Towards Camera Choreography: Physically-constrained Multi-camera Clustering

Eleanor Tursman James Tompkin
Brown University

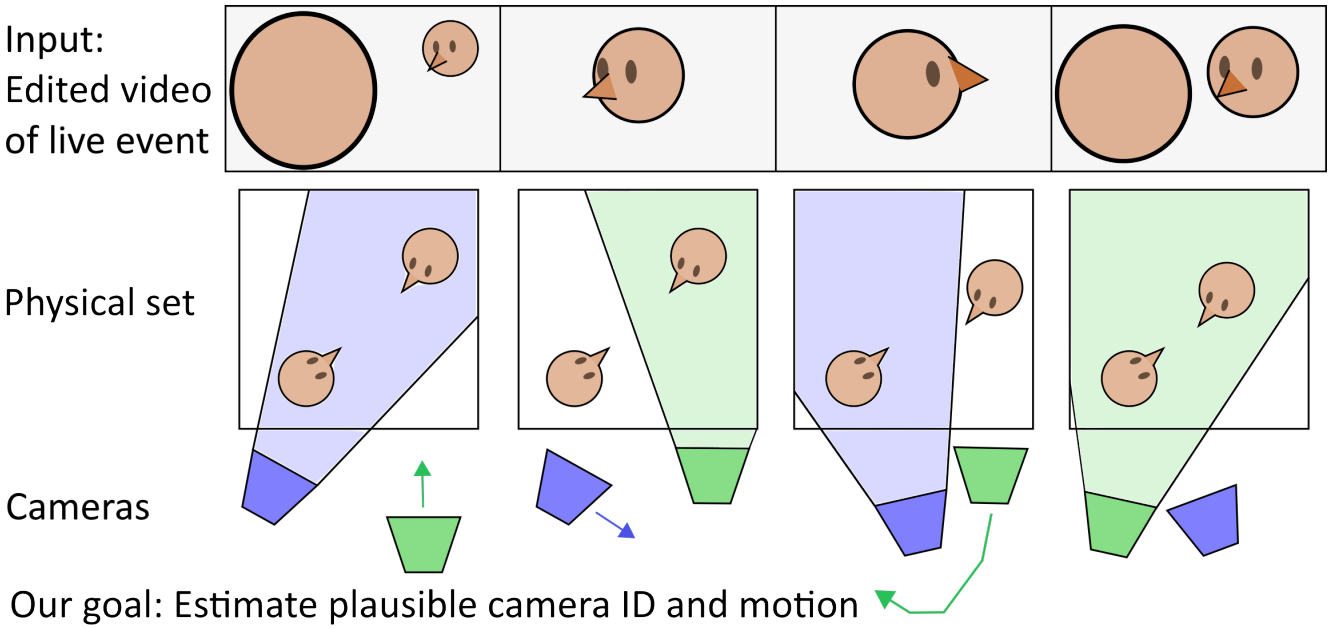

Figure 1. Our task. *Top:* Edited video shots of two people; the event was captured live. *Middle:* A top down view of the physical set. *Bottom:* A top down view of the camera capture setup. As one camera is shooting, the other moves into position such that only two cameras are needed to capture the event. The goal of our work is to cluster camera identities and assign motions to an edited video such that the real world hidden motion of the cameras is physically feasible.

Abstract

We present a method to cluster shots by camera identity in a synthetic 2D space which simulates edited video input. We show that our methodology isolates the space of physically feasible camera movements given a desired edited output, with the goal of minimizing overall camera motion. The problem space appears at first glance to be under-constrained, as the video is discontinuous due to edits, and is shot using an unknown number of uncalibrated cameras. We create a model to generate viable random synthetic 2D datasets of multi-camera shooting setups, and show that by using physical shooting constraints and various path planning methods, we are able to calculate a space of feasible camera clusterings and their associated movements between shots. We compare greedy and dynamic clustering methods which use linear and piece-wise linear trajectories. We find that the dynamic linear method yields the largest number of clusterings with costs less than or equal to that of the ground truth, and is one of the best methods for capturing clusterings with minimal motion. The dynamic linear method is able to sufficiently capture the underlying motion of the cameras to create good clusters. This work contributes towards making it possible both to synthesize plausible camera motions given a set of shots with target camera positions, and to infer plausible camera identities and hidden motions from an edited video input.

1. Introduction

Expansive video data are accessible to the general public, with many of these recording live events. Theater, sitcoms with studio audiences, late night talk-shows, ballet and opera, and other productions recorded live typically use multi-camera setups, where all cameras are shooting concurrently, and where their feeds are edited together in post or in real time. To effectively analyze and manipulate these datasets, we typically isolate important moments in each video by first splitting it into a series of shots and scenes whose image content can then be parsed. Using the image content within a shot, we can recover camera pose, or extrinsics, through structure from motion [4]. However, this set of camera poses gives us no information about camera identity, nor the motion of the cameras outside of the edit sequence. Camera choreography, or planning camera movement to recreate a desired edited look, is not possible without this information.

Intuitively, as we watch multi-camera video with continuous time across edits, the cuts between shots give us a physical sense of the camera positioning—the changing camera extrinsics between and within shots provides an impression of the camera setup and camera movement around the set. We aim to algorithmically exploit this intuition to recreate feasible camera setups for edited video input.

Given some edited video of a live or live-captured event, we aim to infer the way that the video was shot (Figure 1). To accomplish this goal, we need to determine how many cameras were used to shoot the edited video, and which camera was responsible for which shot. The initial problem space is under-constrained, since our edited video input has been shot by an unknown number of uncalibrated cameras, and undergo unknown motion while they are not actively shooting in the edit sequence. Because of these issues, we are unable to reliably isolate the ground truth sequence of camera assignments to each shot. Instead, we carve out the space of *possible* camera assignments, where a possible sequence of assignments has to have a physically feasible set of camera motion paths during shooting.

Problem Statement Suppose we have some sequence of n shots $s_1, \ldots, s_n$ with shot lengths $t_1, \ldots, t_n$, known camera intrinsic and extrinsic poses $p_1, \ldots, p_n$ for the currently-viewing camera, but an unknown number of cameras k and unknown hidden camera motions for not-currently-viewing cameras $h_{k1}, \ldots, h_{kn}$. We wish to assign a camera identity to each shot and generate a plausible hidden camera motion.

In essence, when we make a sequence of assignments, we are clustering shots by camera identity (Figure 2). We use path planning as our clustering evaluation metric, since a given clustering is only possible if the cameras can be pushed into position in time to shoot. For each shot s_j, we determine the validity and path length of assigning it some camera c, given the camera's previously known location in time and space s_i. As we create a path, we are constrained by the field of view of the active camera and by the shot lengths. Over the course of our camera assignments, we aim to minimize the overall necessary movement to hit our target camera poses at each of the n shots, and then rank possible clusterings using overall distance travelled as a cost.

Our method relies on edited video input which is pre-processed in two ways. First, the we assume video is separated into a sequence of shots. Second, we assume that estimated camera intrinsics and extrinsics exist per shot (e.g., using feature point correspondence). Given this pre-processed video, we aim to assign a camera identity to each shot, which we formulate as a clustering problem.

We circumvent these pre-processing requirements by creating synthetic datasets that start with this information. We randomly generate 2D synthetic datasets which have camera assignments, intrinsics, extrinsics, and lengths per shot. The intrinsics, extrinsics, and shot lengths are then passed through our clustering process. Clustering shots by camera identity is constrained by the physical feasibility of creating paths between target camera positions. These possible clusterings are then ranked based on minimizing overall camera motion.

Applications Once we have assigned camera identities per shot and have calculated physically-feasible camera motion paths, we have two major application possibilities: synthesis or forward applications, and analysis or backwards applications. By synthesizing our results, we can design camera choreography that achieves a desired edit look, e.g., by meeting a set of user-specified constraints within a shot sequence design. By analyzing an existing edited video, we can potentially learn how to shoot in a particular style.

Contributions These constitute synthetic 2D multi-camera dataset generation, and a method for isolating the space of physically feasible camera clusterings while minimizing overall camera motion.

2. Related Work

We cluster our shots based on camera identity, using how these cameras could possibly move around our set as our clustering metric. To establish a choice for this clustering metric, we look to path planning methods from robotics. Since there is no direct analogue for clustering given this particular problem setup, we pull from related work which tackles similar clustering setups.

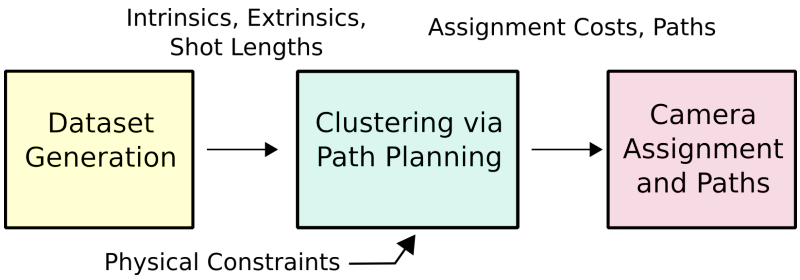

Figure 2. We generate 2D synthetic data with known camera assignments per shot. We make the assumption that we would be able to procure this information through structure from motion and self-calibration in a real world scenario. Then, we pass this information to our clustering algorithm. We use physically feasible path planning as our clustering metric. We rank each clustering of camera identities to shots by minimizing the overall distance travelled by our cameras while path planning.

2.1. Path Planning

In robotics, path planning methods vary depending on how the space is both represented and traversed. In this work, we use both sampling and model based methods, which either discretely sample the space, looking for obstacles, or determine how to reach a goal by modeling object kinematics [10]. LaValle's Rapidly-Exploring Random Tree (RRT) is a popular sampling method which uses a randomized graph [6]. Different branches grow off of the tree and explore the regions that are reachable from the root position. While the shape of the tree is dependent on the sampling method we use for our space, RRT can converge more quickly than an exhaustive search of a given physical space. We use RRT with cumulative distance travelled from the root as edge weights as one of our path planning metrics. Masehian and Habibi proposed tessellating space using Delaunay triangulation and take an average of selected triangles to find a path from a target to a goal in 2D space [7]. However, we are able to sufficiently explore our space using either piece-wise linear exhaustive searches or RRTs, which do not involve pre-processing the search space.

2.2. Clustering

Clustering methods are principally subdivided by data representation and distance or similarity metrics [9]. The major class of methods most related to this work are those created for spatial-temporal data, like Birant and Kut's ST-DBSCAN algorithm [2]. ST-DBSCAN is a version of the density based clustering algorithm DBSCAN [3], which is modified to handle spatial-temporal data, like regional temperature data taken at different times of day. In our problem space, we cannot run a clustering method like ST-DBSCAN directly on tuples of the position and time associated with each shot, because our goal is not to group shots that are similar in length or physically close to one another, but to associate shots with camera identities, given position and time information. These genres of clustering algorithm will assign some number of cameras to points that are close to one another physically, ignoring the specific constraints of our problem space– for example, a camera cannot shoot twice in a row since we see an edit. We do pull from the core concept in these algorithms that points in a cluster have to be "densely reachable," but we parameterize this concept literally by using path planning to determine if we can hit a target position in a given amount of time.

Instead of clustering spatial points, Yuan et al. examine major methods for clustering movement behavior [11]. Much like traditional clustering, trajectory clustering relies on trajectory similarity metrics. The methods and metrics that they study are about clustering paths that look similar, whereas in our problem space the shape of a given camera path is not important. Rather, we care about its start, intermediary targets, and the physical feasibility of a path through that set of points.

Kheirkhah and Khansari create clusters of wireless cameras based on detecting overlapping fields of view in order minimize energy consumption by reducing overlap without sacrificing coverage [5]. They create their clusters greedily, adding new nodes depending on how they correlate with existing clusters. Ala-Eddine et al. approach the problem of clustering wireless cameras by instead creating maximum cliques, where each camera is a node, and each clique has a high degree of field of view overlap [1]. It is not sufficient for us to use overlapping field of view as our main clustering metric, since unlike the stationary setup presented in these two papers, our cameras can move between recorded shots. Initially, we follow the rough methodology of these works with our greedy clustering setup, where clusters correspond to camera identity, though we use a different metric that corresponds to the feasibility of a given camera movement. However, we find that this is not sufficient for our problem space, where many possible clusterings will be able to reproduce a given edited sequence, which is why we expand our search space using a dynamic method.

Zaïne and Lee's work on clustering given physical constraints is closely related to our problem space, since in addition to density based clustering of spatial data, they model physical constraints which need to be avoided [12]. While

we have temporal data and changing physical constraints as the shooting camera changes, we follow their core method of representing physical constraints as polygons in space that cannot be intersected by our data.

Moving outside the realm of computer vision, Shahab Pasha's thesis work on clustering audio signals of an uncalibrated, moving array of microphones is similar to our own problem, in that both scenarios have uncalibrated sensors with unknown positions moving through some fixed space [8]. Given known room geometry, they localize the sources of acoustic signals by using time delays in impulses and by looking at the amount of noise in the resulting signals. However, unlike us, they start with a known number of sensors, and have access to all of the data recorded from these sensors at all points in time. If given an edited input of audio signals, they would not be able to cluster their audio signals with their current methodology.

2.3. Limitations of Existing Clustering Approaches

To cluster our shots by camera identity, our problem space requires a more specialized metric of comparison than the methods presented thus far. We cannot use a density based algorithm and cluster shots that have physically or temporally close positions, since depending on when those positions arise in our sequence, moving a camera between those close positions may cross the field of view of an actively shooting camera. Our data points are codependent—deciding how to cluster one point will affect how future points are clustered. Therefore, while we adopt certain elements of existing clustering work, we approach this problem from a different angle via path planning.

3. Method

We aim to cluster shots using the physical constraints of a real-world setup, since without taking advantage of the way that multi-camera video is shot, we do not have enough information from the structure from motion and self-calibration shot by shot results to constrain the problem space. In other words, since all cameras in this kind of setup are shooting at all times, then have their feeds edited together for final assembly, cameras need to be pushed into position while the scene is being shot. While one camera is shooting a close up, another camera is being moved into position for the next shot. We can take advantage of this information to determine how likely it is for a new shot with some extrinsics to be an existing camera in the scene. Since to our knowledge, there are not any datasets with ground truth for tackling this type of problem, we must construct our own data with ground truth for validation purposes.

3.1. Synthetic 2D Dataset Creation

Let C be a function parameterized by integers n and k, such that $C(n, k)$ produces a series of n shots $s_1, ..., s_n$, using k cameras $c_1, ..., c_k$. We define each shot $s_i = (c_{j,i}, p_i, f_{j,i}, t_i, a_i)$, where $j \in [1, k]$, p_i is the camera position, $f_{j,i}$ is the focal length associated with the camera c_j, t_i is the number of frames in shot i, and a_i defines the set of feasible movement area for cameras c_1,...,c_k at shot s_i.

The function C creates a series of shots following the subsequent constraints:

1. no one camera can appear for two assignments in a row, so $\forall i \in [2, k]$, given $c_{j,i-1}$ and $c_{l,i}$, and $j, l \in [1, k]$, $j \neq l$,
2. p_i cannot be in the field of view of the actively shooting camera $c_{j,i-1}$,
3. and the most recent positions of all cameras in $s_1, ...s_{i-1}$ cannot be in the field of view of $c_{j,i-1}$.

The first constraint is necessary because an edit between two shots would not happen if we were not moving to a new camera. The second constraint keeps the current camera from moving into the actively shooting camera's field of view. The field of view is geometrically calculated using the sensor width of the camera and its focal length. The third constraint ensures that when the current camera is active in s_{i+1}, no other cameras will be blocking its field of view.

When a camera first appears in the sequence, we initialize its position in an arc facing the positive y direction. This enforces that the cameras are initially spread some distance apart, and are all facing the "set." Intuitively, if a camera has appeared before, we need to determine if it would have been able to reach this new position given our real world physical constraints and time. Each existing camera has an associated area of feasible movement for every given shot i, a_i. Assuming for simplicity that cameras can move with a maximum velocity of one meter per second, our distance travelled is equivalent to the time we have to move a camera. For every shot where a camera is on set, but is not actively shooting, we increase its area at shot i, creating a_i, by drawing circles of radius t_{i-1} around each vertex on the current boundary of a_{i-1}. We sample points along each circle by intervals of $\frac{\pi}{16}$. If this area intersects with the field of view of the actively shooting camera, as per item two listed above, we take the intersection of this circle and the constraint. We calculate a new position for our camera by picking a random point from within its associated feasible area of movement. After calculating the new position of the camera, we randomly rotate it such that it is still facing the set, and we reset its feasible area of movement. Figure 3 provides an example of one randomly generated datasets.

3.2. Camera Clustering

Given a dataset generated via C, our goal is to take all the information in each s_i except the ground truth camera assignment, and arrive at a physically plausible attribution

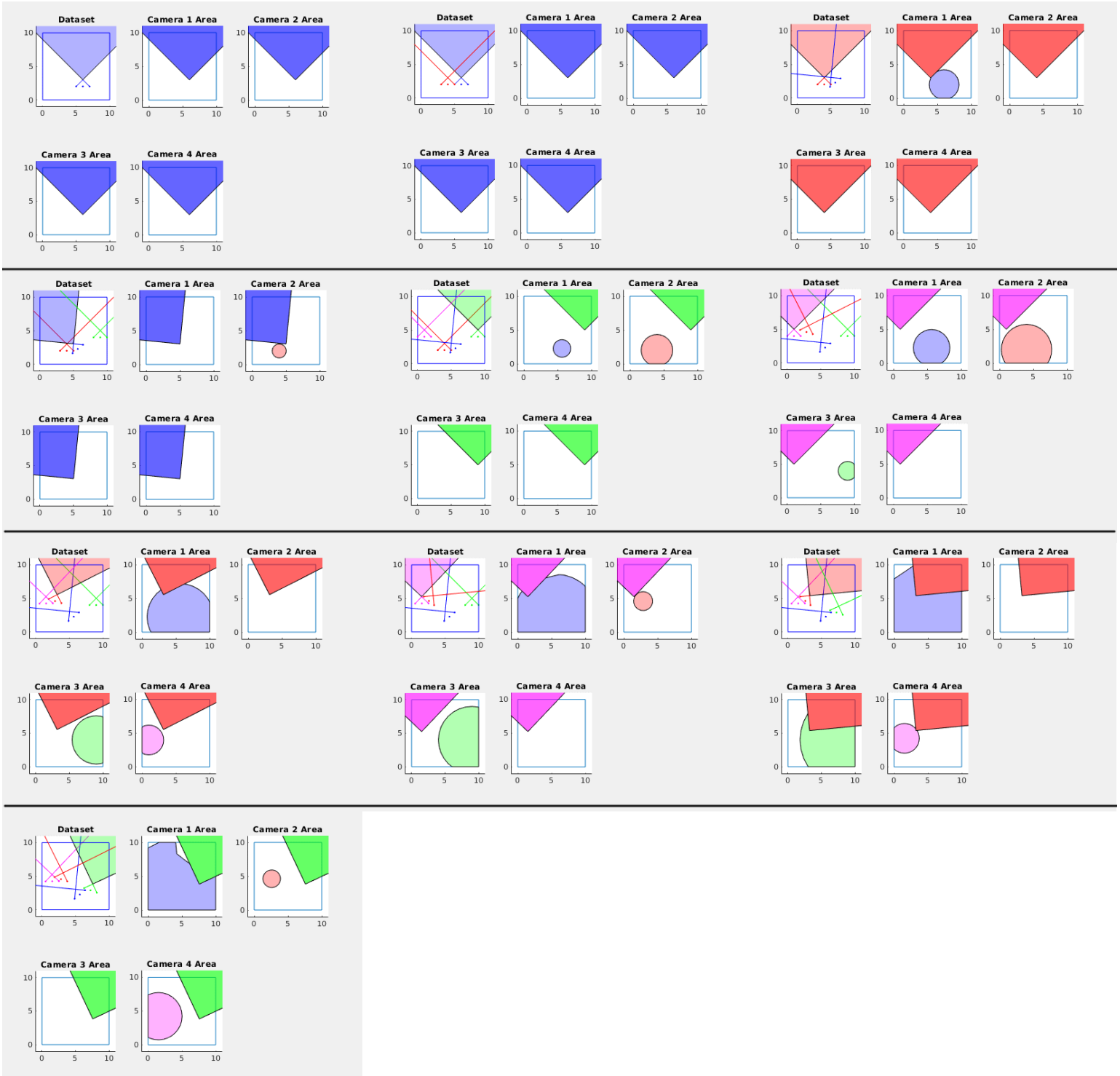

Figure 3. We generate a dataset $C(10, 4)$. From left to right, top to bottom, we have shots one through ten. In each image, a camera is represented by a 1D image plane, and each camera's field of view is denoted by a triangular space. A field of view is filled when that camera is actively shooting. Cameras are colored based on identity. The remaining set of images represent the physically feasible area of movement, a_i, for each shot i. Note how when a camera is actively shooting, its area is reset. Otherwise, as a camera remains still, its area of feasible movement grows. Relative area growth is determined by shot length. The blue square represents the space the cameras can move around in. Note that in the last shot, the feasible area of camera one has become warped due to its intersection in the previous shot with the field of view of camera two.

Clustering Type	Path Planning
Greedy	Line Piece-wise Line RRT
Dynamic	Line Piece-wise Line RRT

Table 1. We evaluate three different path planning metrics for both greedy and dynamic clustering methods.

of cameras to shots. We pose this as a clustering problem, where we cluster shots by camera identity. We present and compare two clustering algorithms, one greedy, and the other dynamic, and evaluate the use of three separate path planning methods for each one (see Table 1).

3.2.1 Greedy Clustering

The greedy clustering method chooses the camera assignment per shot that immediately minimizes the amount of overall camera movement while hitting the target positions. We start by assuming the first two shots are from distinct cameras, since there is an edit between them. Let our current set of cameras in the scene be $\hat{c}$. For all subsequent shots s_i, $i \in [3, n]$, we minimize the following energy,

$$E_i = argmin_{x \in \hat{c}, x \neq \hat{c}_{i-1}} \phi(p_x, p_i, \hat{c}_{i-1}) \quad (1)$$

where ϕ is our path planning cost function, p_x is the position of some camera x in the set of known scene cameras $\hat{c}$, p_i is the position of the unattributed camera of the current shot, and $\hat{c}_{i-1}$ is the field of view constraint defined by the position and focal length of the previously shooting camera. Intuitively, we take the last positions of all cameras we have placed in the scene so far and see how long it takes for each of them to reach the newest position, while avoiding the field of view of the actively shooting camera. The currently shooting camera cannot be assigned to this new shot.

E_i gives us a camera assignment and the length of the path this camera takes from p_x to p_i. Assuming a maximum velocity of one meter per second, distance and time are proportional. Then, we compare this path length d to t_{i-1}, the length of the previous shot. If $d \leq t_{i-1}$, it is physically feasible for that camera to travel to the new position given our stated physical constraints, and we assign this camera to s_i. If $d > t_{i-1}$, s_i must be made by a new camera, and we add a new camera with starting position p_i to $\hat{c}$.

Our feasible movement areas allow us to assume that cameras can move at any time unless they are actively shooting. Our greedy method first calculates the number of shots in which a given camera could be moving, or $M = i - x - 1$, where i is the index of the current shot, and x is the index of the last shot the given camera was active. We now use either straight line distance, piece-wise line distance, or RRT as our ϕ. Note that for all of these methods, the constraint region formed by the currently shooting camera's field of view changes with every shot.

When ϕ is straight line distance, we accumulate distance M times, travelling along the line connecting p_x to p_i. At each step, we stop either when we hit the wall of a field of view constraint, when we've run out of travel time, or when we reach p_x. We throw away the path if at any given step, the current line segment is completely inside a constraint, since physically this would entail our moving camera being seen by another actively shooting camera.

When ϕ is piece-wise line distance, we again accumulate distance M times. We create a graph of possible movement, where each node is a position, and each edge is weighted with the L2 distance between its endpoints. From our starting position, p_x, we draw a polygon of feasible travel area, just like we do in dataset generation, with intervals of $\frac{\pi}{16}$ between polygon vertices. We connect the node p_x to all vertices that it can reach. For each new shot, we expand our polygon representing our feasible travel area. Unless p_i is within reach during the current shot, we exhaustively draw edges from every existing node to any new polygon vertex that can be physically reached. Otherwise, we connect all nodes within reach to p_i. We connect all possible nodes, not only the outermost nodes, as we want to simulate paths where the camera is still for a shot, or travel away from the goal position to get around a constraint. To obtain our best path from p_x to p_i, we take the shortest path along the graph.

When ϕ uses RRT, and we have M shots in which to move, we create a tree to sample our feasible movement area. However, unlike in the piece-wise line case, RRT is less exhaustive. We uniformly randomly sample the "set" space 90% of the time, and sample p_i, our goal, 10% of the time. We find the closest node in the tree to this sampled point, and travel some Δd distance towards it, creating a new node. We weight the edges of this tree based on L2 distance between endpoint nodes, except this time we accrue distance travelled along a branch. We stop our tree from growing into the constraint area of each shot, and cap the total length of a given branch based on total accrued travel time of our M shots. If the goal is in reach of a node, instead of moving Δd towards it, we connect directly to it. We grow the same tree for all M shots. Note that while our modification of RRT is not guaranteed to find the shortest path from p_x to p_i, it is able to get around constraints, and is faster than the exhaustive piece-wise metric. In our implementation, we add 100 nodes to the tree per shot, and use $\Delta d = 0.01$. See Figure 4 for an example of the differences between an RRT tree and a piece-wise line graph.

Clustering the cameras greedily will keep us from underfitting, since we start with two cameras and only increase

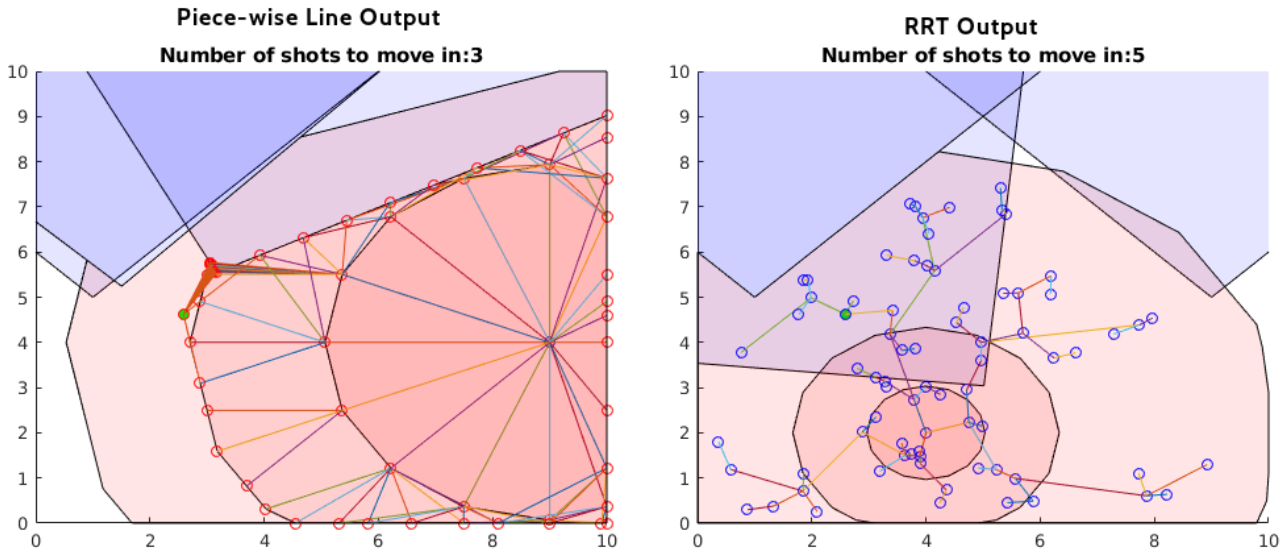

Figure 4. On the left hand side, we have an example of piece-wise line output. On the right hand side, we have an example of RRT output. For visual clarity, the circular sampling in these cases is reduced to $\frac{\pi}{8}$, and the Δd of RRT is increased to 0.5. In both images, a feasible area in red grows three times, and each blue triangle represents a constraint for one of those three shots. The starting point is the center of each bullseye, and the target is the bright green point.

the number of camera clusters when we have a point an existing camera could not physically travel to within the shot time. However, the best camera assignment shot to shot does not guarantee the best overall assignment. For this reason, we also explore clustering the cameras dynamically.

3.2.2 Dynamic Clustering

Our dynamic clustering algorithm computes the cost of every possible valid sequence of k cameras over n shots. We incrementally build a set of sub-sequences each starting from the first shot. A valid sub-sequence has an associated cost, while an invalid sub-sequence has a cost of $-\infty$. Adding a new camera assignment to a valid sub-sequence will produce a new valid sub-sequence with an incrementally calculated cost, or an invalid sub-sequence. We do not add new camera assignments to invalid sub-sequences, which can cut off large branches of sequence calculation.

Through this process, we are able to look through all possible sequences of our cameras without having to constantly recompute travel costs. Additionally, since the positions and constraints at each shot are fixed, we are able to memoize our cost function by these constraints, greatly reducing computation time. However, unlike in the greedy case, we need to give our dynamic clustering algorithm a maximum number of cameras k.

Given some k, we recursively explore all valid sequences that are n shots long. Each time we add a camera to a sequence, we accrue cost, where we travel towards a goal p_i for M shots using either straight line, piece-wise line, or RRT path planning. All of these metrics work in the same way as they do in the greedy algorithm described above, except we reformat them slightly to memoize them.

3.2.3 Edge Cases and Assumptions

Since positions are fixed at every shot, given a particular camera clustering, it is possible that the most recent position of an existing camera on the set will be in the field of view corresponding to the current shot. While it is physically possible to have that existing camera move out of the way, there is no simple principled way to determine how to move it within our framework, so we throw away sequences that violate this condition. In the greedy algorithm case, this means we have no output; in the dynamic algorithm case, this means that the number of total outputs will be reduced.

In a rare edge case, if the current shot is very short, it is possible for the entire camera sensor width to not be able to fit in the feasible area. For this reason, we only count the center point of the camera sensor as the physical "camera position." In other words, only that center point is not allowed to be in other cameras' fields of view.

Finally, we make the assumption that overall distance travelled by our cameras is a good metric to minimize to determine the set of feasible clusterings. We do this because while we could theoretically have many circuitous camera paths, in practice we hope to do the least amount of work, or moving around, as possible.

Table 2. We calculate the empirical runtime of each path planning metric by running our dynamic method ten times on ten datasets, timing our result, and taking an average. We formulate the average big O runtimes for the path planning methods, where k is our number of cameras, n is the number of shots, r is the sampling rate of the polygon expansion, and p is the number of nodes we add at each call to RRT. While our empirical runtime for RRT is slowest, it has a better overall runtime complexity than that of piece-wise lines.

Path Planning	Empirical Runtime (s)	Complexity
Line	0.59	$O(n \cdot k)$
Piece-wise Line	139.79	$O(n \cdot r^k)$
RRT	149.64	$O(n \cdot p \cdot k)$

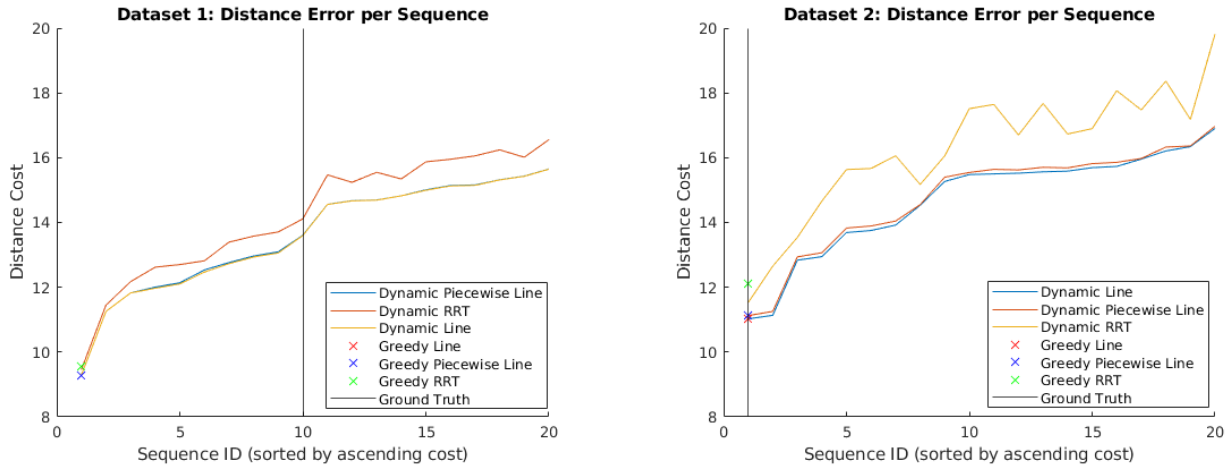

Figure 5. Each integer on the x-axis represents a different unique sequence of camera assignments for all shots. We align our data such that, for each value on the x-axis, we are comparing the distance cost of a specific sequence for all methods. Not all methods pick up the same total set of possible sequences, so there are some implicit zeroes in each line plot. Each image represents results from a different dataset. While the greedy RRT method tends to be the most expensive, all greedy methods choose the same output sequence. The distance cost of the dynamic methods varies depending on the dataset. The RRT metric is always the most expensive.

4. Results

We evaluate output based on counting the number of sequences with costs at least as good as that of the ground truth sequence, since we care about finding the largest number of feasible low cost sequences. We also look at the normalized difference in cost between the best sequence of each method and the cost of the ground truth sequence. More precisely, we calculate

$$cost_{norm} = \begin{cases} \frac{cost_{GT} - cost_{SEQ}}{cost_{GT}}, & \text{a sequence exists} \\ -1, & \text{no sequence exists} \end{cases}$$

where $cost_{GT}$ is the ground truth cost using straight line distance, and $cost_{SEQ}$ is the best cost from a given algorithm. The closer $cost_{norm}$ is to one, the better our sequence. When $cost_{norm}$ is close to zero, $cost_{SEQ}$ is close to the ground truth cost. In certain edge case scenarios, an algorithm may not produce a valid sequence. In this case, we penalize this lack of output with a negative one. We use these metrics instead of an edit distance or a direct comparison to the ground truth sequence, because we are attempting to encapsulate the space of feasible sequence solutions instead of only arriving at the ground truth sequence.

We create 50 datasets, each using $C(10, 4)$. Shot length is randomly selected between 30 and 120 frames, or one to four seconds. We include the simpler versions of some of our methods, which assume that cameras can only move right before their shot assignment, to confirm that our performance improves with the more complex methods. We discuss these methods in more depth in the appendix. We evaluate the average runtimes of our planning methods in Table 2. In Figure 5, we see that, across various datasets, the overall cost of RRT distance is highest. All dynamic methods have roughly the same shaped curve, indicating that they are all finding similar paths between target positions. All of the dynamic methods also converge as they approach their lowest cost sequences. This convergence makes intuitive sense, as those sequences require less motion, so the differences in the path planning metrics will be less apparent. The greedy methods all tend to either converge to the lowest cost sequence, or not converge at all. When they find a valid output, the cost of the RRT solution is slightly higher than that of the other two, which are practically overlapping.

Table 3 shows that overall our straight line and piece-

Table 3. We calculate the mean and standard deviation of c_{norm} and the number of sequences that cost at most that of the ground truth sequence over 50 datasets. We penalize greedy results with $k > 4$ with a -1. In the greedy case, our piece-wise line method performs best. In the dynamic case, the simpler methods perform worse, as we expect. As exhibited in one of our example datasets, the straight line and piece-wise line methods perform almost identically for both tests. While RRT is better than the simple cases, its performance is worse than the other two planning methods.

Clustering Type	Path Planning	Average Offset from GT Cost (m)	Number of Good Sequences	Unsolvable Datasets
Greedy	Simple Line	0.8209 ± 0.4066		17
	Simple Trajectory	0.6837 ± 0.5223		16
	Line	0.1159 ± 0.4383	N/A	**9**
	Piecewise Line	$\mathbf{0.0978 \pm 0.4122}$		**9**
	RRT	0.1507 ± 0.4399		**9**
Dynamic	Simple Line	-0.1835 ± 0.2125	4.1429 ± 4.2594	43
	Simple Trajectory	$\mathbf{-0.2214 \pm 0.1471}$	3.6364 ± 3.6952	39
	Line	-0.2186 ± 0.1480	$\mathbf{23.0513 \pm 27.7574}$	11
	Piecewise Line	-0.2124 ± 0.1497	21.9500 ± 27.1094	**10**
	RRT	-0.1652 ± 0.1661	13.5000 ± 17.7334	12

wise line methods have the best performance, with average offsets being slightly higher than the ground truth sequence cost. This trend toward negative values may be influenced by our edge case outlined in Section 3.2.3. In this case, we throw away sequences where clustering assignments may cause existing cameras to appear in an active field of view. This issue occurs because all of the cameras on the "set" are not moving in the same way or at the same times as they are in the ground truth.

It is possible that performance may be roughly the same for both piece-wise lines and straight lines because our graph creation is not exhaustive enough at sampling intervals of $\frac{\pi}{16}$, but we believe that it is more likely that the underlying movement for four cameras in this size of "set" is well parameterized by the straight line metric. It is also possible that our generated datasets were not complex enough to frequently require more winding paths given our number of cameras, scene size, and sensor width. Performance for the dynamic results is also influenced by the number of cameras, k. Since our datasets are randomly generated, it is possible that some results only have three cameras instead of four, which will reduce performance results.

5. Conclusion

We have presented methods for randomly generating realistic 2D synthetic data of live edited video, and compared various clustering processes and path planning metrics. Upon evaluation, we found that straight line distance with dynamic clustering produces the largest number of feasible results with shorter overall accumulated camera travel distance than the ground truth assignment. These surprising results indicate that the underlying motion of the data can be sufficiently parameterized by simpler path planning than RRT, piece-wise lines, or simple trajectory optimization. Moving forward, we will apply our dynamic straight line distance clustering to a synthetic 3D dataset, where we will then be able to re-shoot the action of our head models using our resulting output sequence of camera assignments.

References

[1] B. Ala-Eddine, F. Brahim, and M. Kurulay. Efficient camera clustering method based on overlapping fovs for wmsns. *International Journal of Informatics and Applied Mathematics*, 1(1):11–27.

[2] D. Birant and A. Kut. St-dbscan: An algorithm for clustering spatial–temporal data. *Data & Knowledge Engineering*, 60(1):208–221, 2007.

[3] M. Ester, H.-P. Kriegel, J. Sander, X. Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996.

[4] R. Hartley and A. Zisserman. *Multiple view geometry in computer vision*. Cambridge university press, 2003.

[5] M. M. Kheirkhah and M. Khansari. Clustering wireless camera sensor networks based on overlapped region detection. In *7'th International Symposium on Telecommunications (IST'2014)*, pages 712–719. IEEE, 2014.

[6] S. M. LaValle. Rapidly-exploring random trees: A new tool for path planning. 1998.

[7] E. Masehian and G. Habibi. Motion planning and control of mobile robot using linear matrix inequalities (lmis). In *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4277–4282. IEEE, 2007.

[8] S. Pasha. Analysis and enhancement of spatial sound scenes recorded using ad-hoc microphone arrays. 2017.

[9] D. Xu and Y. Tian. A comprehensive survey of clustering algorithms. *Annals of Data Science*, 2(2):165–193, 2015.

[10] L. Yang, J. Qi, D. Song, J. Xiao, J. Han, and Y. Xia. Survey of robot 3d path planning algorithms. *Journal of Control Science and Engineering*, 2016:5, 2016.

[11] G. Yuan, P. Sun, J. Zhao, D. Li, and C. Wang. A review of moving object trajectory clustering algorithms. *Artificial Intelligence Review*, 47(1):123–144, 2017.

[12] O. R. Zaïane and C.-H. Lee. Clustering spatial data when facing physical constraints. In *2002 IEEE International Conference on Data Mining, 2002. Proceedings.*, pages 737–740. IEEE, 2002.

A. Simple Line and Trajectory Methods

For the simple methods we assume that a camera cannot move until right before its turn in the sequence. We use either straight line distance or trajectory optimization as our ϕ. In the straight line case, a path is infeasible if a straight line from p_x to p_i intersects the constraint. Similarly, in the trajectory optimization case, a path is infeasible if one can't be made that doesn't intersect a constraint, but this path can curve.

More specifically, when using trajectory optimization, our constraints on the path ϕ creates are as follows:

1. the points cannot appear in the field of view of the current camera, $y_i - (m_l(x_i - f_x)) - f_y \leq 0 \vee y_i - (m_r(x_i - f_x)) - f_y \leq 0$,
2. the path is smooth, $\sqrt{v_{xi}^2 + v_{yi}^2} \leq v_{max}$ and $\sqrt{a_{xi}^2 + a_{yi}^2} \leq a_{max}$,
3. and the path of points q_i itself is physically realistic, or $q_{i+1} - (q_i + \frac{\dot{q}_i + \dot{q}_{i+1}}{2} dt) = 0$.

For the third constraint, when we move q_i forward using the appropriate fraction of time, velocities, and accelerations, the position of this projected point must be equal to the position of the next point in the path sequence. We handle edge cases in ϕ by clamping large slopes at ± 100.

Note that we cannot use trajectory optimization for our current datasets, because there appears to be no principled and exhaustive way of calculating intermediary target points where the constraints in the environment are constantly changing.