Duan Wang, Jintian Zhu, Daniel Webb

Computer Vision - CS-GY-6643

Final Project

Estimating Basketball Shot Trajectory using Optical Flow and Depth Calculations

**Summary**

As one of the most popular sports across the World, basketball is known by all and loved by many. Our team wanted to try and figure out a way that we could estimate a basketball shot and its trajectory through the three-dimensional space. In this way, we can then try and conclude where the ball might land. Futhermore, we might be able to predict if the ball would, in fact, go into the hoop, thus making the shot.

For this approach we want to try and make use of some computer vision techniques that can approximate the ball traveling through the air and the depth of that ball in the world space. Optical flow naturally would give us the function to track pixel movement across a set of images to estimate the rate of change over time concerning pixel locations and their brightness values. With this estimation we can create a two-dimensional curve projection through space to approximate the ball’s path. To get the depth information we used our two parallel camera setup to calculate the disparity and depth in the three-dimensional space. Combining these approaches can attempt to give us a way of estimating the ball trajectory in three-dimensional space.

The suggested approach above is what we strived for but as will be explained, this was far from an easy task and there were many obstacles and challenges that we faced. Our data set produced quite a lot of noise for us and complicated the scenario. In the forthcoming text and descriptions we will present what we achieved, what we did to try and overcome these challenges, and what approaches could have been used if time permitted.

**Experiment Setup**

For this project we decided that we would have a dual camera parallel setup for the easiest way to calculate the depth of the ball through space using the disparity. Having the cameras parallel allows us to ignore any correcting or more complex geometry. We needed to position these cameras upon a set off boxes to prop it up high enough to capture the entire scene. The picture below shows the setup of the camera and orientation that we used for the project. We found that because the ball could travel upwards of about 13 to 14 feet during the arch of the shot through the air, that to capture the scene in it’s entirety we needed to position these cameras at a distance of about forty feet.

*dual camera setup positioning dual camera setup showing entire scene*

The cameras used in this experiment were of the exact same manufacturing so that their frame rate and image frame size would be the same. We took two videos simultaneously using 1080p resolution with 30 frames per second, planning to sync up the videos in absolute time so that we could extract the images from the set of videos and do our calculations. We used a stopwatch presented on the scene to manually cut the videos to sync them in time.

We took many shots in the scene from all different angles to present a thorough set of data to test our algorithms. We presented some going from right to left across the scene. We also had some where the ball was coming at the camera to get a more complete estimation. Below is a screenshot of just one of these shots. This shot is the most used one by us to demonstrate all the algorithms that we came up with to try and estimate the trajectory of the ball through the scene.



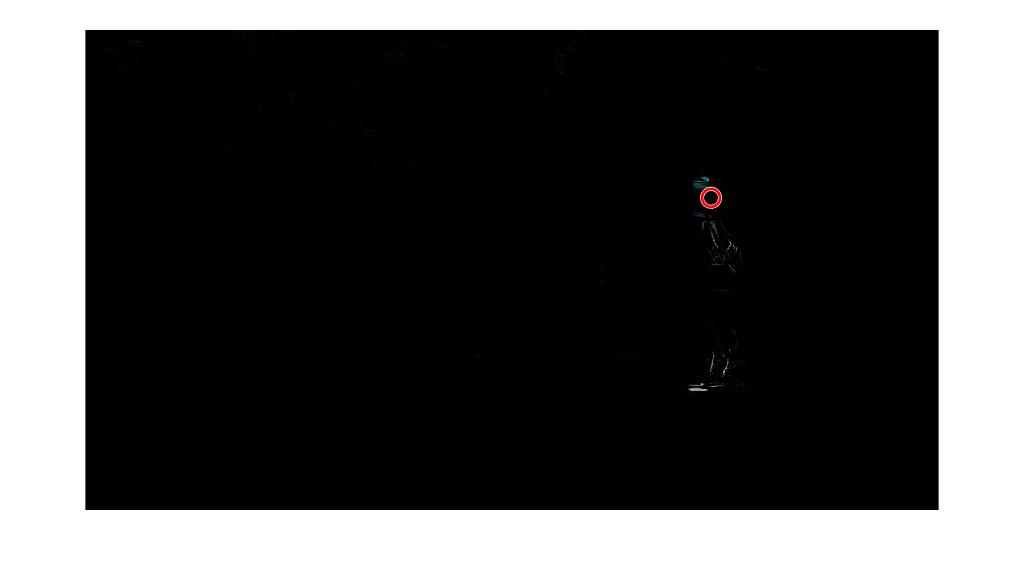
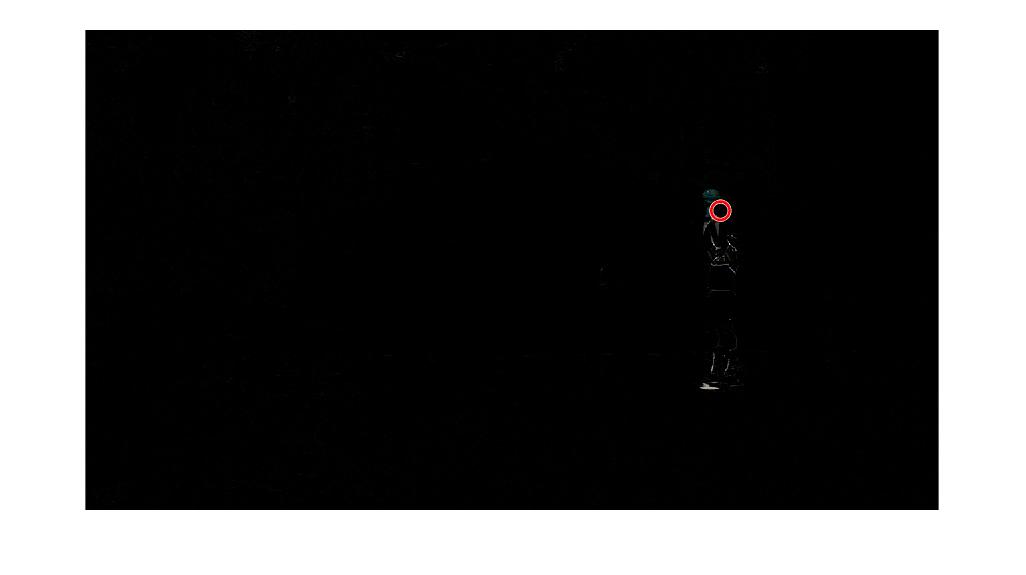
*Shot from right to left shot from back to front*

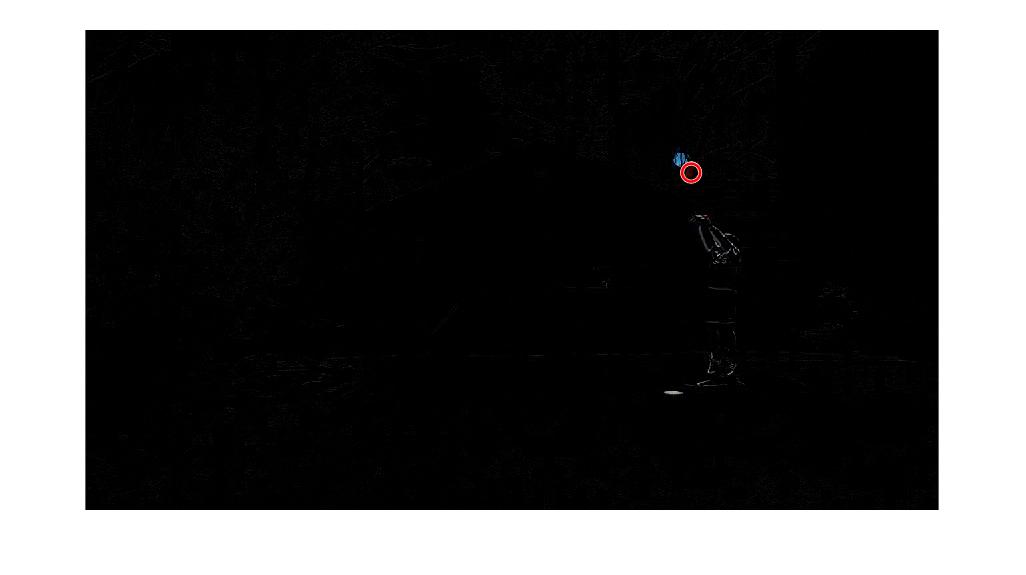
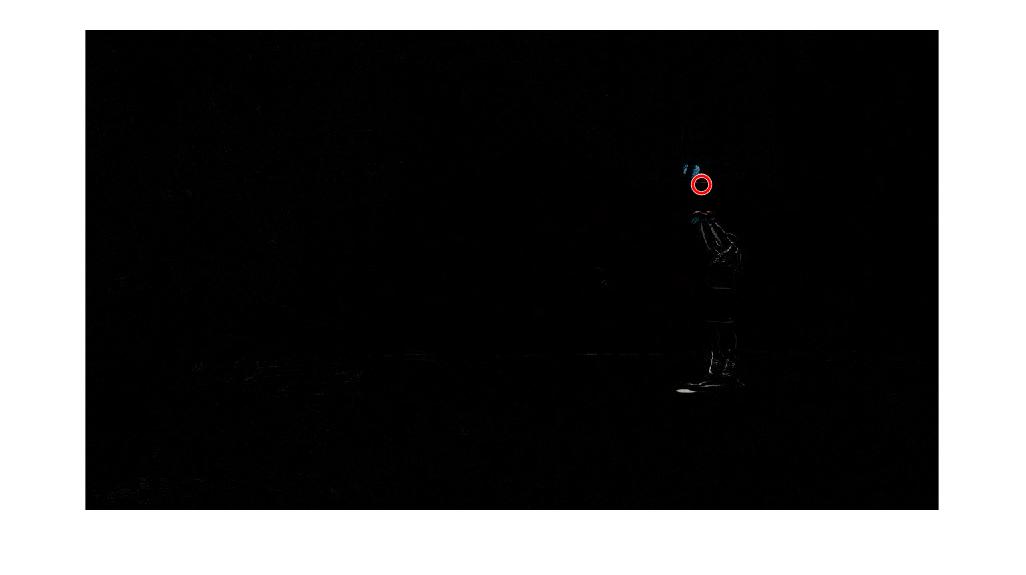
**Video Pre-Processing**

The first thing we should do is splitting our video by frames. Since we have two videos coming from different viewpoints, we should synchronize them frame by frame. The way we did this is simple: using a timer counter in one second and placing it within the frame in the video. Than we can cut the video according to the timer display. However, since our camera has a video rate 29 frame per second, it’s still hard to guarantee a perfect synchronization manually. Thus we need a synchronization algorithm which we will discuss below. Also, we need ball center position on image coordination to get a more accurate results than by manual. The position of ball center will be used in our algorithm to calculate the depth. Because the depth calculation is sensitive to any kind of noise, we need a high level of accuracy which means we can’t manually point the ball center each frame.

*Circle detection*

We use Hough Transform function in matlab to detect the circle. But the challenge we faced is the high speed of ball and the complex background. The speed of ball causes a move blur on the frame which need to loosen the constraint of circle detection algorithm and causes huge amount of circles being detected. We try to remove the background by doing I(t+1)-I(t), but this doesn’t help, because the player body is still moving and can produce several circles. The final method we used to select the circle of ball is calculating the circle’s pixel mean value and compare with some value based on basketball color. This worked because our basketball has an almost unique color above the whole background. We also get the radius of the ball here which we used in depth detect algorithm too.





Circle detect result frame(t) to frame(t+3)

*Fix synchronization problem*

As for synchronization problem, we compare the result of video split, found that the time disparity between two camera is within 1/58 second where 58=29\*2. We use left frame to fix right frames by considering I(t+n). Because the ball movement track will be some parabola, we can predict a simple estimate the Bezier curve by three node along the time. And since our geometry guarantees the parallel of camera and world x axis, in the same moment, ball in left frame and right frame should have the same v value. That’s our method to estimate and fix ball position and get a more accurate disparity. However, the movement blur of ball cause some noise on the ball’s position, this noise also pass to our result of disparity. The world coordinate position calculated base on this can’t be used. Thus we use the ball’s radius to calculate world position which has less noise according to experiment result. And using radius only needs one camera video which reduce the cost of our algorithm. The method we used to calculate world position by radius will be discussed in **Calculate Depth** part.

**Calculating Optical Flow**

*General Approach*

In general, optical flow algorithms find the temporal and spatial derivatives between images to try and approximate object movements through the scene captured. These differences between time and space find how the pixels intensities move over time using the brightness constancy that optical flow relies on. These methods produce the velocity vectors which represent the rate of which these intensities move through space and the magnitude of these velocity vectors for each picture throughout the image.

The brightness constancy constraint is deployed here to help explain why the differences give us information in the real world. More formally, the brightness constancy constraint can be defined by, *I(x + Δx, y + Δy, t + Δt) = I(x,y,z) + dI/dx \* Δx + dI/dy \*Δy + dI/dt \* Δt.* This initial equation can be simplified to the form of, *dI/dx\*Vx + dI/dy\*Vy + dI/dt = 0.* As can be seen in the simplified equation before that what is at work here is the difference in pixels in the x-axis, combined with the difference in pixel intensity in the y axis, together with the difference in intensity of the image pixel intensities over time produces the constraint that an images intensities cannot wildly change from frame to frame, otherwise it would not be possible to track them in any consistent way over time.

*Implementation*

Our initial approach to estimating the optical flow was from our dual video camera setup, to take that video and extract the frames necessary to do our calculation. We took our raw video film which was approximately five minutes long and used video editing software to cut up the portions that we wanted to use. Each one of the clips that we used to test our algorithms were of about three to four seconds, just enough for the ball to be pushed up into the air and then the end of the clip is when the ball hits the ground.

Once we have this clip of the shot taking place, we create our functions in Matlab to pick how many images we want to analyze, the space between the image frames that we pick, and also other parameters depending on the algorithm such as neighborhood size, error filtering, etc.. The respective algorithm runs and smoothes the images using the gaussian filter. Then it produces a set of velocity vectors and magnitudes for each of the pixels analyzed in the image. From there we can display the relevant and significant vectors throughout the image. In our case, these would be the body movement of the shooter, the ball traveling through the air, and also the shadow that is produced from the ball given that the sun is nearly directly overhead in our images so the shadow produced is within the captured image frame. We of course, only concern ourselves with the velocity vectors in the area of the ball movement. Therefore, we take into account the pixel region produced from the image processing finding the circle in the images and concern ourselves with that region to try and get the velocity of the ball moving through space.

After obtaining these velocity vectors within the area of our concern we planned on using these, from the first couple frames, to get the velocity of the ball through the scene, and from that how far the ball would travel and with what arc. As will be explained in subsequent sections, the challenges to actually get velocity vectors that we could deal with and reliably use in our situation was quite problematic. There seemed to be quite a significant amount of noise in our dataset which may have produced bad results for our algorithms. This is looked into more in the next section.

*Working with a noisy data set*

We call our data set very noisy based on the preliminary results of using the stock optical flow algorithms available within the Matlab libraries. A few test runs of the optical flow algorithms showed that not much movement was being captured and the movement that we did concern ourselves with was very unpredictable. On nearly every approach we took we had lots of data that looked like the below picture.



We speculated for much time on what could be causing all of this noise and bad results in our algorithms. To be brief because this will be covered more later, we think that the combination of a background whose colors are close in intensity to the ball in some circumstances depending on the shading of the ball. The picture above shows that the sun is nearly directly above the ball because the ball is nearly white on top, this effect caused shading on the rest of the surface of the ball. The true color of the ball is a dark orange color, but this shading effect caused the color to be anywhere between a darker brown color, to a bright white.

One additional explanation for the noise may have been the image resolution. These videos were shot in 1080p resolution from the set of Iphone 6s phones that we had, but after taking the clips, the videos were transformed into 720p with 30 frames per second. Again, looking at the picture above, this graininess around the ball is quite apparent, and even after us implementing the Gaussian smoothing, the calculations were not much better. It may just have been that the ball was too deep into the scene and the graininess of the resolution gave many problems. These explanations that we hope can provide insight into our challenges can be overcome in one way or another using some other image processing tools.

It is quite obvious in this visual example that there are some pretty good looking velocity vectors overlaying the center of the ball in the above image. At the same time though, there is a whole host of noise below the ball where velocity vectors appear to be quite random and sporadic. If we had tried to use this calculation at least, there is no possible way we could have received any projection curve that would make any kind of sense. After realizing this problem it became quite apparent to us that we might not be able to project the actual arc of the ball using the optical flow methods we had chosen. It became clearer and clearer that we might need additional enhancements and image processing techniques to not only create a better image for the algorithm, but also to help make sense and take somewhat of an average over the velocities to limit the noise and get something that can be used in a function so that we could eventually figure out the curve. At this point we pivoted our portion concerning optical flow towards what algorithm and what set of parameters gave us the best results. We would use a visual analysis to see which algorithms best handled the noise and if we had more time, which algorithm we would move forward with to use for our optical flow calculations.

*Lucas-Kanade algorithm*

The Lucas-Kanade approach to optical flow is a hallmark in the field as it introduces an interesting and powerful concept. That is if one pixel moves from point x1 to point x2 across a set of images, than its neighbors probably also moved. Because no object in an image is likely to be only one pixel large, taking the neighborhood of the pixels shows how a group can move together. Of course, in our example, the ball is quite large moving through the air so it would be wise to track the neighborhood.

Using the standard matlab implementation of the Lucas-Kanade algorithm we could manipulate the amount of noise that is let through quite easily. Increasing the noise threshold provided less velocity vectors as many of them would be characterized as noise, while decreasing the noise threshold would designate less as noise. With a lower noise threshold there is a lot more activity on the image but this is not necessarily what we want. Below we provide a set of pictures all using different noise thresholds to try and determine which noise threshold gives us the best results.

For this test, we kept all other parameters equal. We used the same video, started at the same frame point, and used the same amount of frames, with the same amount of space between frames. See the pictures below for the variation of the noise threshold and the results it produced.



*noise: 0.005 noise: 0.010*



*noise: 0.015 noise: 0.020*

Visually analyzing these results reveals the obvious, that the more you increased the noise threshold number, the less velocity vectors you get because more of them are considered noise, and only those with the most prominent features are left behind. When trying to decide which of these is the best for our situation, none of the choices are really that great for us. For each there are many that are pointing in the complete opposite direction than what the ball is actually traveling (right to left). But, even with these bad choices it seems that in the noise range of 0.015-0.020 is the sweet spot where the volume of velocity vectors that are filtered out provides a clearer picture, but with many vectors moving in the opposite direction it is unclear how these calculations could be used reliably for our algorithm if we wanted to calculate the velocity of the ball through space in time.

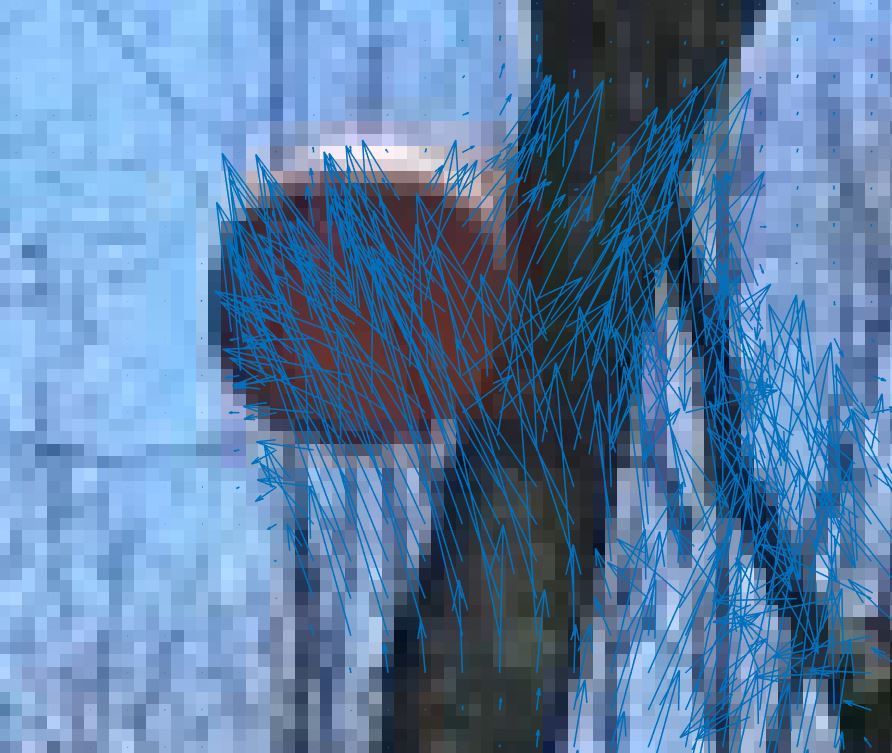
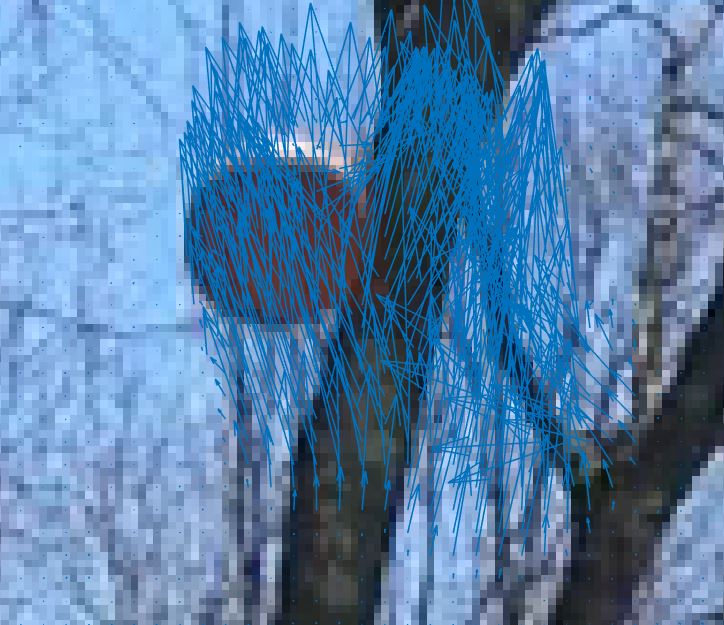
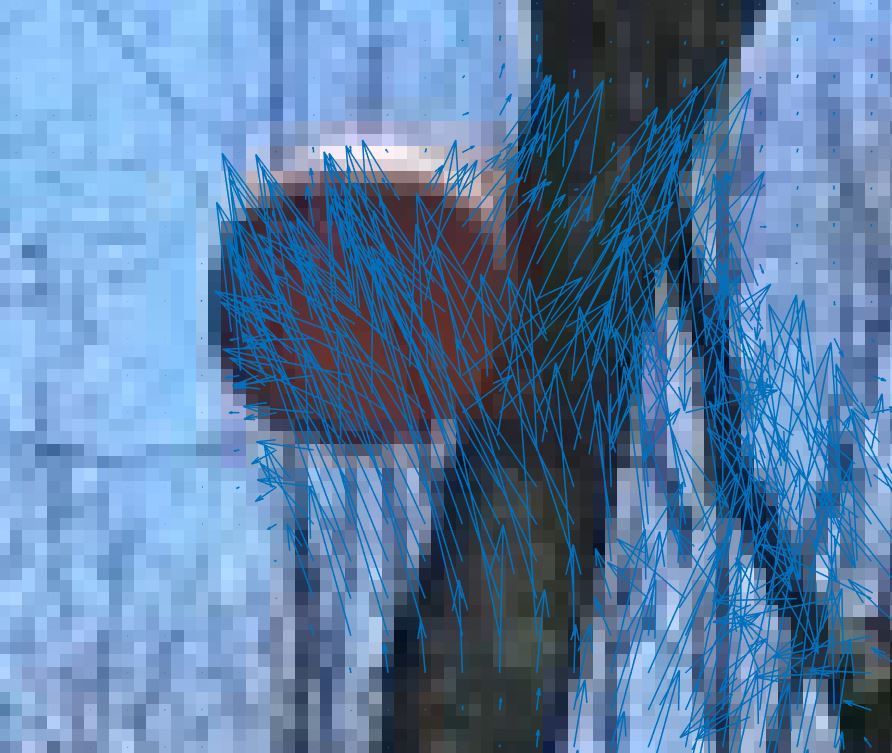
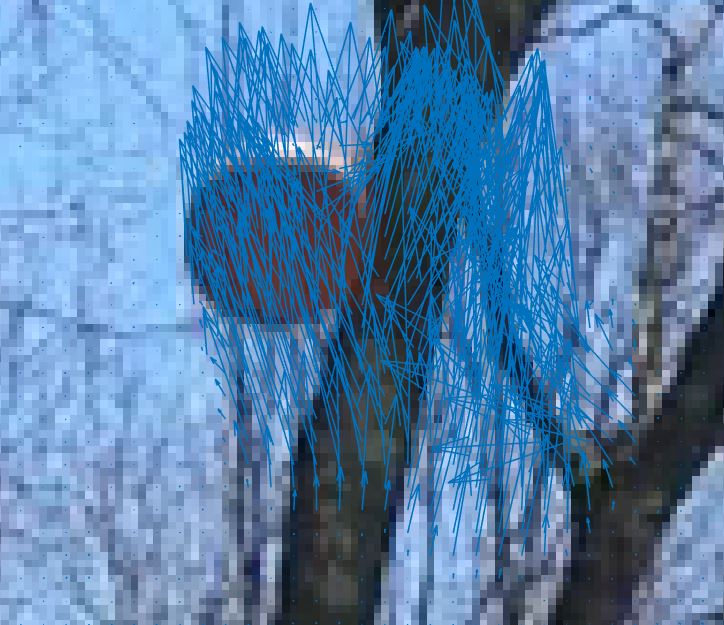
In these images many of the vectors are moving down, and that may indicate the brightness problem that we are having because of the sun in the image. The sun is helping to violate the brightness constancy constraint that is at the very core of the optical flow algorithms. Other factors are certainly at work here to provide wholly undesirable and in many ways, inaccurate results.

*Gunnar-Farneback algorithm*

The Gunnar-Farneback algorithm is a dense optical flow calculation. It uses the neighborhoods between two frames to approximate quadratic polynomials. Furthermore, these quadratic polynomials are used to create the functions needed to produce velocity vectors to track the movement of an object. This implementation allows for the optical flow calculation between just two image frames.

The matlab implementation of this algorithm has a multitude of parameters that can be manipulated to refine our results. The ones that we use include ‘FilterSize’ and ‘NeighborhoodSize’. The NeighborhoodSize parameter is one that we are familiar with, it examines a series of pixels in some neighborhood together when it calculates its optical flow. The ‘FilterSize’ parameter goes through afterwards and applies a Gaussian filter of that size over the neighborhoods to average their velocities and direction.

Below are a number of images with what parameters that were used during that run of the algorithm.



*Neighborhood: 10, Filter 15 neighborhood 10, filter 20 neighborhood 15, filter 20 neighborhood 15, filter 15*

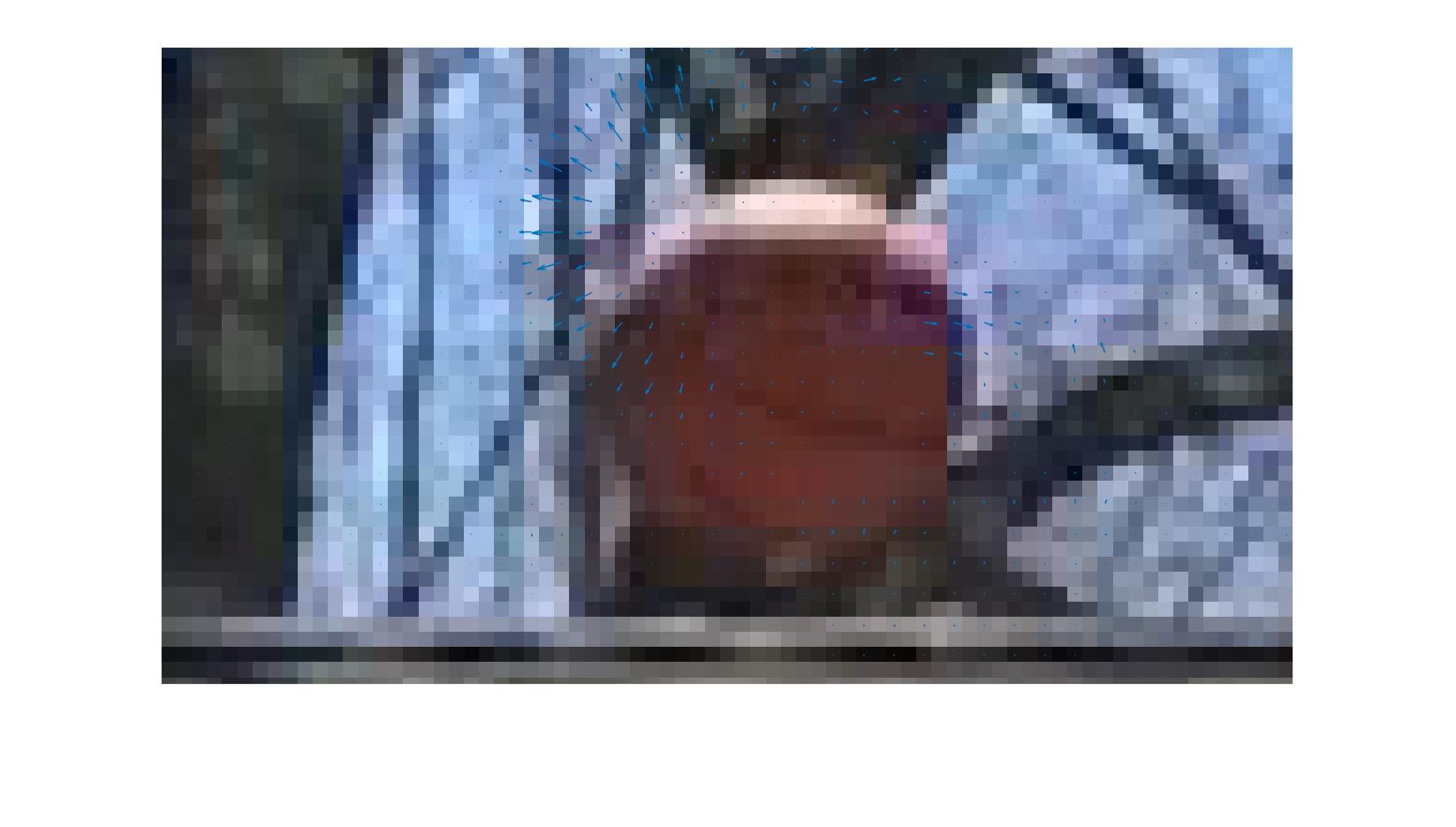
All of the above pictures do show quite attractive velocity vectors for our application. The ball is moving from the bottom right to the top left of this perspective, so the best vector orientation would be pointing in the northwest direction. Each of the parameter permutations achieve this in one way or another, but it looks like that which has a neighborhood size of 10 and a filter size of 15 and a neighborhood size of 10 produces the best results in this scenario. This picture (leftmost one) has a bit of noise towards the right side of the tree branch there. There is also quite a heavy bit of vectors running through the ball pointing it towards the top left corner slightly.

These results look better still from the Lucas-Kanade method that didn’t produce anything useful for us really, but there can be argued there is almost too much activity here. The algorithm seems to do a decent job of averaging the velocity vectors within the neighborhoods to give a more uniform direction to the surrounding pixels. The parameters that produced the best results must have been because of our data set, or the ball size, or a combination of both working together to create a somewhat usable result. It’s obvious that we would still need to use some more kind of smoothing technique to these vectors to get just one vector that indicated the velocity of the motion through space, that then we could take to solve a function outputting the ball trajectory curve through space.

*Horn and Schunck*

We can quickly focus our attention on the Horn-Schunck algorithm which expects a degree of smoothness to the image in our to operate and solve the aperture problem. We have already implemented Gaussian smoothing in the preliminary processing of the images before the optical flow algorithms calculate their velocity vectors, but it’s clear that Horn-Schunck is more sensitive to these constraints than other algorithms may be.

We only cover this method briefly because of how poor the results are that we received from the functions. As you can see below, the velocity vectors are very faint if at all apparently showing up for our dataset. We presume that the smoothness qualities did not match what was expected by the algorithm and to make this work we would need to take significantly more care towards the pre-processing of these images before running them through the algorithm.



As foreshadowed above, the activity displayed in the above images is quite minimal and not as nice or obvious as the previous two methods. This is certainly not something that we could use reliably with our process to project the basketball's trajectory through space. One of the other algorithms would most certainly be more appropriate the the Horn and Schunck algorithm.

*Kalman Filter*

We had some extra time toward the end of the reporting period, so we thought that we could try and just experiment with the Kalman filter to see how it reacts to our dataset and how it could have been employed by us to get better results and maybe be able to complete our algorithm.

The Kalman filter works, by training itself across the data set and then estimating where a ball might move in the future based on what it has seen before. In this way, it is very good at predicting constant motion in one direction. It is not as good in estimating motion that may move in jagged or unpredictable directions. Below show some of the results that we received after running a very basic algorithm to use the Kalman filter in our project.



In the two images above it is apparent that in this limited test the Kalman filter did not seem to do a great job at tracking our object through space. We can blame this on the discoloration caused by the sun and shadows on the ball. Most notably, in the left image, the top of the ball is a nice bright yellow and the bottom is nearly black. The algorithm actually had an easier time trying to track the shadow produced by the sun that traced the driveway, than the actual ball moving through space.

To remedy some of these problems, we could have used some more filtering of our images or only had the kalman filter project between regions that we isolated where the ball is. Restricting this range may have given some better results as it would be less processing intensive, and also the ball could be more easily isolated and the shadow would not be an easier thing for the filter to pick up.

*Further work*

We have explored many regions of study within optical flow and the associated image processing techniques at work here, and still did not come up with nearly enough good results to produce a working algorithm for the ball projection through space. There are many other approaches we could have taken if time permitted, but we decided to do this comparative analysis to display which method we could use for our further work towards completing this project. Some points of further investigation and explained below.

If more attention was paid to the pre-processing of the images it certainly would have benefited us greatly. One way of improving would have been to use some kind of background subtraction techniques to eliminate the noise we saw from the trees and other background objects that may or may not have been near the same color as the ball we were using. Also, only doing our calculations over the region of the ball that was in the images would have optimized the performance of our algorithms greatly.

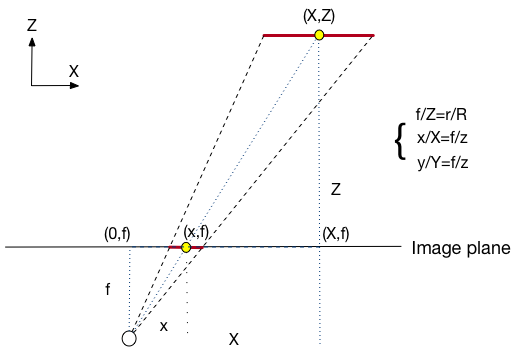
Additionally, combining the kalman filter with the optical flow algorithm could have introduced some interesting results. We could have replaced the ball with some solid color or images so that the Kalman filter could track the ball more easily and predict its motion ahead of time. From there we could have also calculated the actual speed of the ball and angle through space to try and create the function needed to project the ball trajectory in the three dimensional space.

Generally speaking, some image processing techniques that were not readily available to us could have really helped here especially when we were to try and average the vector magnitudes and direction. With more time and resources, we are confident we could have come much closer to our initial goals.

**Calculating Depth**

Because of the problems we countered during the calculation of disparity, there is much error in the derived array of center depths, and this result can hardly be utilized. Thus we need find another method to calculate depths.

By having a deeper thought of geometry and relationship between the basketball and the camera, we realized that we can actually take advantage of the fact that the shape of a sphere never changes from whatever viewpoint. Thus its circumference remains the same, which indicates its diameter and radius also remain the same. If we already know the diameter of the basketball, and using some circle detection algorithm we can get the position and size of the ball and calculate its length in the image plane, by we could deduct its depth by comparing the actual ball diameter with the image plane diameter. In this method we do not even need double view. The figure below shows the basic reasoning.



Assume we do the camera calibration and assume the world frame is the same as the image frame, so we have the intrinsic parameters. Then we can get the length of basketball diameter in the image plane by calculating positions of two end points, multiply them by K, and then subtract one from the other. In this experiment because after calibration we found intrinsic parameter 𝞪 is very close to 𝞫, meaning horizontal and vertical pixel densities are almost the same, and 𝞱 is nearly 90 degree, we reversed the order of calculating by finding how many pixels a diameter occupies first and then divide it by the mean of 𝞪 and 𝞫. For simplicity we consider a 2D cross section perpendicular to the y-axis first. In the figure we can see two similar triangles, one defined by the image plane diameter ***r*** and the camera center, and the other defined by the real world diameter ***R*** and camera center, thus we have

,

After getting the depth value, we could calculate x-axis coordinate of the center using similar way. And because the situation remains in a cross section perpendicular to the x-axis, we can calculate y-axis coordinate in the same way. Thus we could get the 3D coordinate of a ball center in the camera frame (also the world frame) from the image.

However this method is still far from ideal because there is still much error in finding a circle from a frame, especially in the case of shooting from back to front, because the basketball is far away from the camera, its circumference seems rarely change. The circle detection algorithm is not perfect and what we actually got is not a sequence of diameters getting larger in every frame but randomly rise or fall. To cope with that, we first calculated the 3D coordinate of each center using the method above, then did polynomial curve fitting respectively for three axis coordinates. If we simplify the influence of air resistance and assume the reduction of velocity remains the same for each frame(or other unit of time), then we can fit y-axis coordinates in a quadratic curve and the coordinates of rest two axis linearly. In fact this easy while defective method works well in our experiment circumstance because the whole period of shooting a basketball is done in nearly only a second, thus the horizontal (x and z axis in our camera setup) velocity change is not obvious.

**Trajectory Estimation**

Now we have a list of x,y,z coordinates that are polynomial curve fitted, the estimation part remaining seems merely a physical problem as we know the trajectory of the basketball is a quadratic parabola if irrespective of air resistance. But we want to introduce a more general(natural) method of calculation first.

Assume we input *n* frames for analysing, i.e., we have *n* center points *‘p’*, from a differential point of view, the velocity of ball in a frame, say *ith* frame, can be calculated by simply subtract the center coordinate in the *(i+1)th* frame by its *ith* coordinate:

Decompose positions and velocities to each axis and the formula holds, here we still use horizontal to represent x and z axis:

,

.

Overall we can have *n*-1 velocity values of three dimension available to analyse. Given that the air provides an acceleration to the ball and its value is always directly proportional and opposite to the current speed, so we could model of air resistance from the following recursion formula:

Thus

the horizontal velocity for the *ith* frame is

So to calculate ‘*k*’ we actually need exponential fitting. After obtaining ‘*k*’, we could estimate x and z coordinate afterwards. While the modeling for vertical velocity is a bit different because it is influenced by not only air resistance, but gravity. We assume the gravitational acceleration value is ‘*a’*, then:

Thus

So for the *ith* frame, the vertical velocity is thus

With *n-1* ‘*v*’ values available, we could calculate *n*-2 estimates of *a* and choose the average value. The equation above is a bit complex though, we in fact do not need it as we can iteratively call the recursion formula.

As mentioned above, the captured point centers and ball radiuses are erroneous, and unfortunately using exponential fitting, the result of ‘*k’* value is quite large, which is certainly unnatural. Further, by applying such ‘*k’* value, we get weird ‘*a’* value that is far from the actual gravitational acceleration we know. As a result, we get ridiculous trajectory of estimation. Instead, we assume the the reduction of velocity is constant for every frame regardless of current speed, then we got the following formula:

With *n-1 ‘v’* values, we could have *n-2* sets and *n-2* estimation of horizontal differential coefficient of velocity *dvh*, then we could use the average value as *dvh* for later trajectory prediction. then we calculate the vertical differential coefficient of velocity *dvv* in similar way according to the follow formula:

This method actually provides more reasonable estimation, and we will provide two examples of the estimated trajectory using this algorithm in the following part.

**Draw Predict Trajectory**

We draw our prediction of Trajectory by circles, whose position represent ball’s x and y value in the world coordination radius represent the depth value. We manually choose the frame count as the training set to predict a better trajectory. Using less frames cause an inaccuracy of the result and using more frames cause a waste, predict a ball just above the loop is meaningless. Finally, we draw the trajectory on our raw video and output as video. The trajectory will follow the frame index and keep on predict ball’s movement by frame.

However, we didn’t consider the collision of ball and backboard or loop or the ground. It can be done by re-construct these object world information and add into consideration of predict trajectory in a future work.





**Result/Challenge**

*Predict Accuracy*

Our model needs a large amount of frames to produce an accurate enough curve. The biggest problem we face on are the noise. Since our camera setup quite far away from the center line between ball and basket, the main object: ball is small base on the frame size we got from video. Thus, several pixel size mistake will cause a big different in the result of projection matrix calculation and cause even bigger noise on world coordination.



As we can see: the ball’s radius on image only around 12 pixels, and the high light caused by sunlight on the top and the blur on the boundary is the major noise effect our result.

This problem also add the cost of our project since the ball is small, detect it by circle detect function needs loosen constraint and an addition filter.

*Algorithm Cost*

The total cost of our project can’t achieve a real-time calculation as we expected at the beginning. That means we still have room to improvement our algorithm, especially the image processing part. The circle detect method from matlab toolkit we used run in a large cost because of the blur noise of our frames.

**Conclusion**

A lot can be learned from a noisy dataset and this project demonstrated that. On the fly, there were many image processing techniques and tweaks that were needed to be implemented to get this project going from the start. Starting with recognizing the circles in the image was not an easy task as all of the noise prevented very clear edges to the shapes in the scene. The main theme of our project revolves around the noise that we encountered. It affected every single one of our calculations and needed to be accounted for at every turn.

The optical flow calculations suffered quite a bit from this as was explained in great detail above. Looking at four separate solutions to this, came up with maybe just one or two that could possibly give us something to work with. Had we known more image processing techniques, we could have helped our algorithm quite a bit, but in its own way it was interesting to see how and what parameters influenced the calculations. Although this was not the goal of our project, struggling to get realistic result gave a crash course in understanding these algorithms more in-depth.

The most successful portion of our project were the depth calculations. Again here, much noise and some setup issues needed to be handled to get this part working as well. The synchronization problem between the dual video camera setup handicapped our calculations from the beginning. To deal with this, the circle detection algorithm was employed to calculate the diameter of the ball, and across the set of images this depth could be inferred from the similar triangles between the world frame and the camera frame. From there, we were able to calculate the depth in space of the ball and project this curve ahead of the ball.

If there were more time, we could have dealt with all this noise better and our algorithms could have been improved because instead of trying to deal with this noise inside of our algorithms, the noise could have been filtered out ahead of time. Despite not accomplishing our hopes with this project, we found a great amount of value here, and the challenges we faced allowed us in a way to learn more about the subject and the problem we were trying to solve. We feel confident that with more image processing knowledge our application could have been completed.

**Credits**

Experiment Setup/Filming - Dan

Circle Detection Algorithm - Duan

Calculating Depth - Jintian / Duan

Trajectory Estimation - Jintian

Optical Flow - Dan

**Github**

https://github.com/dwebb28/CVFinalBasketballDropPoint