. While it is generalizable to n-jointed objects, it has two main shortcomings. One, it requires that the object's joint motion be projectable onto a 2D plane (and manipulated on the plane), and two, it only can identify revolute joints. We seek to extend this work by using 3D stereographic data to develop methods for determining the structure of kinematic chains and trees using a large library of joints. To make things simpler, we consider the case where an outside agent can intelligently and helpfully manipulate the studied objects to easily generate training sets. Our goal is to develop an algorithm that is reasonably fast and highly accurate.

2 Dataset and Methodology

In developing our dataset, we sought to provide data points for four types of joints – prismatic joints, screw joints, revolute joints (the most common), and ball and socket joints. "Fixed joints," which really resemble the absence of joints between two structures, were not explicitly considered in the design of our dataset; nevertheless, they were considered in the classification stage which we will discuss later. While "sliding" joints were considered, anchor points were considered too difficult to determine, and thus these were left out of the dataset since their motion bore too much resemblance to prismatic joints. Most items we considered contained one true joint, although some items contained multiple true joints. For multi-jointed objects, the types of joints did not necessarily have to be of the same classification; for example, objects could contain two joints, one prismatic and one revolute. We provide in Table 1 below a list of the recorded items, a brief description of them, and their actual joints.

Recording of data was done with the Bumblebee2 stereocamera, which is able to take still PGM images of its left and right cameras, along with 3-D pointcloud data of the scene. The pointcloud data involved the x, y, and z coordinates of points on non-background surfaces that the camera was able to see, and its corresponding location in the left camera. Since the desired dataset was video of jointed items' movement, and the Bumblebee2 can only take still photos and point clouds for our dataset, we had to simulate movement through stop motion. Our stop motion "videos" ranged from 10-30 frames, with the supermajority being in the 15-25 frame range.

Item	Description	Actual Moving Joints
Bike Pump	A small bicycle pump, pumping action	1, prismatic
Bionicle 1	A "Bionicle" series Lego toy, moving arms	2, ball-and-socket
Bionicle 2	A "Bionicle" series Lego toy, moving arm	1, ball-and-socket
Calipers	Analog calipers opening and closing	1, prismatic
D-Link	Wireless router with antenna, antenna being rotated	2, revolute
Droid Phone	Motorola Droid Phone opening and closing	1, prismatic
Falcon Toy	Falcon action figure, flapping wings	2, revolute
Helicopter	4 propeller helicopter, spinning propellers	4, revolute
Peace Sign	Fist to peace sign transition with human hand	6, revolute
1 Nut 1 Bolt	A nut spinning on a bolt	1, screw
2 Nuts 1 Bolt	Two nuts spinning on a bolt	2, screw
Pliers	A pair of pliers opening and closing	1, revolute
Robotic Arm	Multi-linked robotic arm, two joints moving	2, revolute
Single Slinky	A single Slinky tethered to two endpoints	1, prismatic
Double Slinky	Two Slinkys tethered to two endpoints and a midpoint	2, prismatic
Soap Dispenser	A screw rotating on the bottom of a soap dispenser	1, screw
Top	A top rotating on the ground with heavy precession	1, ball-and-socket

Table 1: Dataset Description

3 Algorithm

Here we outline the algorithms for joint location detection and joint classification, both of which build upon a link segmentation algorithm which we implement. We first discuss the link segmentation algorithm before discussing the detection and classification algorithms.

3.1 Link Segmentation

In order to determine any information of the kinematic structure of a given system, it is first imperative that we determine what "links" the objects possess, that is, the structures which are tethered by the joints. To do this, we require three steps – image segmentation, feature generation, and feature clustering. The former two happen during each time step, but the clustering only occurs after the algorithm has observed the entire video. We now consider each of these steps in turn.

3.1.1 Image Segmentation

The principal technique used for the feature generation step is feature tracking, which is very sensitive to abrupt variation in color intensities. We define the foreground as the structure we wish to learn about, and everything else as the background. Image segmentation of the scene was necessary for reducing the amount of noise in the system. Here, we define noise to be details of the structure that provide no information about system structure or background imagery. For instance, dirt on the camera lens, polka dots on a texture, or hair on a human forearm, none of which provide any information about the structure of the object, can add abrupt changes in color intensity to the system, which in turn can provide misleading information to the tracking algorithms. Additionally, background imagery, such as a shadow, which is not part of the object, can waste time and memory by causing our algorithms to track irrelevant parts of the image, or perhaps even mistake a point as moving from the foreground structure to the background. Image segmentation can eliminate localized but abrupt, noisy details, and further weaken the already gentle background gradients, thus reducing these sources of noise.

Of the various background segmentation algorithms we investigated, mean-shift segmentation was found to be the best, as it was a relatively fast method which removed most of the noise in our videos; we used the openCV [2] implementation with a spatial radius of 10 pixels and a color radius of 15 in standard RGB color space. An example of this background segmentation is shown in Figure 1a and 1b. Other segmentation algorithm possibilities are discussed in section 5. While segmentation was not an absolutely necessary step, we found empirically that it leads to more efficient use of tracking nodes.

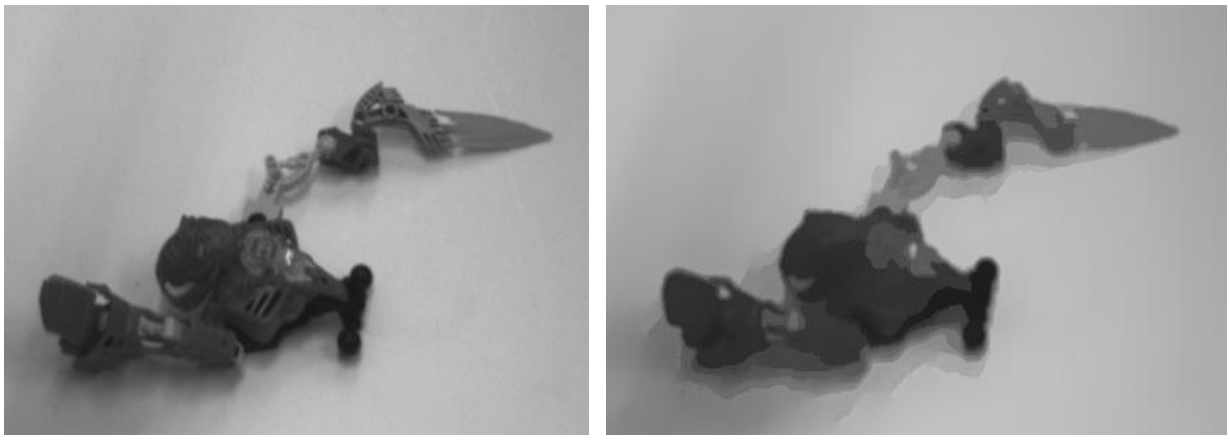

Figure 1a and 1b: Example of image segmentation. Original image on left and segmented image on right.

3.1.2. Feature Generation

Once the image segmentation has been completed, the algorithm moves to the step of cluster feature generation. First, features of the *object* must be tracked from frame to frame, which we do by first generating "nodes" on the image corresponding to points on the object to track. This sub-step only occurs at the first frame, although we discuss ways this can be modified in section 4. The points we choose to track occur at "corners," that is, where there are large color intensity gradients in orthogonal directions. For this, we employed the Shi-Tomasi algorithm. Figure 2 below shows an example of the nodes selected by the Shi-Tomasi algorithm.

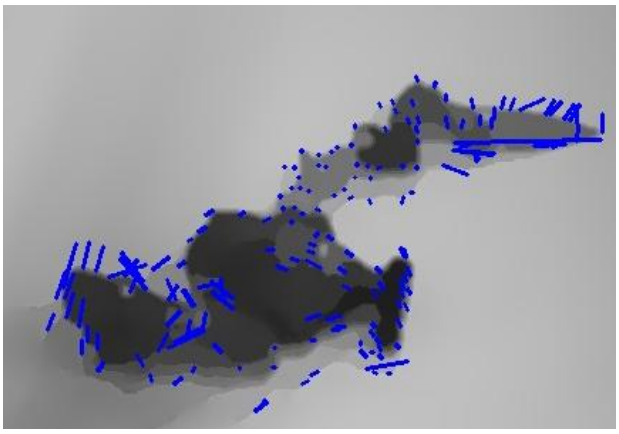

Figure 2: Example of Shi-Tomasi algorithm for feature generation (continuation of Figure 1)

After selecting the nodes to track, the Lucas-Kanade algorithm was used to track the location of the nodes from one frame to another. Unfortunately, this algorithm makes the assumption that points only move only within a small neighborhood and not far distances from frame to frame; since we generated our video dataset by using stop motion, this method was a little bit noisy. Nodes would sometimes "jump" and get lost on the background from frame to frame. We discuss potential future remedies to this problem in section 5. Both the Lucas-Kanade algorithm and the Shi-Tomasi algorithm operate on the 2-D image.

Once we have the nodes tracked, we must develop a feature vector for each node. The goal is to develop a feature vector for each node, describing its movement with respect to the other nodes. From this, we hope to eventually cluster the nodes by the links to which they belong.

We make a key assumption about nodes in order to aid with designing a suitable feature vector: if two nodes occupy the same link, then the 3-D distance between the two nodes *doesn't change* throughout the course of a video. We can weaken this assumption by saying that with noise, the distance between the two nodes *doesn't change very much*.

From here, the feature vector seems natural: we record the relative change in distance between each pair of nodes (including the reflexive pair) over all adjacent time frames. In other words, the feature vector $f(n_0)$ for a node n_0 in a system of N nodes and M time frames is:

$$f(n_0) = \begin{pmatrix} \Delta\left|\left|n_0 - n_0\right|\right|_2^{t_0,t_1}, \Delta\left|\left|n_0 - n_1\right|\right|_2^{t_0,t_1}, \dots, \Delta\left|\left|n_0 - n_N\right|\right|_N^{t_0,t_1}, \Delta\left|\left|n_0 - n_0\right|\right|_2^{t_1,t_2}, \Delta\left|\left|n_0 - n_1\right|\right|_2^{t_1,t_2}, \\ \dots \Delta\left|\left|n_0 - n_N\right|\right|_2^{t_1,t_2}, \dots\dots, \Delta\left|\left|n_0 - n_1\right|\right|_N^{t_{M-1},t_M}, \Delta\left|\left|n_0 - n_1\right|\right|_2^{t_{M-1},t_M}, \dots, \left|\Delta|n_0 - n_N|\right|_N^{t_{M-1},t_M} \end{pmatrix}$$

Here the superscripts denote the pair of frames over which the difference in distances is computed, and the subscripts denote that the 2-norm is taken. So, really, the first term for example represents

$$abs\left(\left|\left|n_0 - n_0\right|\right|_2 \ at\ t_1 - \ \left|\left|n_0 - n_0\right|\right|_2 \ at\ t_0\right).$$

The idea here is that (for the feature for node n_0, for example), for node i not on the same link as node n_0, there will exist some frame in the video (and hopefully many) on which the change in distance between the two is large, but if i is on the same index as n_0, it will be small at every time step. Since a dimension in the vector always corresponds to the change in distance with relation to a specific node at a specific time step, we are consistent in defining our relationship over all nodes, and the above argument thus holds for all nodes. Thus, we would get nodes with similar indices of large nonzero terms to be clustered together, which is exactly what we want. In total, this gives feature vectors of length NM - fine for many cases, but could cause memory problems for large N and M.

It's worth noting here that Oliver [1] uses a different methodology to determine the joint structure. Oliver measures the changes in the distance between nodes over time as well, but uses this data to generate a graph. Each node is represented by a vertex, and two vertices are connected by an edge if the change in the distance between their respective nodes is 0 for the entire sequence. From this constructed graph, a min-cut algorithm [3] is used to segment the graph into links. For us it was unclear how to decide how many times to run the min-cut algorithm here (although this is similar to our choice in k value for clustering, which we discuss shortly). Furthermore, we were worried about this method's sensitivity to recording noise. For example, what if it appears to the Lucas-Kanade algorithm that nodes on the same rigid are moving some very small distance? For this, it would be hard to generate the graph since we'd need to know an additional parameter, some tolerance ϵ on which we'd need to pass before we no longer consider two nodes' respective distances stationary. For these reasons, we opted to use clustering algorithms.

3.1.3. Feature Clustering

Using our nodes and generated features, the last step is to cluster the nodes. We use a spectral clustering algorithm based off the work of Ng et al [4]. In particular, we use the Spectral Library MATLAB implementation [5] [6] and use Ng spectral clustering in conjunction with Ward hierarchical clustering as a mapping method. Figure 3a below shows the nodes after they have been clustered. Figure 3b shows the corresponding clustering after K-Nearest Neighbor (KNN) has been employed to reduce noise, which is discussed in section 3.2.1.

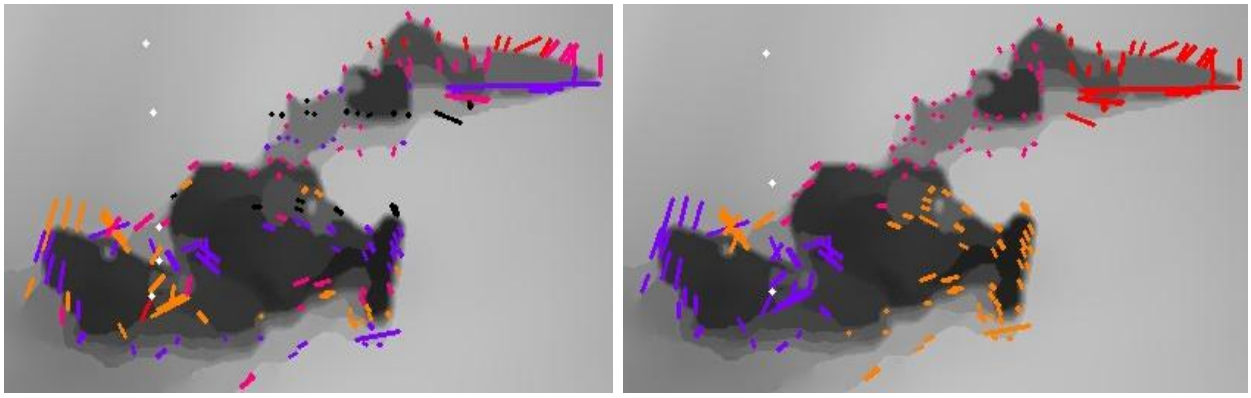

Figure 3a and 3b: Image on left is clustered segments directly after spectral clustering. Image on right is after clusters have been corrected using a k=9 value for KNN.

The one ingredient that we're missing for such a method is the dissimilarity matrix S. For this, we take our feature vectors, and define entry $S_{i,j}$ to be the 1-norm of $f(n_i) - f(n_j)$, with f defined as earlier. The one norm seems the most natural choice here, since we just want the total amount all the terms differ by; there is natural mapping to a Euclidian space.

One caveat is that the choice in k is a little unclear. Since we are writing this algorithm for the general case, we seek to overestimate the number of clusters. In practice, we've noticed that this has led to the same number of "big" clusters as if you were to use the most accurate k value with a priori knowledge of the system, and several singleton clusters. Most objects have approximately 2-3 joints. Therefore, suggesting that there will be 9 links is in fact an overestimate. This is the value we choose for k in practice.

3.2 Joint Detection

At this point the method we've outlined has allowed us to get a decent approximation of the various link segments, albeit with some noise because of the overestimate of the k value in clustering. At this stage, we wish to determine where the joints are located. Theoretically, a joint can link any pair of clusters, so for $k = 9$, there are 36 candidate locations for joints. We now present two steps for decreasing the number of joint candidate locations to consider, and give a simple method for approximating where the joint is located.

3.2.1 K-Nearest Neighbor Correcting

From the results of the clustering algorithm, it is often the case that, because of the combination of small clusters introduced from the overestimate in k and noise in the optical flow tracking algorithm (In particular, occlusion often led to the "jumping" of background tracking points onto moving foreground objects), that there are many small "subclusters" mixed in the true link clusters. We seek a way to correct for this and relabel the link as one cluster. Figure 3b above highlights the effect of KNN correcting.

Since it seemed that the outlier mode labels in our identified clusters were few often found in the middle of other clusters' node labels, we applied k-nearest neighbors to each data point, re-identifying its "true" cluster label as the majority of its k nearest neighbors. After all "true" labels were identified, the labels were updated; this was repeated for many epochs. The choice of repeating this for multiple epochs was because sometimes only a few outliers were removed on a given iteration, and thus several iterations were needed to eliminate all of the outliers. Empirically decided, 10 was chosen for the epoch number, and 9 was chosen for the k value in k-nearest neighbors. The distance metric used here was Euclidian distance in 2-D. 2-D was chosen for the node locations over the true 3-D locations since it was originally 2-D tracking which caused the noise.

3.2.2 Location Detection

In order to determine the location of the joints, all pairs of remaining clusters after the k-nearest neighbor correction step were considered. Then, the two closest points between those clusters was taken, and the 3-D average point of those two points was taken. For visualization purposes (although it has little to no purposes with respect to the world we are learning about) the point cloud was searched for the 3-D point that was closest to this midpoint, and the joint was drawn on the corresponding 2-D point. Lastly, any pair of chosen points whose distance was greater than a certain threshold value was considered to have no joint, as this meant the two clusters probably had little to nothing to do with one another. This procedure works well, making two assumptions. The first is that there is at least one tracking point from each cluster that is close to the joint; this is true for most objects, since most objects' structure extends around the joint region. The second is that, for the visualization process, that there is a pointcloud data point relatively close to the estimated joint location.

In some cases, this still led to the indication of nonexistent points. This is corrected for later in our joint classification step, by allowing for the potential types of joints to include "fixed joints," or joints that allow for no effective movement between its conjoining links.

3.3 Joint Classification

Once the joint locations have been found, the next step is joint classification. For this, we first need to acquire some features about the movement of the involved segment clusters.

We begin by choosing two clusters thought to have a joint between them, c_1, and c_2, determined from the previous step. From here, we choose one as the "anchor," say c_1, and the other as a moving piece. From here, we generate the relative position of c_2 with respect to c_1 by first calculating the centroids of the clusters, and then performing the following transformation:

$$x_{new} = x_2 - x_1$$
$$y_{new} = y_2 - y_1$$
$$z_{new} = z_2 - z_1$$

Where here, "new" denotes the transformed reference frame, the "2" subscript represents the original coordinates of c_2's centroid, and the "1" subscript represents the original coordinates of c_1's centroid. From here, the coordinates x_{new}, y_{new}, and z_{new} are converted to spherical coordinates. The maximum and minimum r, ϕ, and θ coordinates of the transformed system over the length of the video are recorded (with respect to the starting position of the system), and this six-tuple defines our feature set. These values can also be used to guess limits of motion, if the agent running this algorithm believes the motion of the studied object is limited. Features are then normalized over the dataset.

It is reasonable to believe that these features will define a space with a separating hyperplane. A fixed joint's features should be characterized by the maximum and minimum value of each coordinate being roughly equal (but none other). A revolute joint's features should be characterized by the maximum and minimum r and θ values being roughly equal (but none other). A ball and socket joint's features should be characterized by the maximum and minimum r values being roughly the same (but none other). A prismatic joint should be characterized by the maximum and minimum ϕ and θ values being roughly the same. A screw joint should be characterized by having its maximum and minimum ϕ values being roughly the same. We believe intuitively that a multiclass SVM will thus be able to use these characteristics about the data to generate separating hyperplanes, especially since the feature space is small (and SVMs scale better in small feature spaces). We use SVM-Multiclass [7] with the parameter c = 200, as a linear classifier.

We also briefly tried using a radial basis kernel, but got roughly the same results. There is also reason to believe that a semi-supervised clustering algorithm may be able to notice the difference between these joints based on the features set, but might not be as robust to changes in the original labeled set.

4 Results and Discussion

4.1 Speed

The algorithm worked at a very efficient pace, taking only about several minutes to cluster and classify an object. There were several phases that went into joint classification and detection. In the beginning, segmentation was performed, which took several seconds per frame. Feature generation also averaged several seconds per frame. Most of the computational time was spent accessing the 3D point cloud data, which contained a massive amount of data that needed to be scanned frequently. This computationally intensive task needed to be done in two different phases of the process, once to generate features from point tracking and a second time to find joint locations. In comparison, the spectral clustering and SVM algorithm only took several seconds to compute. Though this entire algorithm cannot be used to do real time joint detection and classification, some improvements can be made to optimize the speed, such as more efficiency scanning the 3D point cloud data, which could eventually allow for more real time detection and classification.

4.2 Cluster Accuracy and Joint Detection

Cluster accuracy plays a crucial role on the overall accuracy of the entire algorithm. By manually going through the dataset, and visually counting, the clustering had an accuracy of 55.5%, as it was able to often cluster the rigid bodies, but occasionally added unnecessary clusters to an object. Table 2 below shows the breakdown of the cluster accuracy.

There were some constant inconsistencies in the data that can be explained because of the sparseness of the point cloud data. When finding a point cloud point to correspond with the 2-D image, some exactness was lost going both from 2-D to 3-D and vice-versa. You may notice in the appendix that some joints seem carry the same structure as the object, but are offset by a certain amount. It is the authors' best guess this happens because of the point cloud and 2-D image correspondence.

All in all, all joints were found correctly 10/18 times, more than half. We defined a correct finding of a joint as any time the algorithm found all of the joints, but keep in mind that this means the algorithm could find more than just the correct joints. This then becomes the responsibility of the SVM to detect that these are "fixed" joints and thus not real joints. 10/18 is a slightly disappointing rate, but we believe that by improving the tracking that we could improve the precision of the clustering, which would in turn improve the finding of the joints. We also note that there seems to be little bias as to the accuracy of finding the joints per *type* of joint, suggesting that this approach could be effective for all types of joints.

Item	Found all joints correctly?	Segmented correctly?
Bike Pump	Yes	No
Bionicle 1	No	No
Bionicle 2	Yes	Yes
Calipers	No	Yes
D-Link	Yes	Yes
Droid Phone	Yes	No
Falcon Toy	No	No
Helicopter	No	No
Peace Sign	No	No
1 Nut 1 Bolt	No	Yes
2 Nuts 1 Bolt	Yes	Yes
Pliers	Yes	No
Robotic Arm	Yes	No
Single Slinky	Yes	Yes
Double Slinky	Yes	No
Soap Dispenser	Yes	No
Top	No	No

Table 2: Cluster and joint detection accuracies

Note: For a complete graphical description of the results, please see the appendix, where we provide "ball and stick" models of our results.

The segmenting results, as described in the table above, looks a little alarming – if segmenting did so poorly, then why did joint detection do so well? Was it just dumb luck on the joint detection part? Well, the fact is that while the table says that segmenting was only completely correct on 7/18 items, it only completely failed on 3 of them: Falcon Toy, Top, and Peace Sign. Top and Peace Sign were labeled completely as one cluster, and Falcon was a patchwork of bunches of clusters. These were probably due to bad data collection – the Falcon Toy and Top motions were too jagged and spaced apart for the tracking to perform accurately on, while the Peace Sign might have failed because it is hard to discern where a finger ends and the hand begins. On the other 8 objects that the joint segmentation failed on, the error was very small. By this, we mean that there existed only one segment or partial piece of a segment in each of these videos that had the same classification as another, unrelated segment in that video. Figure 4 below is another example of the clustering after KNN correcting on the double slinky item.

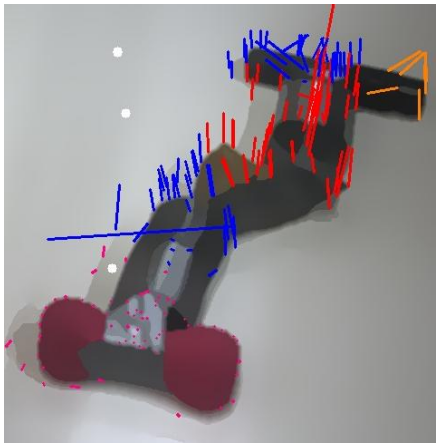

Figure 4: Double Slinky clustering

Here, the distinction between adjacent clusters is clear and accurate – except the blue cluster label is used twice for two unrelated clusters. If we increased the number of clusters, or decreased the noise elsewhere in the image (this is a cropping of a larger image with more background structures), then we believe that the clustering might perform

better here. Another solution is to note that the error didn't matter in many cases – sometimes, the same color was used for two distant clusters that showed either no movement (often the case) or the same, small movement during the video. In this case, they would be clustered together by the algorithm – a fair call, since there is no clue other than proximity that they are not related somehow. While it might be less likely that distant segments are somehow connected, it might not be worthwhile to take that guess.

4.3 Joint Classification

The collected data was split into a training set, consisting of 35 pieces of data, and a test set consisting of 5 pieces of data. Leave-One-Out-Cross-Validation was used to find an optimal value for *c*, the cost for misclassified points to be used in the SVM algorithm. The best accuracy that was found on the training set was 48.57% using a *c* value of 200. Using this learned model with our optimal *c* value on the test set yielded 2 correctly classified joints and 3 incorrectly classified joints, yielding a 40.0% accuracy on the test set. This error is not bad with the caveat of the small size of the test set. With a baseline of 20% accuracy (random selection), this algorithm performs well in comparison. Many things can be done to possibly increase this value, including building a larger training set, which would allow for more robust training and testing.

One additional thing to note is the similarly of certain joints and difficulty of correctly capturing the necessary movement. Take a screw joint, for example. Without precise measurements, it may just be seen as an image moving forward and back without any spin, making it look more like a prismatic joint (in fact, we have some cases where this occurred). This is an example of the type of error the algorithm can give due to poor data.

5 Future Work

As mentioned previously, there were two main considerations for improving our results, which we discuss here, with hope that they be explored in the future.

First, while we settled on mean shift filtering as an image segmentation method, there may be other segmentation methods that will work as well or better. In particular, we think that Felzenszwalb and Huttenlöcher's graph-based segmentation algorithm would work particularly well [8]. It has the property that each segment of the image is colored a particular color, causing sharp jumps in color intensity. These would likely be chosen by the Shi-Tomasi algorithm as good features to track, and would be easier for the Lucas-Kanade optical flow algorithm to follow, thus resulting in overall better feature tracking.

Unfortunately, while segmentation would aid in the precision of the tracking, it would not be a panacea for this algorithm's tracking woes – it still would be thwarted by the occlusion of links, and would likely still have "jumping" nodes if the frame rate were still poor or if the motion appeared jumpy due to the angles observed. Obviously, improving the frame rate would be very desirable. An even more preferable solution would be to incorporate some form of 3-D tracking, where the point-cloud itself is tracked. Unfortunately, one does not have control of which points are recorded by the stereocamera; the Bumblebee2 software makes the choice for the user. While 3-D tracking of point clouds is difficult on its own, with the tracked features changing on each frame, this problem could become difficult or impossible using the current software. We hope, however, that someone can come up with some method, perhaps using different software, which is able to use 3-D point cloud tracking in order to improve the algorithm.

Another possible problem with the SVM classification might have been with our choice in feature space. We used θ and ϕ to denote the angular movement in spherical coordinates of cluster centroids. It might be though that the axes for these θ and ϕ movements are not well aligned. In that case, we should consider different types of bases, such as quarternions and different rotation matrices.

One last idea is that the features get a little distorted over time due to the "jumping" effect described earlier. One possible solution is to come up with a heuristic for restarting the algorithm every ten or so frames, and then finding a way to match up the results from one set of frames with another. This would

keep the feature tracing accurate; of course, the difficulty is determining how often to restart it and how to tell if a joint detected in one set of frames matches a joint found in a previous set of frames.

6 Conclusion

We have presented a new method for both detecting and classifying the joints of an object from around 15-25 frames of stereoscopic video. The use of spectral clustering, as a means of determining the segments and joints, allowed for a completely unsupervised method of using this data to build a kinematic skeleton of the studied object. Meanwhile, the use of a multiclass SVM for classification of joints allowed for a relatively fast and robust classification over many candidate joint types, as well as filtering out any mislabeled joint candidates surviving the link segmentation. Still, the results were far from perfect. Most of the error is believed to have come from errors in the tracking. Unfortunately, since our tracking algorithms are 2-D they are very sensitive to abrupt jumps in position by frame and even slight occlusion – problems with stop motion methodology and difficulty in tracking over dynamically determined point clouds. Still, we believe that with improvements to the tracking algorithms using 3-D data, combined with heuristics for interacting with the surrounding environment, that this will become a very viable method for quickly building a kinematic model of the world.

Acknowledgments

We would like to thank Ashutosh Saxena for his advice and guidance in this project. We would also like to thank Andrew Perrault and Stephen Moseson for their help in operating the Bumblebee2 stereocamera. Lastly, we would like to thank Rohan Sharma and David Skiff for their peer reviews of this paper's preliminary draft.

References

[1] O. Brock and D. Katz. *Extracting Planar Kinematic Models Using Interactive Perception*. 2007

[2] Intel. http://www.intel.com/technology/computing/opencv/.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to Algorithms. MIT Press and McGraw-Hill, 2001.

[4] Ng, Jordan, et al. *On Spectral Clustering: Analysis and an Algorithm*. 2001.

[5] SpectraLIB - Package for symmetric spectral clustering written by Deepak Verma. Updated by Tatiana Maravina.

[6] MATLAB version 7.8.0. Natick, Massachusetts: The MathWorks Inc., 2009.

[7] T. Joachims. *Support Vector Machine for Complex Outputs* version 2.00. 2004

[8] P. Felzenszwalb and D. Huttenlocher. *Efficient Graph-Based Image Segmentation.* International Journal of Computer Vision, Volume 59, Number 2, September 2004.

APPENDIX

White lines indicate links between clusters and squares indicate joints. The color indicates the joint label, as shown in the legend. The images are provided left to right and top to bottom in the following order: Calipers, Top, Bike Pump, D-Link, Bionicle 1, Bionicle 2, Droid, Double Slink, Helicopter, Falcon Toy, Soap Dispenser, Nut and Bolt 1, Nut and Bolt 2, Peace Sign, Super Armatron, Pliers, and Single Slinky.