Investigating The Relationship Between Shot Distance & Shot Mechanics In Basketball

submitted by

Lavy Friedman

for the degree of Master of Science

of the

University of Bath

Department of Computer Sciences

September 2019

❑ **Retain for Sample** **Mark**

UNIVERSITY OF BATH **Department of Computer Science** **INDIVIDUAL COURSEWORK** Submission Cover Sheet **Please fill in both columns in BLOCK CAPITALS and post into the appropriate Coursework Submission Box.**	*for office use* **Date and time received** **This section will be retained by the Department Office as confirmation of hand-in**
How to present your work 1. Bind all pages of your assignment (including this submission sheet) so that all pages can be read by the marker without having to loosen or undo the binding. Ensure that the binding you use is secure. Missing pages cannot be marked. 2. If you are required to submit part of the work on a disk, place the disk in a sealed envelope and bind the envelope into the submission. You must keep a copy of your assignment and disk. The original is retained by the Department for scrutiny by External Examiners.	**Declaration** *I certify that I have read and understood the entry in the Department of Computer Science Student Handbook on Cheating and Plagiarism and that all material in this assignment is my own work, except where I have indicated with appropriate references. I agree that, in line with Regulation 15.3(e), if requested I will submit an electronic copy of this work for submission to a Plagiarism Detection Service for quality assurance purposes.*
CURRENT YEAR EXAM CANDIDATE NUMBER(this is a five digit number) **07549**	FAMILY NAME AND EXAM CANDIDATE NUMBER **Friedman** **07549**
	GIVEN NAME **Lavy**
UNIT CODE **CM50170**	UNIT CODE **CM50170**
UNIT TITLE **Research Project**	UNIT TITLE **Research Project**
DEADLINE TIME & DATE **September 13th, 2019 17:00**	DEADLINE TIME & DATE **September 13th, 2019 17:00**
COURSEWORK PART (if applicable) **100%**	COURSEWORK PART (if applicable) **100%**
	SIGNATURE [illegible]

Abstract

This dissertation addresses the challenge of 3D reconstruction of a parabolic motion trajectory through a single 2D video stream, in the context of a basketball shot. The system in this paper builds on the methodology of previous tracking systems, while adding elements of analytical geometry for the purpose of tracking the ball throughout its movement along with multiple key points on the basketball court, and define a homography mapping in order to create the transformation from the 2D image plane onto the 3D court model, while tackling issues caused by camera movement, occlusion and the ball's rapid movement. The system ultimately is able to extract the shot's trajectory along with defining parameters of the shot (i.e: release angle and velocity), determining its outcome. The system was originally designed to run on a large number of basketball shot videos to answer the question of how does distance from the basket influence shot mechanics. Due to limitations, the project's scope has been reduced to one, proof of concept video, although it is still possible to build on the framework suggested here to process more videos and achieve this objective.

Acknowledgements

I'd like to thank Mr. Wenbin Li for always being available for me and giving me the guidance and direction needed to complete this project. I'd also like to thank all my professors her at University of Bath for teaching me the art of data science and I'd like to thank my family and friends for their support and encouragement throughout this year and during this project.

Contents

List of Figures

1 Research Proposal

1.1 Introduction

Basketball is a highly dynamic spatio-temporal game, that poses many challenges for state-of-the-art machine learning. Given the richness of the game dynamics, it is a very interesting domain to evaluate new learning methods to learn about shooting mechanics, techniques and variables affecting shot success probability. In the game of basketball, there are 2 teams competing against each other and the team with the highest points total at the end of the game (48 minutes in the NBA, 40 minutes anywhere else) wins the match. The point system is explained in Figure 1:

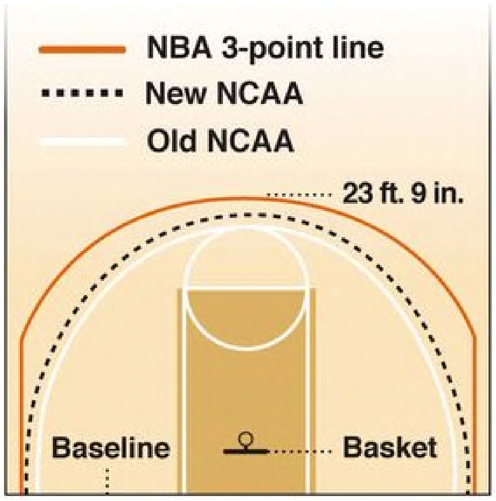


Figure 1: A shot (also referred to as a field-goal attempt) outside the 3-point line that goes in the basket is worth 3 points, otherwise the shot is worth 2 points.

The 3-point line was introduced in the 1979-80 season. Even though a shot from beyond the line is worth 50% more than a regular shot, looking at Figure 2 using data from *basketball-reference.com* (2018), we can see that teams at first weren't eager to attempt a lot of 3-point shots.

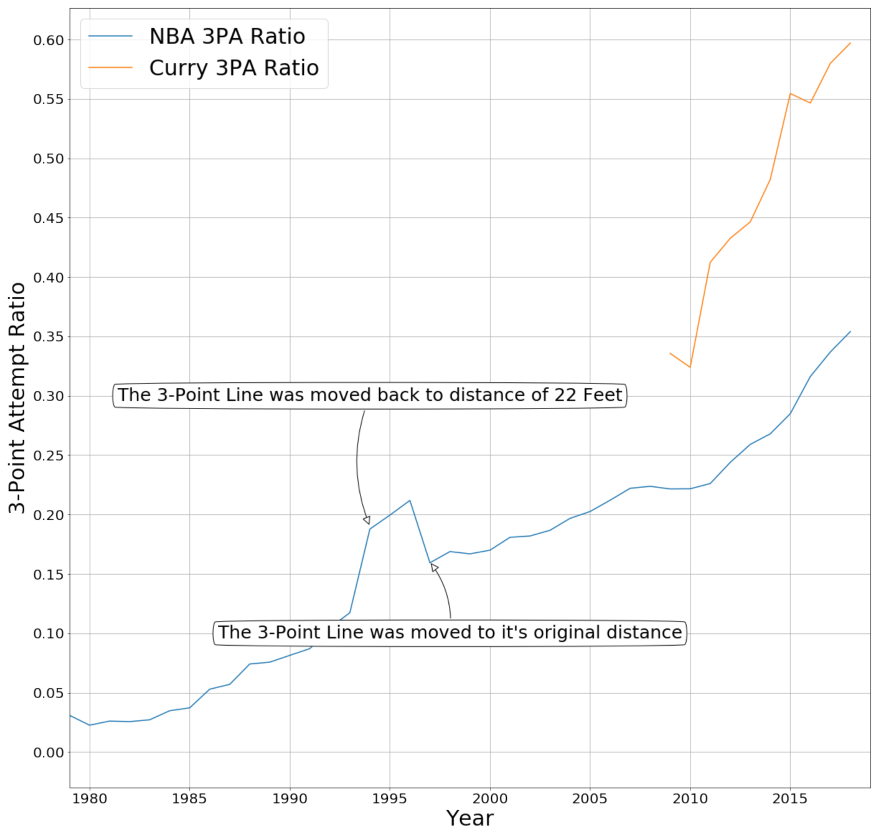


Figure 2: The ratio of 3-Point Attempts over time.
The ratio is calculated as: $\frac{3-Point\ \ Attempts}{3Point\ \ Attempts+2Point\ \ Attempts}$

The increase in 3-point attempts throughout the years seemed to follow a steady linear trend until 1994 when the 3-point line was moved closer to the basket which caused a huge spike. 3 years later, once the 3-point line was moved back to its original position there was a drop off which made the number of 3-point attempts revert back to its previous linear trend. This linear trend continued until it plateaued in 2008 at around 22% . After 3 years it started to rise again at a much faster pace than before. This rise is attributed to the emergence of Stephen Curry, who is arguably the greatest shooter in NBA history. *[Scott Davis (2018),Ric Bucher (2015),Kirk Goldsberry (2019)]*

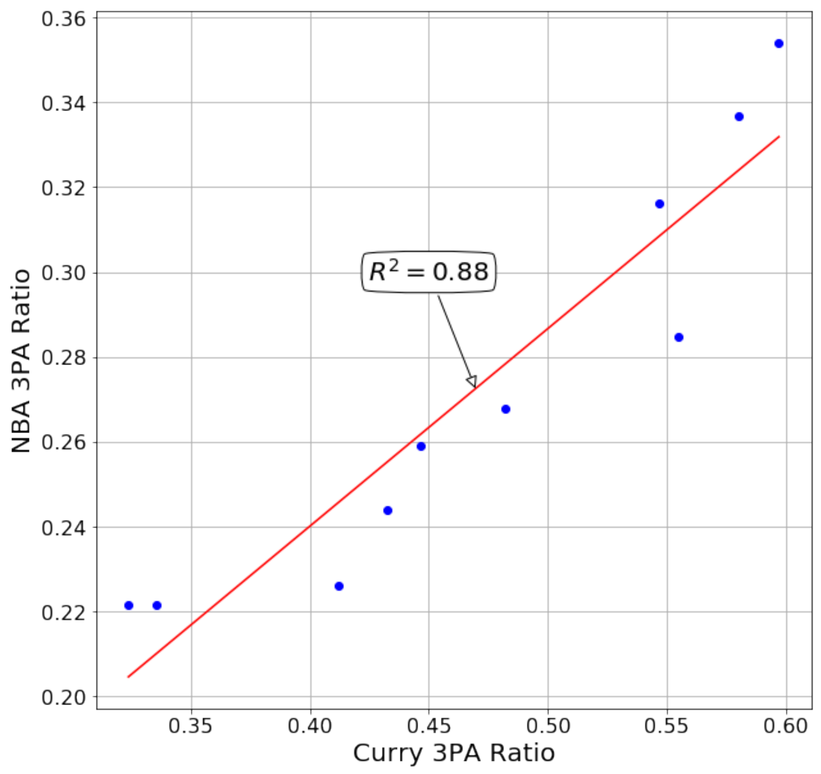


Figure 3: Curry's 3-Point attempt ratio VS. the rest of the NBA's (2009-2019). In this figure we can see how Curry's rise in 3-point attempts correlates heavily with the trend of the rest of the league.

By showing that volume 3-point shooting can lead to winning (Curry's team has won 3 of the last 4 NBA championships), Curry helped popularise increased 3-point attempts as a winning strategy (evidenced by the rising trend in figure 2.)

Naturally, with this recent rise in 3-point attempts the number of "deep" 3 point shots (from 5+ feet behind the 3-point line) increased as well. As we can see in figure 4 the rising rate is astronomic and the number of these kind of shots is 5 times greater than when Curry entered the league and has doubled itself just in the last year.

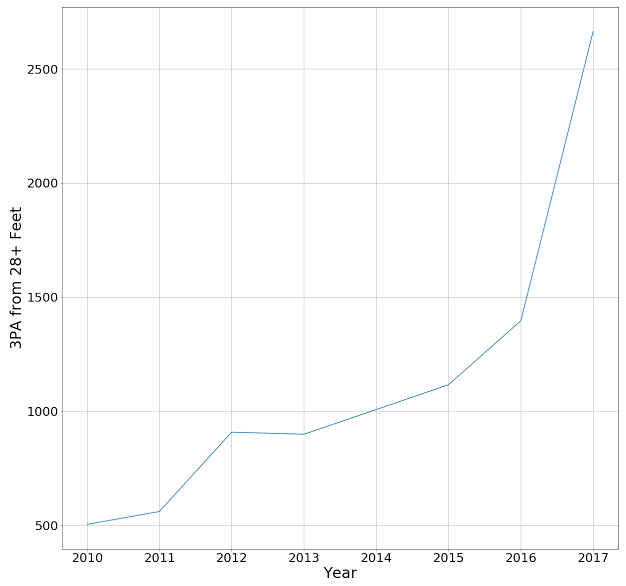


Figure 4: Shots from 28+ feet away plotted vs. Time. Data from NBA-savant database. (2019*b*)

However, the problem we're facing is that since the described meteoric rise is fairly recent, conventional shooting training don't address these type of extra-far shots, they only account for a general 3-Pointer. An example for this we can see in Fontanella (2006): "When shooting a 2-foot shot, you only need a launch speed of approximately 10 miles per hour. **For a 3-point shot** you need a launch speed of approximately 18 miles per hour."
In this project I intend to use and add on Machine Learning techniques to investigate the manner in which shooting mechanics change with distance, namely shot angle and velocity.

1.2 Project Plan

In order to investigate this matter I will use video streams of NBA games, and look at 3-point shots from varying distances. Previous work has been done in terms of looking at a sports match stream and mapping the elements on the court. *[Liu et al. (2006),Farin et al. (2005),Fu et al. (2011)].*
In order to do this the videos have to be segmented into frames to capture the entire trajectory of the ball from the moment it leaves the shooter's hand until

the ball hits the rim or the net.
Once that is done, we can handle the challenge of estimating a homography from "real world" coordinates into image coordinates since in every frame we can potentially deal with a different camera angle. To do that we need to do the following (for each frame):

1. Determine which colors are most dominant in the image, to help us differentiate the court from the rest of the image.

2. Once we have created a "mask" of the court, we can then use Hough transform to detect the lines on the court which we have knowledge on their "real" length and position on the court. (see figure 5)

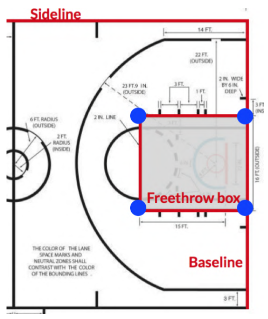


Figure 5: The points marked in blue would be good candidates for the Hough transform to detect since they are a intersection of lines.

3. Now we have several points on the court which we know their real location on the court as well as their image coordinates. Therefore, we can map out the homography of the court.

The next part would be to detect the ball's 3D coordinates throughout its movement. Since the ball is so small compared to the rest of the picture and is moving at a high speed the image could be blurry in some frames. In addition to that, the background around the ball is always changing and will be noisy because of the crowd (figure 6). All these factors will pose a challenge for the tracking algorithm.

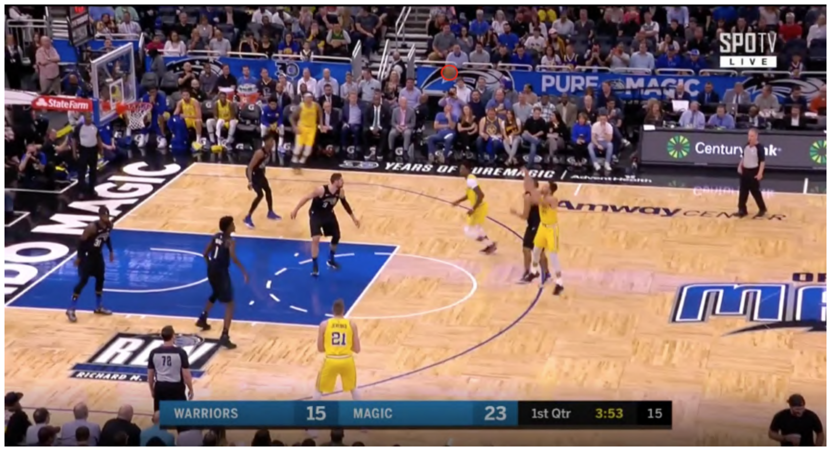


Figure 6: A 3-Point attempt taken in a game. Ball is marked by red circle

In order to overcome these challenges we will use Kalman's Filter as well as Viterbi's decoding algorithm, which have produced promising results in the 3D tracking of a football. *[Liu et al. (2006)]* Since the camera angle is much closer in basketball, I predict that we will get good results here too.

The general initial timeline for this project is described in the following Gantt chart:

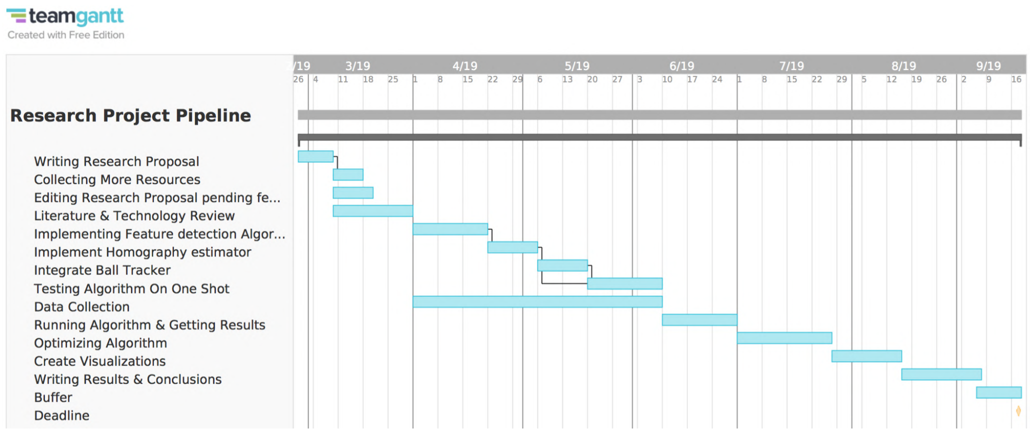


Figure 7: The project pipeline laid out from the research proposal stage until submission deadline

1.3 Aims, Objectives & Deliverables

1.3.1 Aims

The overall goals of this project are:

1. Given a video/sequence of frames of a 3-Point shot. Be able to determine distance from the basket and optimal shooting parameters (namely angle and velocity, but other parameters could be added as well as the project moves forward).

2. Idea: Given a partial video of a shot, be able to estimate probability that the shot was successful.

1.3.2 Objectives

The necessary steps to achieve the goals of this project are:

1. Collect and index the data, convert into sequence of images.

2. Implement feature detection algorithm.

3. Estimate homography of court.

4. Implement ball detection method with good accuracy.

5. Obtain ability to extract ball trajectory throughout frames of a video.

6. Find optimal ball trajectory per distance.

1.3.3 Deliverables

The core outputs for this project that fulfil its aims would be:

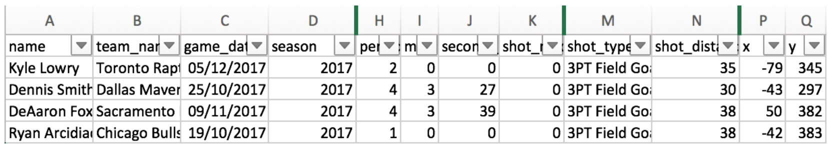
1. A visualization of the optimal shot parameters as a function of distance from the basket.

2. Shot predictor application: Given partial shot video, predict if shot was successful.

1.4 Data Collection & Resources

1.4.1 Data Collection

Data collection will have to be done manually. In order to find a sufficient number of shots I would have to watch many hours of basketball which would be quite inefficient.
Using data from NBASavant, I can index the shots by distance, get the exact moment in the game when the shot was taken and then download the video using NBA.com's interactive box-score. (2019*a*)



A	B	C	D	H	I	J	K	M	N	P	Q
name	team_nar	game_dat	season	pe	m	secon	shot_	shot_type	shot_dist	x	y
Kyle Lowry	Toronto Rapt	05/12/2017	2017	2	0	0	0	3PT Field Go	35	-79	345
Dennis Smith	Dallas Maver	25/10/2017	2017	4	3	27	0	3PT Field Go	30	-43	297
DeAaron Fox	Sacramento	09/11/2017	2017	4	3	39	0	3PT Field Go	38	50	382
Ryan Arcidia	Chicago Bulls	19/10/2017	2017	1	0	0	0	3PT Field Go	38	-42	383

Figure 8: NBASavant's query results

Moreover, since I can also obtain X,Y coordinates of the shooter, in order to get better accuracy on ball tracking I will only select to look at shots that are straight line from the basket to get a more consistent distance from the camera to the ball. In another effort to get a more consistent movement pattern, I can filter out the shots where a player is running while shooting or jumping backwards.

1.4.2 Resources

For the ball's detection I intend to use and build on OpenCV's ball detector using the algorithms mentions in the above Project Plan Section. (2015*b*)
Since I will be dealing with computationally heavy image processing programs, I will need to use some type of GPU, via cloud computing or the university's high performance computing cluster, Balena.

2 Literature Review

2.1 Introduction

Amongst the main objectives of this paper will is to obtain the ability to extract the ball's trajectory throughout the frames of the video in the most accurate and efficient way. Tracking the ball would be a very challenging mission because of multiple factors:

- ***Size-*** The ball is very small compared to the rest of the image, especially when we're looking at 3-point shots where the camera has to zoom-out significantly to capture both the ball and the basket area in the same frame.
- ***Movement-*** After the ball is released it will move at a speed of approximately 20 MPH, which could result in a somewhat blurry representation of the ball in the image and will make the task of tracking more challenging.
- ***Lighting-*** The lighting in the arena combined with the change in location of the ball will cause the color of the ball to change throughout its movement.
- ***Occlusion-*** The ball could be partially hidden by a player's hand or body during the shot, which could cause our ball detector to not recognize the ball's shape, and result in a poor approximation of the ball's location.
- ***Cluttered Background-*** Due to the camera angle, there will be frames during the shot where the ball appears to be in the same region as the crowd, this could make the tracker misidentify the heads in the crowd as the ball.

Since detecting the ball's location is such a key part of this project, the literature review will give an overview of the previous work done in object detection and segmentation. We will start from the original template-based trackers (Wong & Hall (1978)) and move forward to the feature-based trackers and finally deep learning based trackers. Finally, some context will be given to the topic of this research the work relating to the game of basketball in terms of tracking will be reviewed.

2.2 Template matching & Template-Based Object Tracking

A template is a sub-image within our image, which will be the object we want to track. Templates matching is a technique often used for image processing. More recently, it has been used for object tracking. As explained in (Wong & Hall (1978), Chantara et al. (2015)) generally it works as following: A template is taken from a representative image. The objective is to match that template with another area in a different image. To do that we locate the corresponding areas in the 2 images. If there are multiple areas that qualify, intensity and geometry transformations will be made until a match is made.
For the sake of this research's implementation our template would be a picture of a basketball within a picture of a game. Then we would compare other "basketball candidates" images' similarity to the reference basketball, based on a statistic (e.g Normalized Cross Validation, Sum of Squared Distance) and based on that statistic we would classify the candidates as a basketball or another object/background.

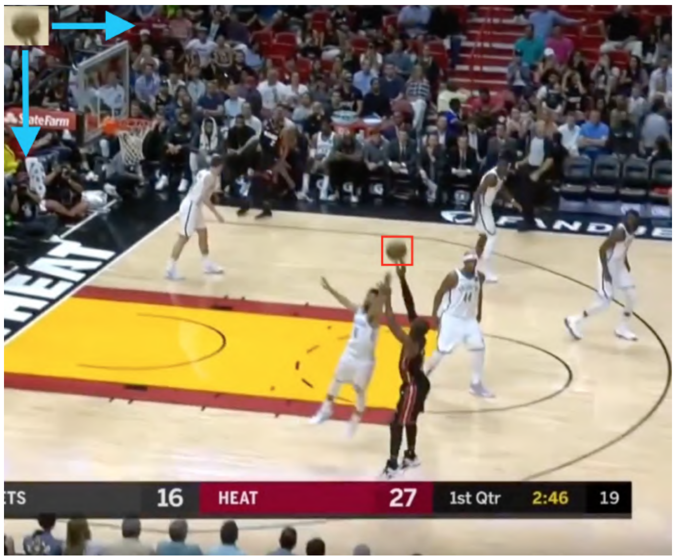


Figure 9: An example of how a very basic template-based ball tracker would work. The source image is overlaid with the template (top left). A box the size of the template will then slide across the image and calculate the difference from every box that size to the template. The box with the minimal difference (or max correlation) will be chosen as the object we wish to track.

Looking at the way template matching is described in Wong & Hall (1978) it seems to be prone to error for this project, as it doesn't address occlusions which could be a significant hurdle for the task at hand. Moreover, for our purposes, we can model the physics of the expected flight trajectory of the ball to limit our search region and filter out as "ball" candidates large regions of the image, thus saving significant computation. Seems like a less naive implementation needs to be incorporated in order to make the tracking more accurate which brings us to the next section.

2.3 Mixed Template-Based & Feature-Based Trackers

Ladikos et al. (2007) tried to integrate a more feature-based approach in their suggested tracking system to handle more dynamic scenarios, where there are lighting changes, occlusions and the object we wish to track is moving. Their system is described in the following figure:

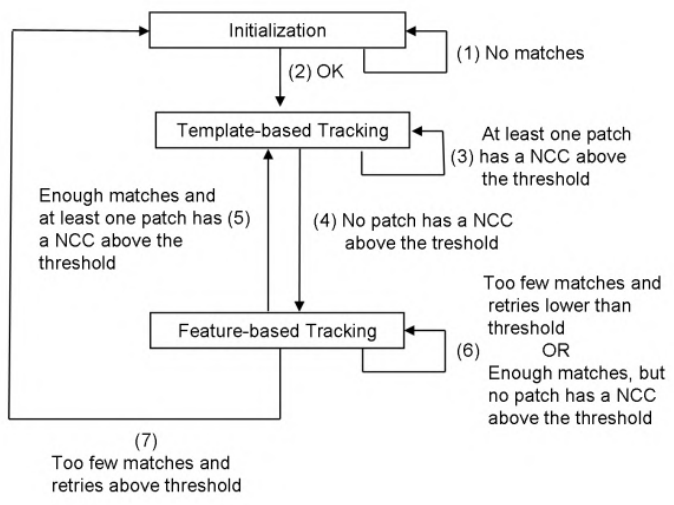

Initialization
(1) No matches
(2) OK
Template-based Tracking
At least one patch (3) has a NCC above the threshold
Enough matches and at least one patch has (5) a NCC above the threshold
(4) No patch has a NCC above the treshold
Too few matches and retries lower than threshold OR Enough matches, but no patch has a NCC above the threshold
Feature-based Tracking
(6)
(7) Too few matches and retries above threshold

Figure 10: As we can see in the figure, the system defaults to template-based tracking, trying to find patches with sufficient correlation to the template and selecting the maximum amongst those who do. In the case where no patch surpasses the threshold (due to occlusion, rotation,etc.), the system moves to "feature-based" mode. Ladikos et al. (2007)

In Ladikos' case, since they are tracking box-shaped objects Harris' corner detector is used for feature-based tracking. In our case, since a ball has no corners we would have to use a different feature detector, for example the circle Hough transform using one of the methods described in Yuen et al. (1990). Another method combining both template-based and feature-based tracker is described in Liu et al. (2006). The context of this method is especially relevant to this research as this paper aims to track the football players and ball location from a broadcast image.

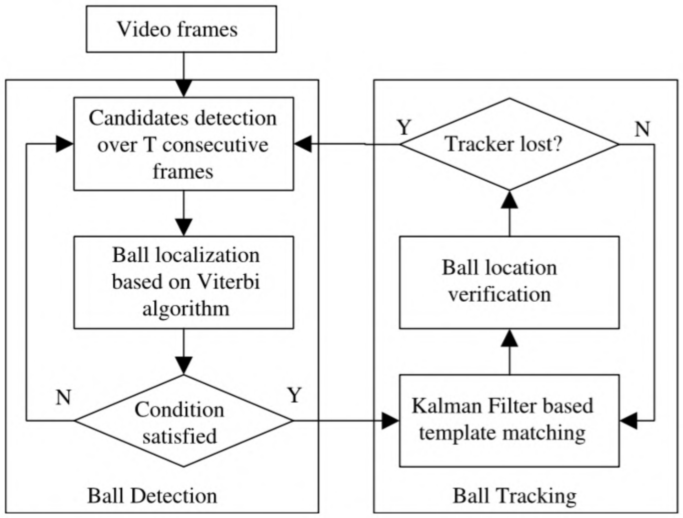

Video frames
Candidates detection over T consecutive frames
Ball localization based on Viterbi algorithm
Condition satisfied
N
Y
Ball Detection
Tracker lost?
Y
N
Ball location verification
Kalman Filter based template matching
Ball Tracking

Figure 11: A flow chart of the ball detection & tracking system described in Liu et al. (2006)

The model of the system has similarities to Ladikos' model in terms of switching between template-based and feature based, however in this case it first uses Viterbi's decoding algorithm which utilizes information over consecutive frames to find the most likely ball path in order to detect the ball. In addition to that, this model incorporates Kalman filters which use information from previous frames to help approximate the ball's location in the next frame. What's truly impressive about this system is that it is able to provide promising results predicting the ball's height without information regarding another known object's (i.e the goal-post) height, in this research's case we do have this information (height of the basket is 10ft), making life a bit easier.

Since these papers were released (2007), deep learning has developed immensely and with this development deep learning based trackers were developed.

2.4 Deep-Learning Based Trackers

2.4.1 Introduction

As described in Feng et al. (2016), with the ground-breaking revolution of deep learning, neural networks have become a powerful tool for object tracking and detection, amongst other tasks (e.g: image classification). As pointed out in Zhang et al. (2017), despite the success of traditional, hand crafted trackers, deep learning based trackers have largely dominated the visual tracking field since their introduction. Deep learning's greatest strength is perhaps its incredible ability to extract important features. Deep networks can get high level features such as colors, edges and curves from an image. These features have high dimensionality and distinction and would take a very long time for a human to manually extract. This point brings us to another one of the advantages of deep learning, it extracts features automatically without a human having to specify to the computer what to do.
There are many different types of trackers based on different neural networks. We will review 3 different ones that are relevant to the problem at hand to better understand what kind of structure should be used for the purpose of this research project.

2.4.2 Stacked Denoising Auto-Encoder (DAE)

The tracker suggested in Wang & Yeung (2013) is based on a DAE. The general steps it goes through are described as follows:

(a) The model takes as input a natural image for training.

(b) Some noise is injected into the image.

(c) encoder extracts the features of the object that are resistant to variations such as illumination changes, rotations, partial occlusions,etc.

(d) Once the model is trained, the information about those features is used for the tracking process.

(e) The object is bounded by a small box in the first frame of the video

(f) Using importance sampling based on that box's location, we draw N particles, which will represent N possible states (locations) which the object we wish to track could be in the next frame.

(g) We pass those particles through the network and to get a confidence measure regarding the accuracy of each particle.

(h) We take the particle with the max confidence (assuming it passes a certain threshold) and use that to mark the box for our object in the next frame.

(i) Repeat stages (f) to (h) till the end of the video

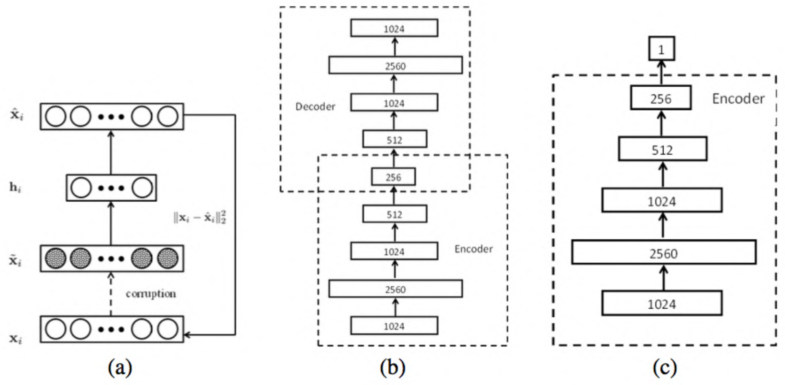


Figure 12: (a) The architecture of the denoising auto encoder: The model takes as input the original image, adds noise to it and outputs a restored version after passing through the hidden layer/s. (b) Number of units in each layer. (c) The network during the tracking process. Wang & Yeung (2013)

In Wang & Yeung (2013) this tracker was compared with 7 other trackers on 10 different video sequences. It achieved the best results in terms of both performance and accuracy.

2.4.3 Convolutional Neural Network (CNN)

Even though CNNs have been successful in regards to the mission of object detecting and classification it took a while for CNNs to break their way into the visual tracking field. This is because, like most neural networks, CNNs require a massive training dataset while tracking apps will generally have just one labelled frame (at the start of the video). The CNN suggested in Wang et al. (2015) called SO-DLT (Structured Output Deep Learning Output) appears to address this issue by splitting the process into 2 stages, similarly to the DAE.

- Pre-training the network offline by using the imagenet dataset to extract the robust features that differentiate the objects from the background, meaning the features that make objects, objects.

- Following training, this learned information is transferred to the online tracking framework.

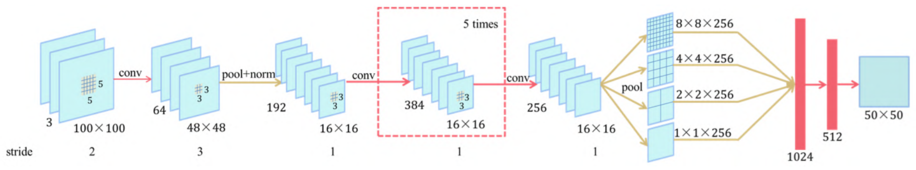


Figure 13: The architecture of the CNN SO-DLT uses for training. As we can see the model's input is a 100x100 image and after going through 7 convolutional layers combined with max pooling, the model outputs a 50x50 probability map where every cell in that 50x50 map refers to a 2x2 region in the original image. More specifically, it gives a measure of how confident we are that the object is in that region.

The online tracker part of the SO-DLT uses 2 CNNs simultaneously that will collaboratively decide on the bounding box location and scale. One CNN (CNN_S) will be updated very frequently and will account for short term appearances to adapt to changes in appearance of the object. However, if we update our model too aggressively, it will be easily influenced by noise. This is where CNN_L comes into play as it will be updated more conservatively. In the end, the decision on where to place the bounding box will be made by the CNN that is more confident. This collaboration makes the tracking more immune to problems like drifting which will be caused by occlusions or cluttered backgrounds (both problems will definitely be encountered in the course of this research).

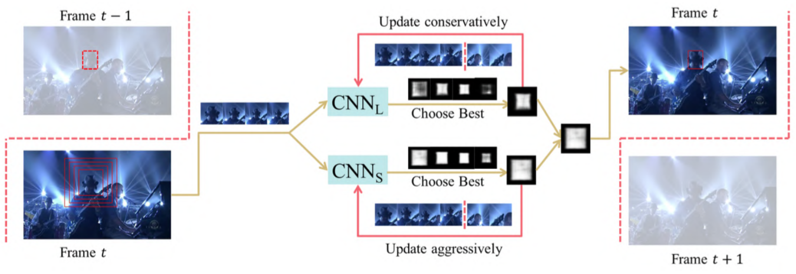


Figure 14: The pipeline of the tracker SO-DLT uses. Given the previous frame's bounded box location and the new frame the 2 CNNs update (or not update) their model and in this case the short term CNN was more confident, thus it will decide on where to mark the bounding box.

In Wang et al. (2015), this tracker was compared with 6 different trackers including the deep learning tracker proposed in Wang & Yeung (2013) and showed significantly more accuracy than all of them. Even though this tracker displayed good results, the paper has had reservations regarding the tracker's success in cases where the background includes distractors that are similar in shape to the object we intend to track. In those cases it may track the distractor rather than the target. This is definitely something that will be tested in this project since the ball's shape is similar to a human's head and the background is full of people in the crowd.

2.4.4 Structure Aware Network (SA Net)- Mixing CNNs with Recurrent Neural Networks (RNN)

RNNs are mostly used for modelling dependencies in sequential data. Hence it is natural for RNNs to be integrated in the task of visual tracking. While CNNs have been fairly successful in this task, as noted preciously, CNN based visual trackers have had issues differentiating the target object from similar distractors. In order to solve this issue, the SA Net model proposed in Fan & Ling (2017) utilizes a RNN to model the object's structure and incorporate this information into a CNN thus improving the model's resiliency to distractors of similar appearance.

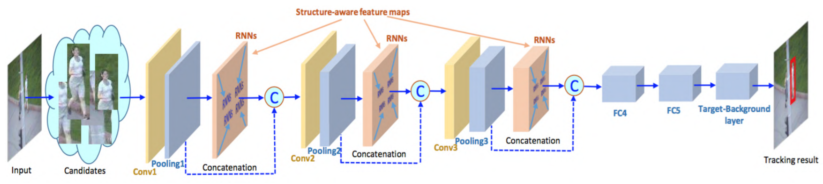


Figure 15: The architecture of SA-Net Fan & Ling (2017)

As we can see in the figure above, the same way each convolutional layer extracts features from different levels, RNNs are used following each of those convolutional layers to model the object's structure from different perspectives.

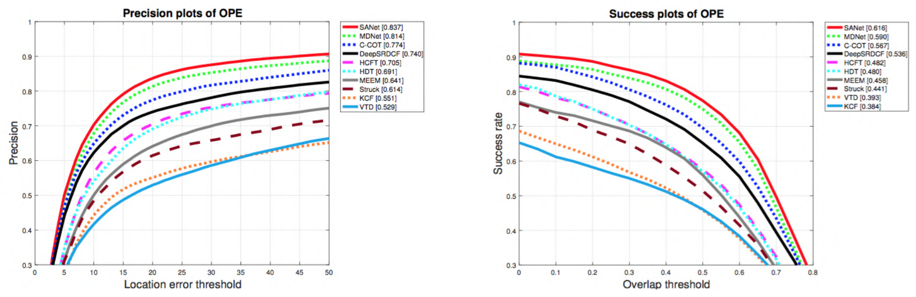


Figure 16: Precision plot(left) and success plot (right) for comparison of different trackers on a benchmark dataset (TC-128). For both, SA-Net obtained the best results Fan & Ling (2017)

2.5 Application of Trackers in Basketball

As mentioned in the proposal section, basketball is an incredibly dynamic sport, full of movement. It is close to impossible for the human eye to be able to keep track of everything that happens on the court, so naturally tracking applications are very useful in this field. One tracking application suggested in Fu et al. (2011) is able to track every player's 2D location on the court to extract information regarding screen strategy (A screen occurs when one of the offensive players blocks out his teammate's defender to free him to make a play or shoot the ball) in order to be able to learn about offense tendencies of opposing teams and strategize accordingly.

Due to its time of publication, this paper doesn't deploy the enormous power of deep learning, however, the techniques used to create a homography between the broadcast image and the basketball court using the court's consistent and known features will definitely be useful for the sake of this research. It is possible that player detection technique can be used in order to extend our algorithm to include closest defender distance.

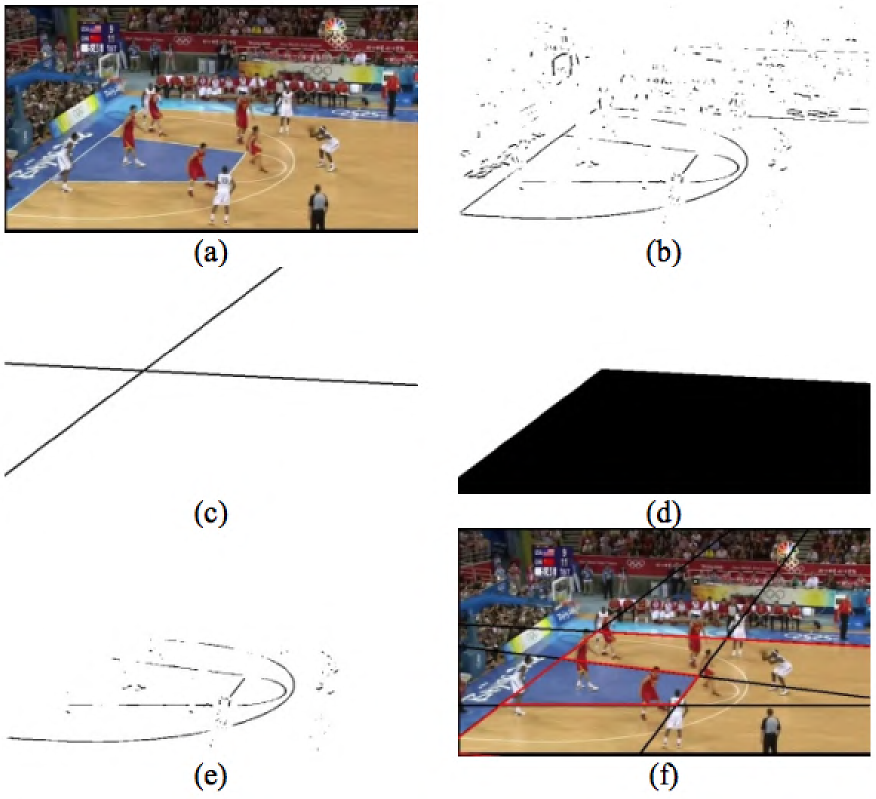


Figure 17: The process of creating a homography in Fu et al. (2011): From the original broadcast frame (a) the white line pixels are detected (b). According to the longest vertical and horizontal lines (c) the court's region is determined (d) so we can filter out white pixels from beyond that region (e) and calibrate according to the lines of constant length and orientation (f)

To the best of my knowledge the only application developed for 3D trajectory reconstruction of the basketball is proposed in Chen et al. (2009). It uses similar methods to Fu et al. (2011) in terms of the 2D mapping of the court. It extends past that to account for the physics of the ball motion in the 3D tracking model. Like Fu et al. (2011) because of this paper's publication date it doesn't use any deep learning methods. The 3D tracker suggested in this paper achieves very solid results although, as admitted by the authors, like every tracker this one also commits false detection when there is a ball-like object in the extension of the ball's trajectory track. After having a closer look into the venues chosen to validate this tracker, it appears that these venues tend to have a very ”clean” background, unlike NBA arenas, as seen below.

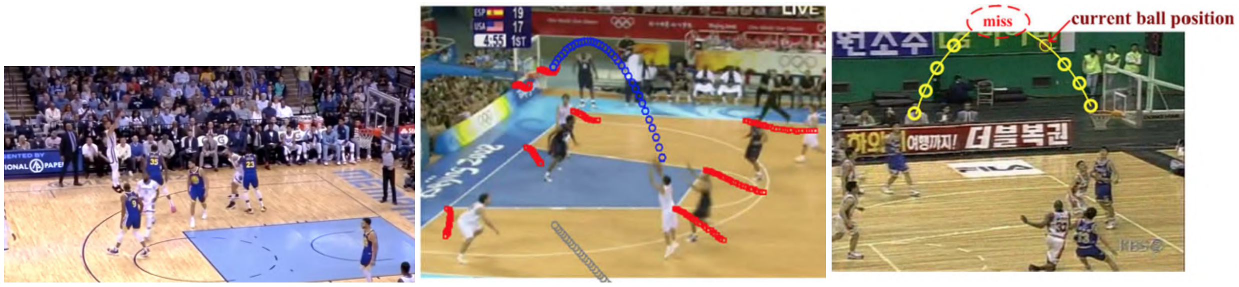


Figure 18: An example of a shot from an NBA game (left) and shot trajectory reconstructions from Chen et al. (2009) (center & right). Notice the difference in terms of the clutter in the background, it is very hard in the NBA game picture to differentiate the ball from the heads in the crowd.

This seemingly minute detail could make a huge difference in the tracking accuracy as many, if not all trackers (including this one) struggle with cluttered backgrounds and similar shaped distractors. It is important to note that for the purpose of this research, only NBA videos will be used for 2 main reasons:

- As demonstrated in the proposal section, the trend of increasing shooting distance from the basket is an NBA phenomena. There isn't data to support that it is a phenomena in other leagues.

- The availability of data and videos from other leagues is very sparse compared to the NBA.

In this project I will try to build on the knowledge modelled in Chen et al. (2009), Fu et al. (2011) and combine it with the great power of deep learning to create the ultimate ball tracker.

2.6 Observations & Conclusions

This literature review shows the evolution of trackers over time. It was eye-opening to see the amount of effort and time spent on hand crafting the features for early-day trackers, as opposed to how nowadays the neural networks do a lot of this work for us, however there is still a lot to consider while modelling these trackers.

Another small observation is that many of the deep-learning trackers are based on bounding the object with a box in the first frame. Since the object we wish to track will be round for the most part, unless its occluded, perhaps ignoring the edges of the box and implicitly using a bounding circle would deliver better results.

A point that stood out regarding the deep learning based trackers is in how many different ways we can account for the aspect of time in our model, whether it's using the particle filter approach (Wang & Yeung (2013)), using 2 different CNNs (Wang et al. (2015)), incorporating a RNN in the process (Fan & Ling (2017)) or using other ways that are not specified in this review. It will be interesting to see what way would work best for the purpose of this research.

A recurring theme in all the trackers reviewed, including the basketball related tracker is their struggle with cluttered backgrounds and similar shape distractors. Since the crowd doesn't move much throughout the course of a basketball shot (1-2 seconds) while the ball moves a lot, perhaps another neural network can be incorporated to 'remember' the crowd's appearance, specifically their heads. This way, once a new circle appears the tracker can be more certain that this circle is the ball and not part of the crowd. This approach will be explored further as this project develops.

3 Methodology

This section features the description of the steps necessary to reconstruct the 3D trajectory of the ball off of a single video stream from start to finish.

3.0.1 Data collection and Pre-Processing

The video used for the proof of concept in this paper is a short video from the NBA's official website (2019*a*). showing a short clip of a 3-point shot from a match between the Memphis Grizzlies and the Los Angeles Clippers from January 26th, 2018. The video was downloaded and snipped so that the first frame will be the frame which the ball leaves the player's hand and the last frame is the frame in which the ball enters the basket.

3.1 Ball Tracking

The first step in reconstructing the 3D-trajectory of the ball is naturally to track the balls location on the screen in each frame. In the literature review many types of tracker frameworks were reviewed and while building a deep-learning based tracker seemed like a good idea at the time, attempting to deploy a couple of "off-the-shelf" object trackers on a basketball shot video resulted in very poor tracking as once another object/person got into the tracker's region of interest it immediately latched on to it and lost track of the basketball.
This happens because those trackers are trained on images of a variety of objects, with different shapes and orientations. In order to use a deep-learning tracker, a large training set of images of a moving basketball is needed. Seeing as that would have been an awfully tedious and long process A decision was made to take a different approach and build a computer-vision based tracker. The tracker was built using the following steps:

1. Mark the ball's location in the first frame of the video, which needs to be the exact frame where the ball leaves the player's hand.

2. A bounding box is taken around this location. That box is the search region (to be explained later).

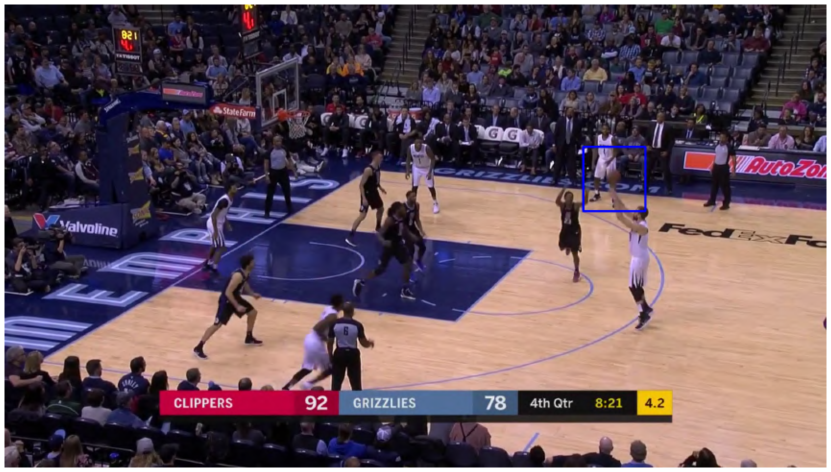


Figure 19: The frame where the ball leaves the player's hand

3. Convert our image from BGR (Blue-Green-Red) to HSV (Hue-Saturation-Value) space. The reason behind this is that using the hue component makes the pixel detection less sensitive to lighting variations. In figure 21 it is evident why that is beneficial for this process.

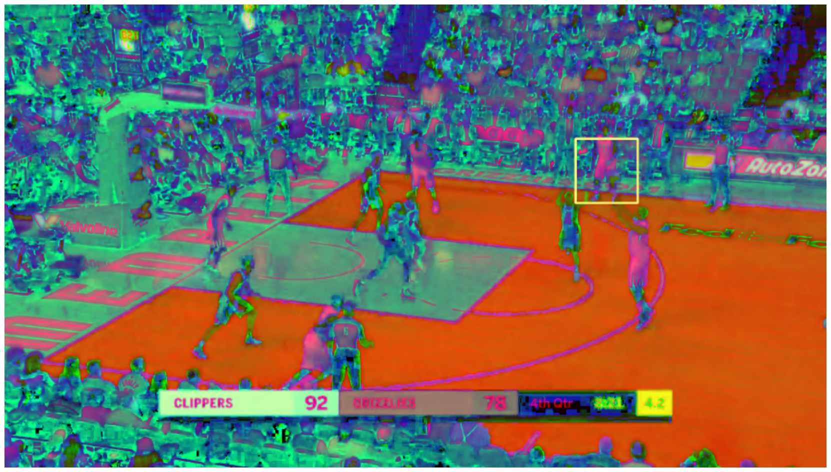


Figure 20: The first frame converted to HSV color space.

4. The next step is to construct a mask that only shows the pixels that are in the color range (in HSV space) of the basketball. In order to gauge what is this color range, cropped out pictures of the basketball in flight were taken,

the color histogram for each channel (hue, saturation and value) is extracted so we can get the lower and upper bound for each channel. Then, by using openCV's *inRange* function, the mask is set up.

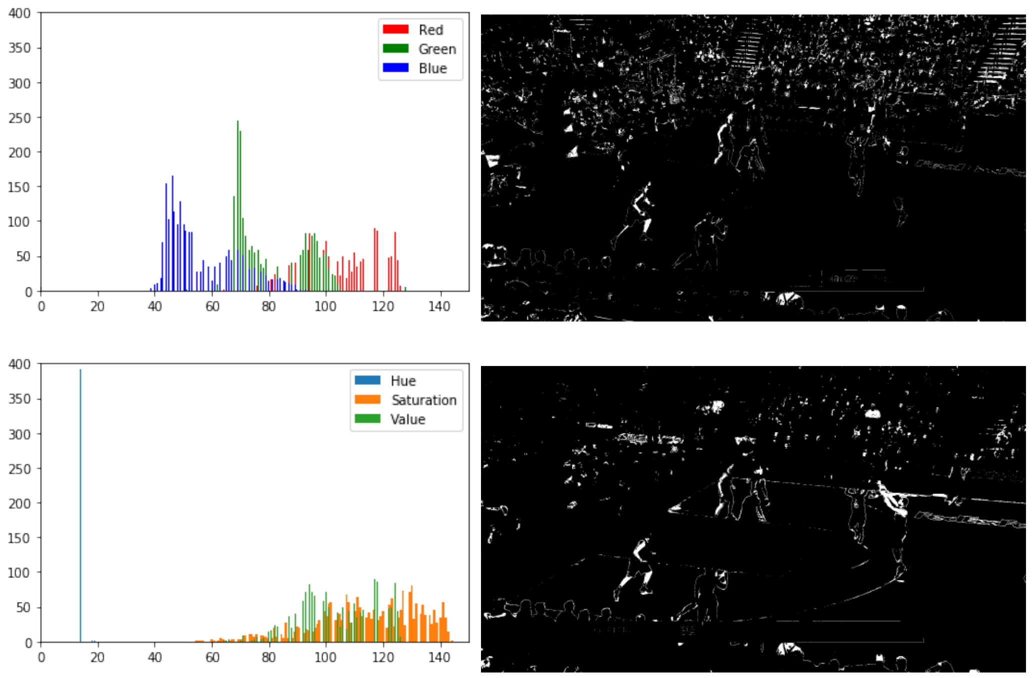


Figure 21: *Top:* The basketball's cropped image's BGR histogram and a frame of the video with the BGR mask applied to it. *Bottom:* The same frame with the HSV mask applied to it. It is evident that the HSV mask does a considerably better job filtering out the pixels that are not part of the ball. This is in large part due to the separation created by the Hue channel as seen in the histogram.

5. In order to eliminate the background, all the frames are read in and then added to a list. Next, the differences between consecutive frames are added to a list and these are the objects taken into account when we track the ball. This fairly simple operation successfully enables elimination of the background, which is important since the object that is being tracked is in constant motion, therefor its immediate background is always changing.

6. Now, a mask is constructed which highlights the pixels that are in the color range of the basketball in frame K that werent in that colour range in frame

K-1. This way if there is a person in the crowd that wears an orange shirt, barring any substantial camera movement, the shirt wouldn't be mistaken for as the ball.

7. Morphological operations (erosions and dilations) are preformed to eliminate small noises and bumps in the mask.

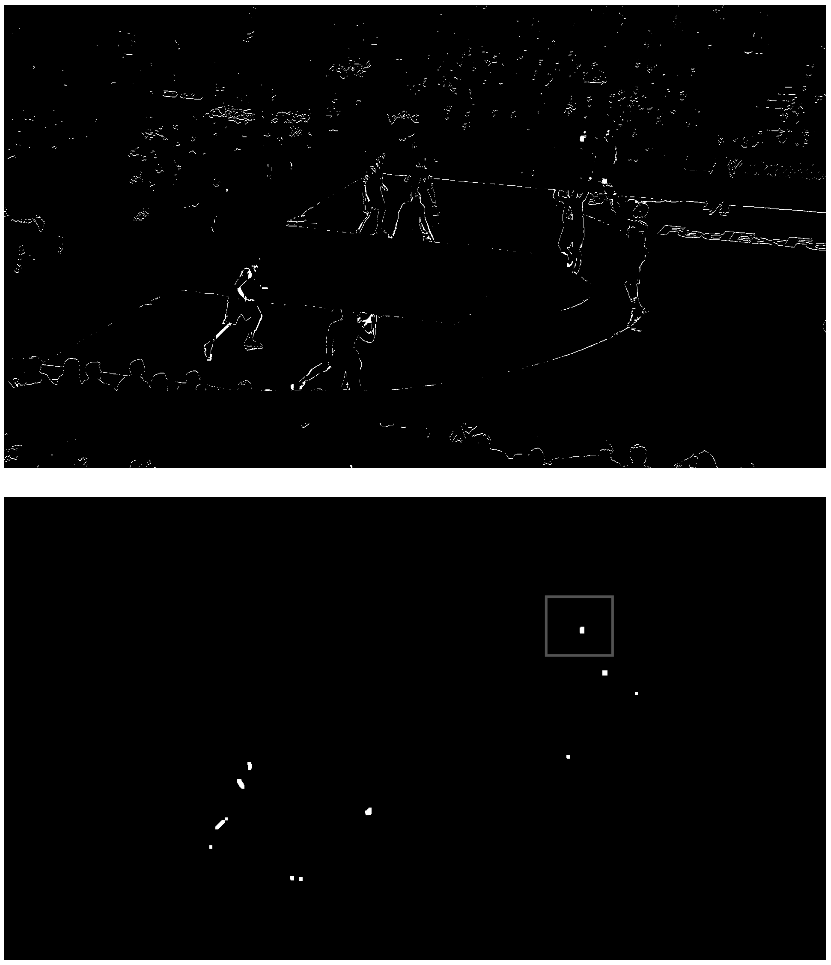


Figure 22: *Top:* An example of the difference between consecutive frames. What appears to be as outlines of the players and the court lines is caused by very subtle camera movements. *Bottom:* The difference between consecutive frames after preforming 3 erosions and dilations to filter out the small noises and bumps, with the actual ball bounded by a box.

8. Now that the number of contours is much smaller, we find all the contours in the search region that match the previous condition, these are our ball candidates.

9. If we have one candidate we choose it as our ball and add the coordinates of the contour's bounding circle's centre to a list. If there's more than one we select the contour that has the most similar color distribution to the color distribution of the ball in the first frame.

10. Continue on to next frame, move the search region in the average direction the ball has moved in its last 3 frames. If there are less than 3, take 2 or 1. If there aren't any previous frames, the offset is fixed at a set value.

11. Repeat steps 3-10.

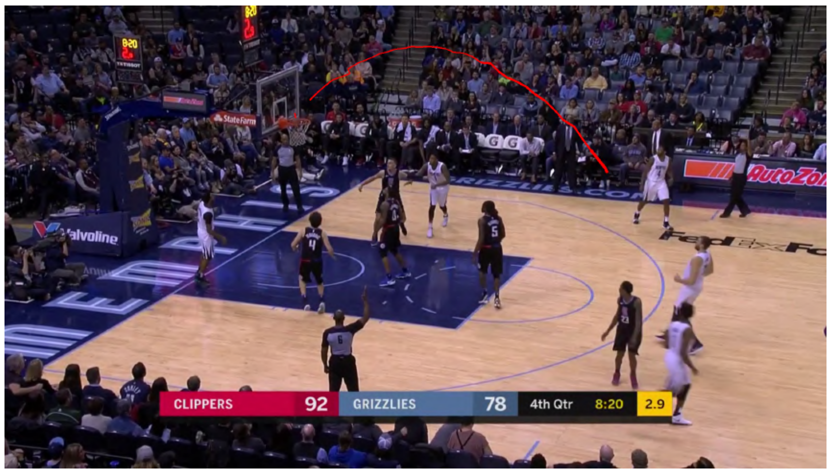


Figure 23: The last frame of the video where the ball is approximately in the middle of the basket, along with the basketball's trajectory in the video.

It can be seen in figure 23 that the ball follows an approximately parabolic line as expected by the laws of physics. The little bumps along the way can be attributed to slight tracking errors as well as small camera movements between frames. The gap in the red line near the end of the shot appears because there were no ball candidates in the search region. Overall in 36/39 frames the tracking was satisfactory. That is more than enough in order to reconstruct the ball's trajectory since we assume parabolic movement of the ball while it's in the air.

3.2 Homography - Detecting Key Court Model Points

3.2.1 Homography

$$p' \propto Hp$$
$$\Leftrightarrow$$
$$\lambda \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \quad (1)$$

Every 2 images of the same plane have a matrix, that maps between corresponding points on that plane (p', p). This matrix is the homography matrix (H). For the purpose of this application, point p is the coordinates of a point on the court in pixel space (denoted by (u, v)) and p' is that same point on the real-life court model (denoted by $(x, y, 0)$). The components of H are the scaling, rotation and translation vectors that encapsulate the mapping between p and p'. In order to be able to create a mapping between the pixels coordinates *(u,v)* to the real-life court model location *(x,y,0)*, we have to track at least 4 points on the court-plane, points of which their real-life location is know. A problem that may arise in tracking points due to the fact that there are 13 people moving simultaneously on the court (5 players per team in addition to the 3 referees) is occlusion. In view of the fact that lines are less likely to be fully occluded than points, we will track lines and then extract their points of intersection (figure 24), rather than just track the points themselves. The steps for this procedure are described in the following sections.

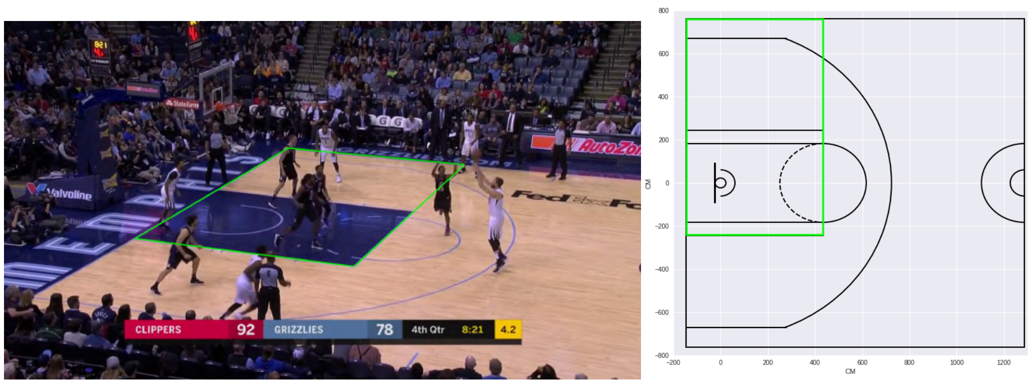


Figure 24: *Left:* The first frame of the video with the points we wish to track being the vertices of the green quadrilateral. *Right:* The coordinate system that the court plane from the image is mapped to with the (0,0) point being the centre of the basket. (shot-chart plot design by Savvas Tjortjoglou (2015))

3.2.2 Tracking Baseline and Sideline

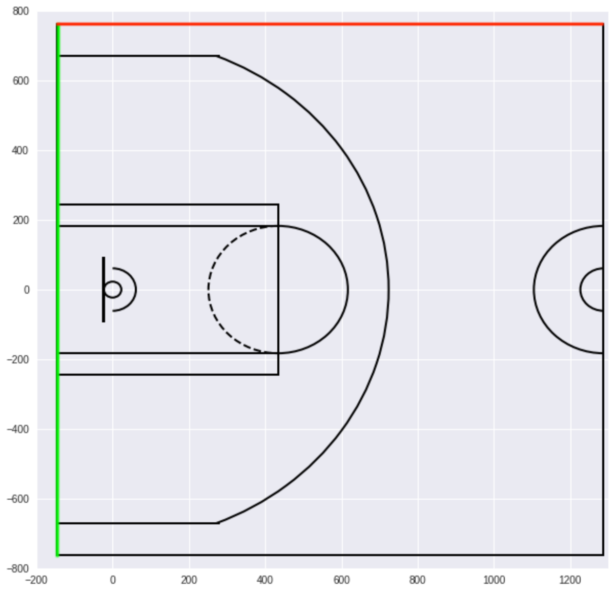


Figure 25: The lines being tracked in this procedure are the baseline (green) and the sideline (red)

1. Similarly to the ball-tracking sub-section, the first step is converting the image from BGR to HSV space.

2. Similarly to the ball tracking process, cropped pictures of the court's floor were taken to gauge the color range of the court.

3. A mask which highlights only the pixels that are in the colour range of the court colours is constructed.

4. Following that, the top 30% of the frame is being cropped out. This is done so that the crowd's appearance in the mask remains minimal (this will make more sense after reading the next step).

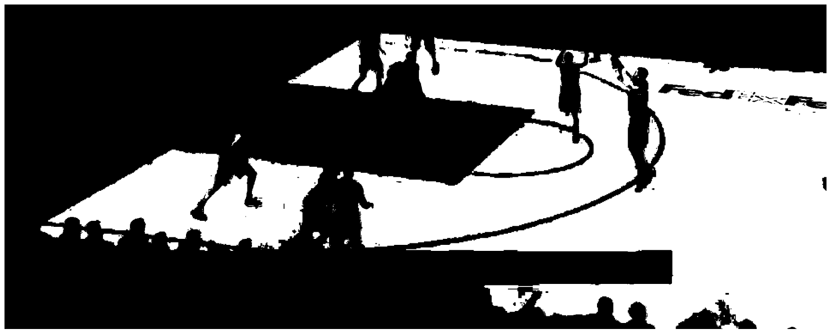


Figure 26: The mask used to detect the sideline and baseline

5. Now, an array is built which contains each column and the location (row wise) of its first white pixel, meaning, the first pixel that is in the specified colour range.

6. Then, the differences between the top pixel location in successive columns are taken.

7. The 3 most common difference values are chosen and sorted.

8. The next step is to detect the *"change points"*, those are defined as the points where the difference is outside the range of the 3 most common differences. These points will serve as an indication of occlusion of the lines that need to be tracked.

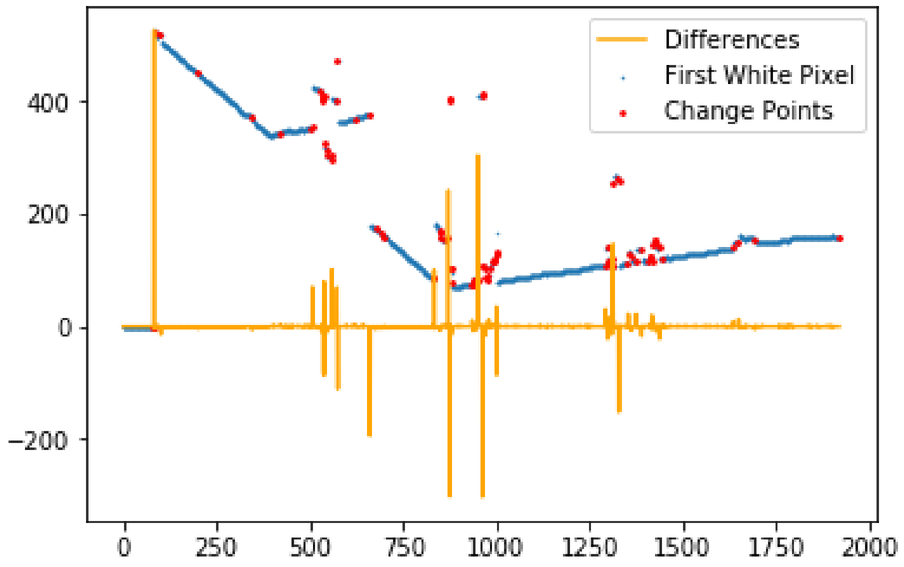


Figure 27: The first white pixel for each column in the mask from figure 26 (blue), the differences between the first white pixel from consecutive columns (orange) and the "change points" (red). Note that in images the *(0,0)* pixel is the **top** left pixel unlike the conventional coordinate system in this plot, that is why this plot seems like an inverted representation of the mask.

9. After that, the 6 longest streaks without a change point are found and extracted.

10. Out of those streaks the 2 longest streaks where the top white pixel has a decreasing trend, these 2 streaks will represent 2 exemplary parts of the baseline, before the paint area and after it.

11. The longest streak where the top white pixel is an increasing function is also taken, this streak represents an exemplary piece of the sideline.

12. After representative pieces of the sideline and baseline are extracted a line is fit using the least squares method to the sideline and the segment of the baseline that intersects with it.

13. The intersection point between the lines is extracted.

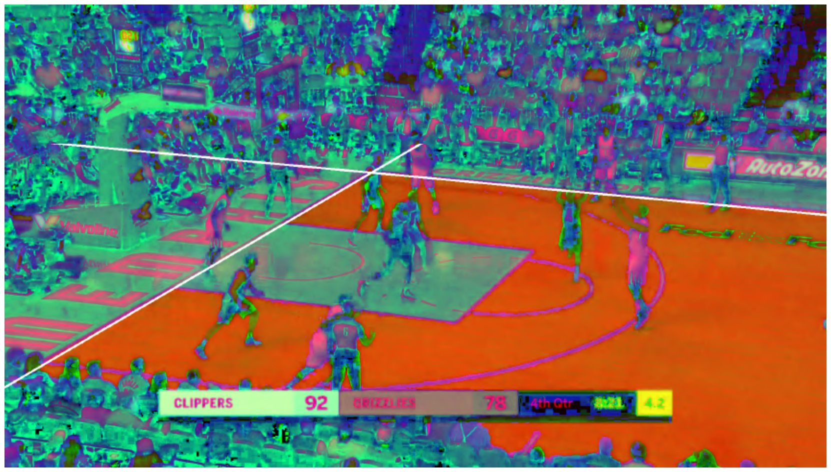


Figure 28: The result for tracking the sideline and baseline in frame 1 of the video.

3.2.3 Tracking Paint and Free-Throw Lines

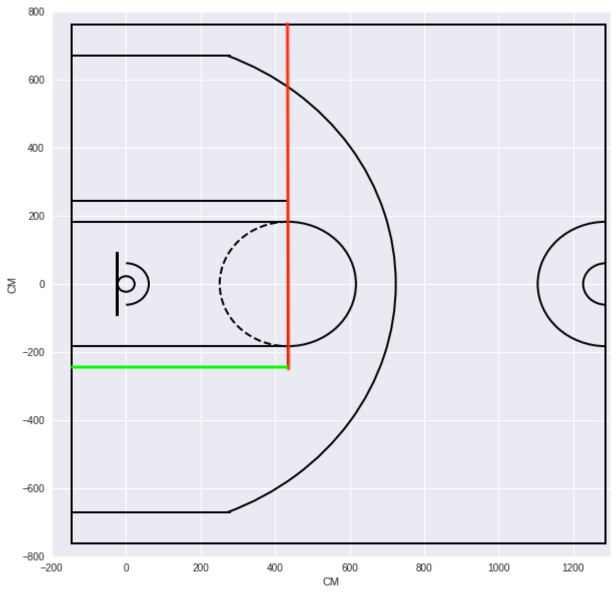


Figure 29: The lines being tracked in this procedure are the free-throw line (extended in red) and one of the paint lines (green)

1. The same mask that we used in the previous section is taken.

2. In step 8 of the previous section, 2 segments of the baseline were extracted, we take the average of the end-point of the segment that comes before the paint area and the start-point of the segment that is after it. This point is referred to as the *"mid-paint point"*.

3. The original mask is then cropped so that it starts from the *mid-paint point.*

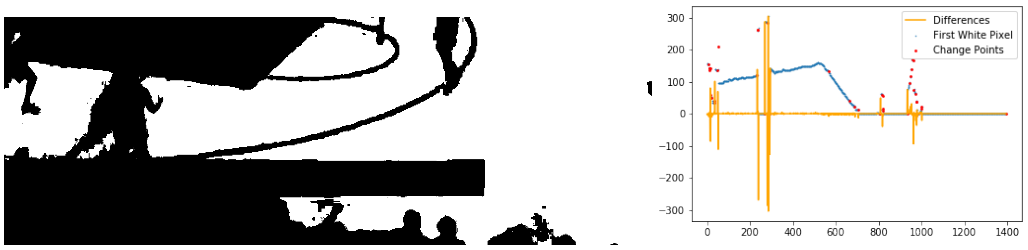


Figure 30: The mask used to detect the paint and free-throw lines along with the first white pixel for each column in the mask (blue), the differences between the first white pixel from consecutive columns (orange) and the "change points" (red).

4. The next steps in order to fit appropriate lines to the close paint line and the free-throw line are done using steps 4-10 from the previous subsection with the following exceptions:

   - In order for a streak without a change-point to be considered as one of the lines that need to be tracked, the top white pixel in the beginning of the streak has to be on a different height than the last one. This addition prevents the side of the mask (right of the free-throw line to be mistaken for one of the lines).

   - Here only the longest increasing streak and the longest decreasing streak are considered and to those the least squares model line is fit.

5. Once these lines are fitted, the coordinates (in pixel space) of the intersections of the 4 lines are extracted and saved.

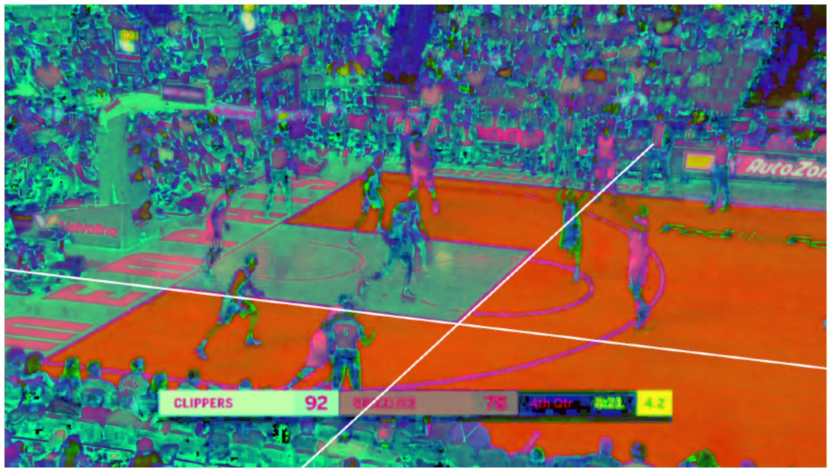


Figure 31: The results of tracking for paint and free-throw and paint lines in frame 1 of the video.

Now that we have 4 points on the floor that are tracked. These points, along with the information we have of their real-life location define the mapping from pixel space *(u,v)* into the court-plane *(x,y,0)*. Using OpenCV's *solvePnP* function feeding in the coordinates of the points we found along with their respective real-life location, returns the rotation and translation vector which define the homography matrix, H. It's worth noting that this transformation will bring inaccurate results if trying to track a point that is on a plane higher than the floor since only the court's plane is mapped. This is a problem that needs to be solved since the object that needs to be tracked is the ball, which is naturally, above floor level in every frame during the shot. Thus, another (3D) plane needs to be accounted for, this plane is referred to as the *shot-plane*.

3.3 Finding The Shot Plane & Extracting Ball's X, Y Coordinates

The shot plane is the plane that is perpendicular to the court's floor, which the shot trajectory is physically glued to. The shot-plane can be defined by the shot's point of release and the point where the ball would contact the plane defined by the backboard (if the basket itself wasn't there), this point will be referred to as the *"contact point"*. For this reason it wouldn't be possible, using the algorithm suggested in this paper to track shots that are released close to the baseline.

The shot-plane is obtained by following these steps:

1. First, the pixel coordinates of the shooter's feet in the first frame are manually taken to estimate the shooting location.

2. Next, the rotation and translation vectors found previously are used to get the shooting location's real-life coordinates. This is also a good way to assess the quality of the homography mapping.

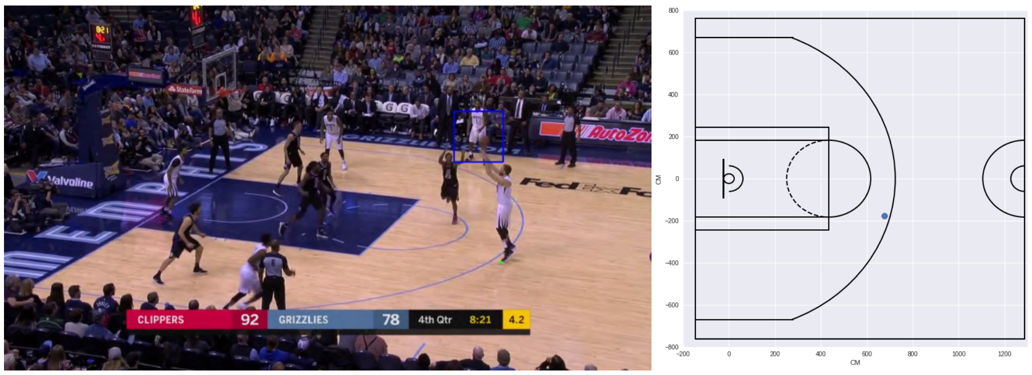


Figure 32: The video's first frame, with the shooting location marked by a green circle, along with the corresponding location according to the computed homography mapping.

3. Next, the shot's projection on the court plane is estimated. This requires projecting the real-life location of the point $(x_{mid-basket} = 0, y_{mid-basket} = 0, 0)$ on to the image, this is where the court homography we computed previously comes into play. Recall equation 1 where the homography matrix H was used to map the image plane into the real-life court plane. In the same manner that H was used, H^{-1} can be used to define the inverse mapping from the real-life court model to the image plane, that is $p' = Hp \Leftrightarrow H^{-1}p' = p$. Using OpenCV's *projectPoint* function makes use of this and allows to re-project point (0,0,0) onto the image plane.

4. A line is drawn between the shooting location and the re-projected (0,0,0) point, this line is the *shot projection*.

5. Since the shot plane is perpendicular to the floor, these are the only points needed to define the shot-plane.

6. A perpendicular is dropped down from the ball (which we have its pixel coordinates from the ball tracking section).

7. In a given frame, the point of intersection between this perpendicular and the shot projection along with H defines the *(x,y)* coordinates of the ball.

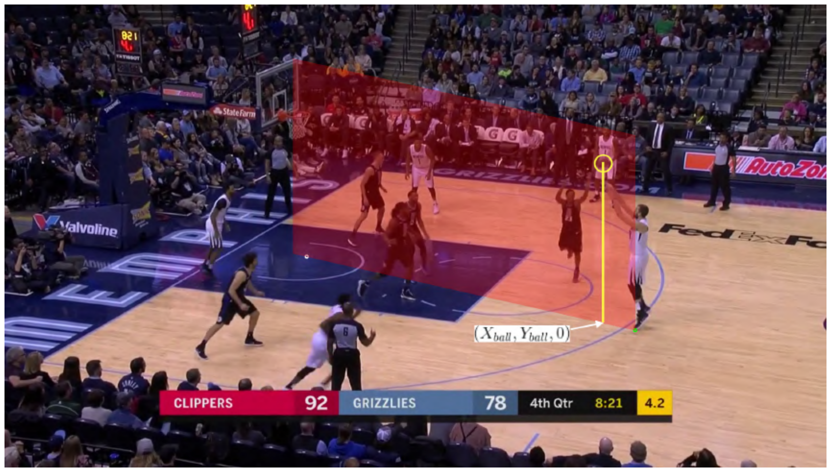

$(X_{ball}, Y_{ball}, 0)$
CLIPPERS 92 GRIZZLIES 78 4th Qtr 8:21 4.2

Figure 33: An illustration of the shot plane. The green circle represents the player's shooting location on the floor and the white circle is the point $(0, 0, 0)$ in the court model (which is the point right under the middle of the basket.)

3.4 Extracting Ball's Height

Now that the x, y coordinates of the ball are obtained, all that is left to complete the trajectory reconstruction is to get its Z coordinate, the height of the ball. To do this, we need to find an object that is convenient to track during the video, with a height that is constant and known.

3.4.1 Tracking The Backboard's Edges

Due to the fact that the top of the backboard is so high (3.96 meters above the height of the court's floor), there are not meant to be tracking issues caused by occlusion which makes it a good candidate. In order to pull out the pixel coordinates for the top edges of the backboard the following steps are taken:

1. In similar fashion to before, The original frame is taken and converted to HSV color space.

2. Similar to the ball tracking process, several images of the backboard edges were taken in order to gauge their color range.
3. A mask which only highlights the pixels that are in the color range of the backboard edges is created.
4. Morphological operations (erosions and dilations) are preformed on the image to remove small noises and blobs.
5. Now, the contour in the mask with the biggest area is chosen.
6. That contour is then bounded by a rectangle.
7. The original mask is then reduced to the coordinates of that rectangle with some padding (15% of the rectangle's width are added in each side).

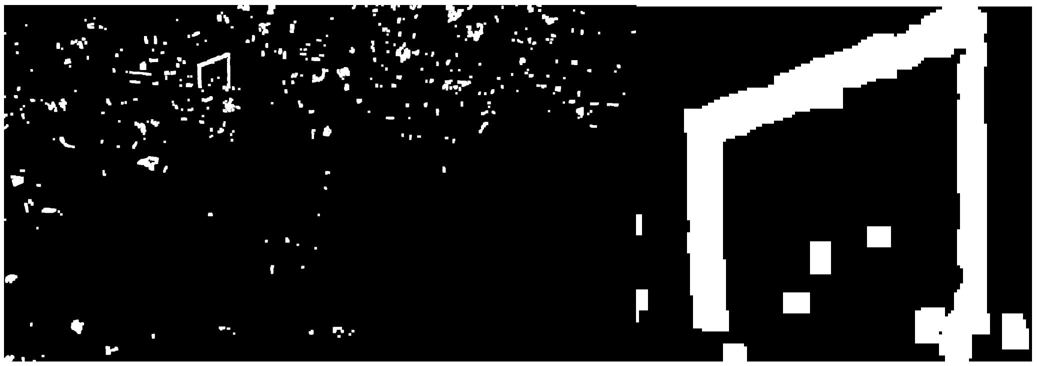


Figure 34: The mask used in order to track the backboard's top edges *(left)* along with the cropped mask from the same frame *(right)* which shows correct detection of the backboard.

8. In order to fit a line to the top of the backboard, the same algorithm as the one to track the court lines is used with the exception that now, only the longest streak without a "change-point" is taken and fitted a line to.
9. As a result of the shape and orientation of the backboard, the sides of it will always appear as practically vertical lines. For this reason, less work needs to be done to fit a line to to the backboard's sides.
10. An array is constructed which gives us each row and its corresponding column of the first white pixel for that row using the *argmax()* function.

11. The column which appeared the most times in the array above is picked as the left side of the backboard and a vertical line is drawn accordingly. (figure 35

12. Multiplying the mask by an array with the sequence of numbers from 0 to the width of the mask (using the *numpy.arange()* function) and taking the maximizing argument of that array along the row axis will return each row and the column of its corresponding most right pixel.

13. The column that appeared the most time is selected as the right side of the backboard and a vertical line is drawn accordingly.

14. Finally, the intersections of the vertical lines and the top of the backboard are extracted and saved.

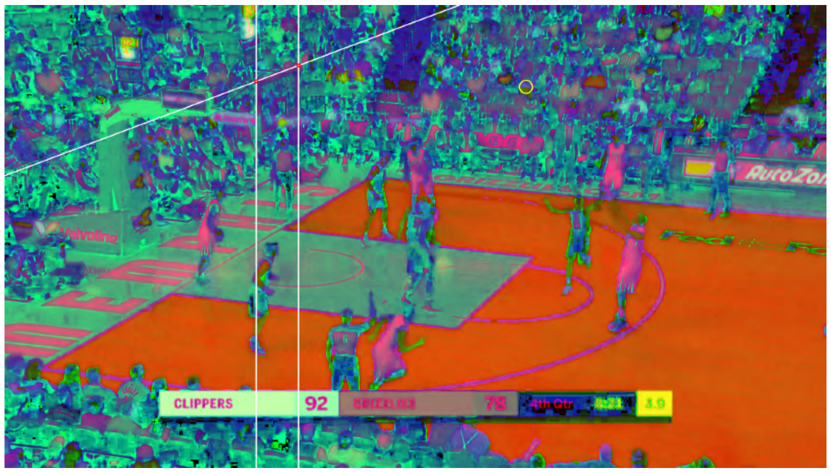


Figure 35: The results of the backboard tracking in frame 9 of the video.

Now that all the necessary points and lines are tracked, the Z coordinate of the ball can be found.

3.4.2 Using the Backboard to Find the Ball's Z-Coordinate

1. In the same way that the $(0, 0, 0)$ was projected onto the image plane in the shot plane extraction process, the points:
$(X_{backboard-edge1} = -22.9, Y_{backboard-edge1} = -91.4, 0)$,

$(X_{backboard-edge2} = -22.9, Y_{backboard-edge2} = 91.4, 0)$ are projected onto the image plane as well.

2. A line is then drawn between these points which represents the projection of the backboard on the court floor.

3. The point of intersection between the *shot plane* and this line is saved.

4. a perpendicular taken from this point to the top of the backboard. The length of this perpendicular line in the real-world is exactly the height of the backboard (3.96 meters according to NBA regulations)

5. Getting the length of that perpendicular in the image plane defines a mapping from height(pixels) to height(centimetres) along the *shot plane*.

6. Now all that is left to do is drop a perpendicular from the ball to the shot-projection, take its height in pixels and convert it to centimetres using this mapping.

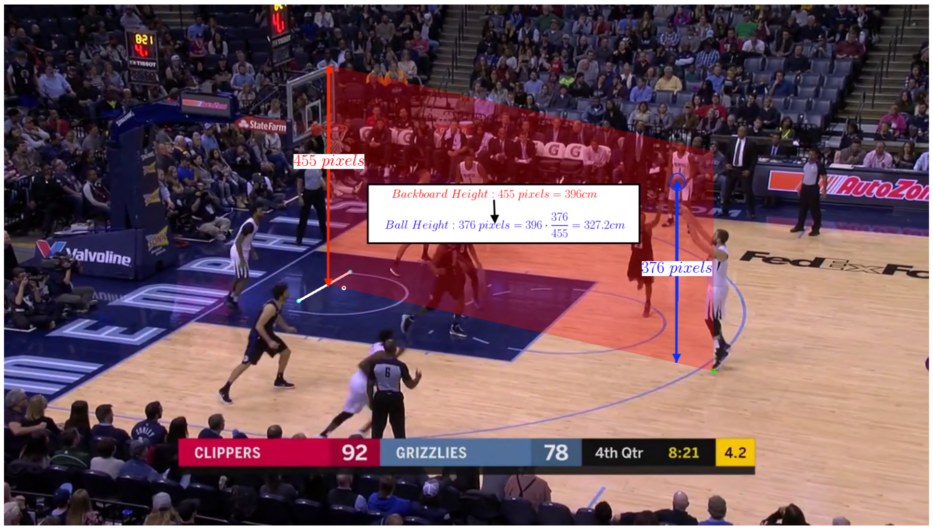

455 pixels
Backboard Height : 455 pixels = 396cm
Ball Height : 376 pixels = 396 · 376/455 = 327.2cm
376 pixels
CLIPPERS 92
GRIZZLIES 78
4th Qtr 8:21 4.2

Figure 36: An illustration of the ball's height extraction process. The white circle and line on the floor are the projected $(0, 0, 0)$ point and the projection of the backboard on the court's floor respectively. The double arrows are the perpendiculars from the top of the backboard (red) and from the ball (blue) to the shot projection.

3.5 Fitting A Curve to the Ball's Trajectory and Getting the Velocity and Angle Of Shot Release

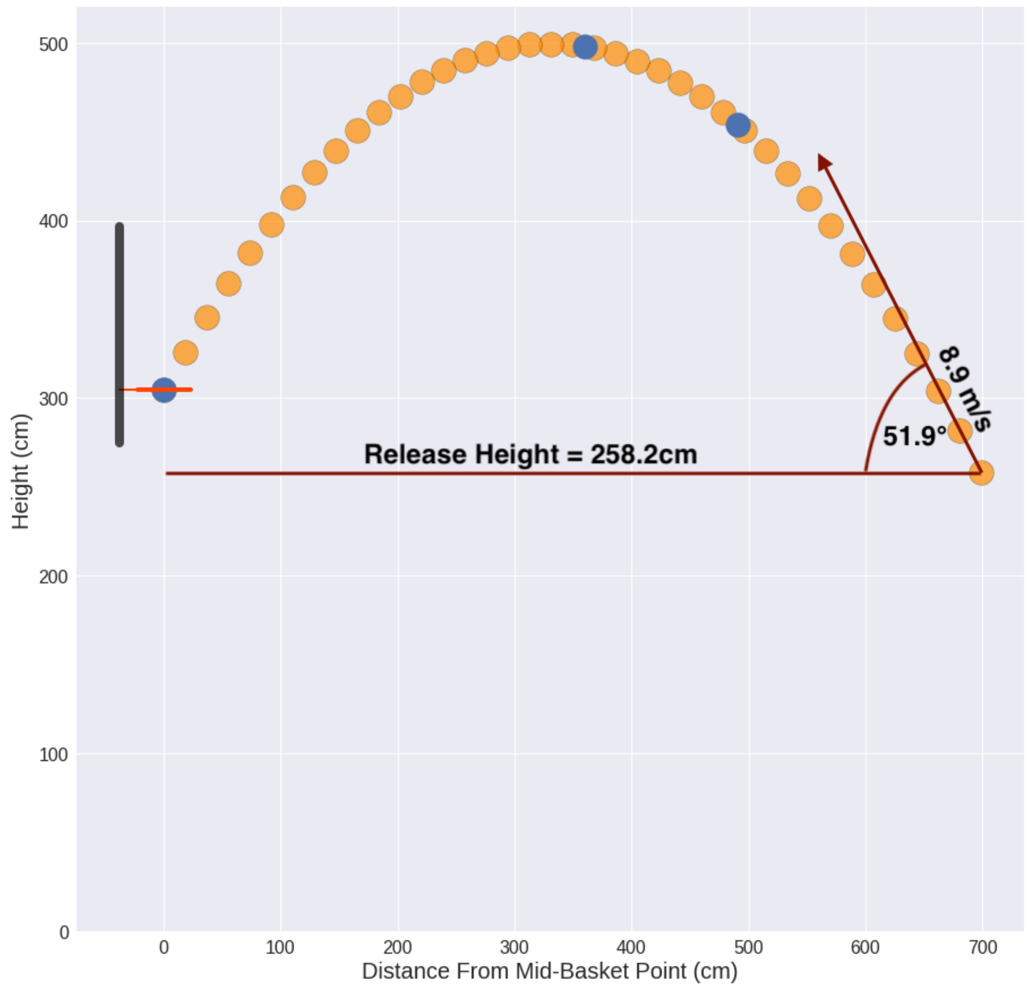


Figure 37: The reconstructed trajectory of the shot with an illustration of the basket. The blue markers are the points that the parabola is fitted to. The shot release height might seem a bit excessive, however please note that the shooting player is 2.16 meters tall with a 2.24 meters wingspan.

As a result of the abundance of lines, points and objects that need to be tracked in every frame, there are only a handful of frames where all of those are tracked adequately. The 3D coordinates of the ball are taken in those frames. In addition to that, the coordinate of the middle of the basket is added to reflect the fact that the shot was a made basket. Since the research's aims involve reconstructing the shot's parameters (velocity and angle) as a function of the shot's distance, the x, y coordinates are reduced to one "distance-from-basket" coordinate (denoted as $d = \sqrt{x^2 + y^2}$) and then fitted a parabola (denoted as $h(d)$) to. In pursuance of getting the release angle (θ) and velocity (V_0) we calculate the speed in the horizontal axis and vertical axis using the information that there are 1/30 seconds between consecutive frames:

$$\left.\begin{array}{r}V_{0d} = \frac{d[1]-d[0]}{1/30} = -551cm/s = -5.51m/s\\ V_{0h} = \frac{h(d[1])-h(d[0])}{1/30} = 705cm/s = 7.05m/s\end{array}\right\} \Rightarrow V_0 = \sqrt{V_{0d}^2 + V_{0h}^2} = 8.9m/s \quad (2)$$

$$\theta = \arctan(\frac{V_{0h}}{V_{0d}}) = \arctan(\frac{7.05}{5.51}) = 51.9^\circ \quad (3)$$

4 Conclusions & Future Work

This project features a system, based on pure computer vision combined with analytic geometry for 3D trajectory reconstruction of a basketball shot. While there are limitations to the system's accuracy due to approximations of the shot-plane and tracking errors this project achieved the objectives of breaking down a video into frames and implementing a feature detection algorithm able to track the lines needed to compute the homography mapping of the court. This combined with the system's ball tracker allows for achieving the key objective of reconstructing the basketball's trajectory throughout the course of the shot. Primarily due to time constraints, the system was only successfully tested on one proof of concept video, therefore not achieving the objective of getting the optimal shot parameters as a function of distance from the basket. More steps can be taken to make the system more robust as described in the future works section. The aim/idea of computing a shot's success probability given a partial video of the shot, is problematic since the step of assessing the shot-plane is dependant on the fact that the ball will reach the basket. The future work of this paper can be divided into 2 sections, The short-term work that can be done to improve the system's performance and the work that can be done to build on that once the system is working accurately enough.

4.1 Short-Term

Given more time to improve this model, I have multiple additions to the model to make it more automated and to increase its quality and reliability.

4.1.1 Using Piece-wise Linear fit for the Line Tracking

The line tracking algorithm sometimes struggles identifying when one line starts and another begins (for example, where the paint-line turns into the free-throw line. This causes less than ideal line fitting. Given more time I would change the algorithm to first remove the outliers of the line and then fit a piece-wise linear model to find the break point where one line ends and the other begins. With so many outliers caused by occlusion, it isn't a simple task.

4.1.2 Checking For the "True" Shot Plane

The fact that the shot release point's projection on the court is approximated by the shooter's feet, is a limitation that makes for an inaccurate shot-plane, which affects the system's results. What can be done to try and improve this approximation is to sample several points around the shooter's feet and add a Bayesian component to the model which computes the likelihood of the shot plane being the "real" shot plane given our tracked points, and then selecting the shot release point that defines the plane with the maximum likelihood. By incorporating this idea, it can also be possible to similarly sample points on the other side of the shot-plane, underneath the basket, thus assessing the probability of a made basket.

4.1.3 Adding a Frame Filter

As the algorithm stands now, the frames where all the moving parts are tracked adequately are chosen manually. Even though it doesn't take much time to see which frames have decent tracking, a relatively simple way to skip this step is to simply keep track of the lines fitted to the baseline, sideline, paint and free-throw line, particularly look at their slope. Since these lines create a perfect rectangle in the real-world, and as such they are built of 2 sets of parallel lines, even though in the image plane the lines are not exactly parallel their slopes still have similar values. This way the frames where the 2 sets of lines have very different slopes can be flagged as frames with poor tracking, and then disqualified from the curve-fitting stage.

4.1.4 Validation

To validate the results of this paper, I would select a video with a shot that is ideally already tagged with its true velocity and angle. If no such video exists (I couldn't find any at the time of writing this paper), I would select a shot with multiple camera angles, this creates a new dimension of depth making for more accurate results.

4.1.5 Getting the Optimal Shot Parameters per Shot Distance

Once The system is properly validated, more videos can be processed to check what are the optimal shot-parameters as a function of the distance from the basket.

4.2 Long-Term Future Work

Presented here are some possible directions to build on this research.

4.2.1 Tracking Additional Shot Parameters - Elbow Angle

Basketball trainers claim it is important that the shooter's elbow will be at approximately a 85-90 degree angle at the set-point of the shot (2017). There are even multiple products out there to help basketball players incorporate this into their shooting mechanics. (figure 38) For this reason, another important component of the shot that can be analyzed using human pose estimation is the shooter's elbow angle at the set-point of the shot.

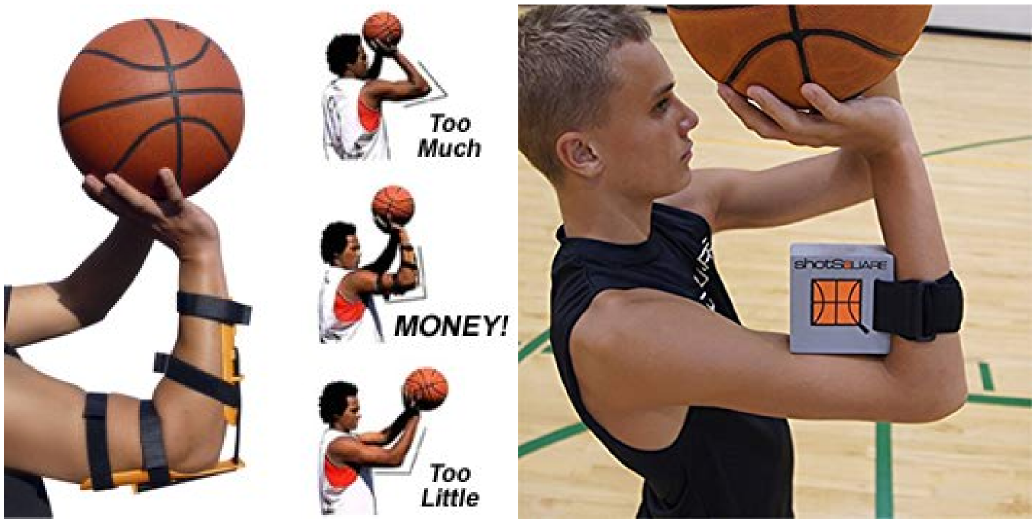


Figure 38: Examples of 2 products used to train basketball players to fix their elbow angles at 90 degrees.

4.2.2 Future Research - Automatic State Machine

As it currently stands, the system has to be fed in a video where the first frame is the frame where the ball leaves the shooter's hand. Since the ball is being tracked, we can detect and point to frames where the ball accelerates suddenly, these frames indicate when the ball is released from a players hand, thus automating the video snipping process. Similarly, the frames where the ball decelerates suddenly, or where tracking is lost indicate the ball's natural movement has been disrupted, either by contact with the basket or by another player deflecting or catching the ball. The ability to detect these disruptions in real-time can have great consequences on the future of basketball. This is because the game-clock is only supposed to start when the ball touches another player on the court. The time-keeping involves a human error component since the game clock is manually started and stopped by a human operator and the average human reaction time to visual stimuli is approximately 0.284 seconds (2015*a*). In a sport where every fraction of a second matters this can and has in the past, decided the winner of a match. An example for that would be Derek Fisher's game winning shot with 0.4 seconds left in a crucial playoff game in 2004. it can be seen in figure 39 that the clock stands at 0.4 seconds while Fisher catches the ball, jumps forward and rotates his body, a seemingly impossible feat for a tenth of a second.

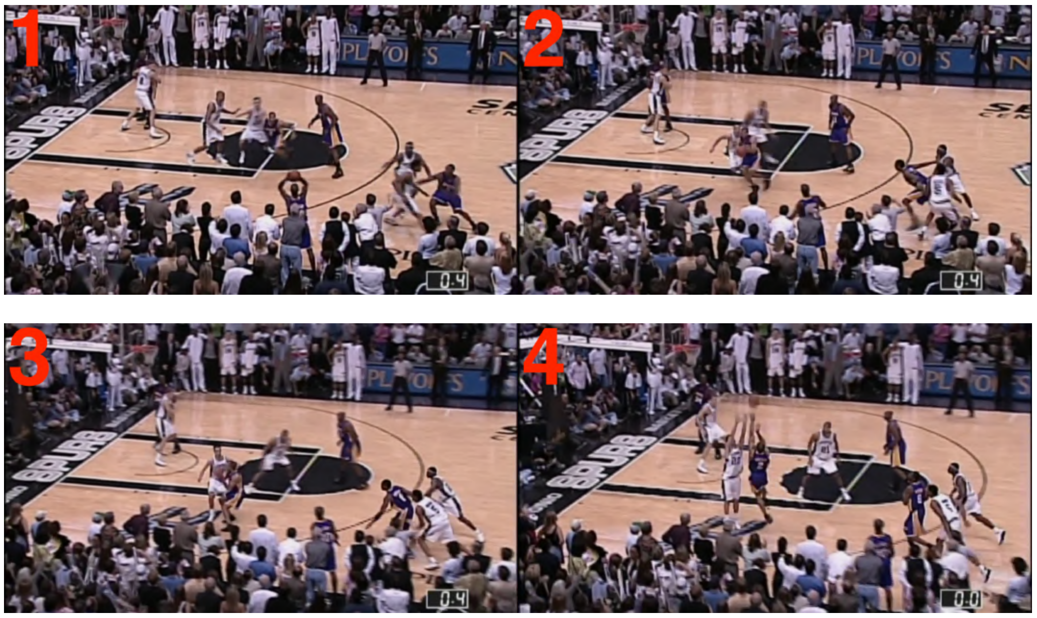


Figure 39: The sequence of Fisher's shot. *1)* The moment the pass is in bounded, *2)* The moment the ball is caught (where the clock should have started), *3)* Fisher takes another step, starts to jump and contort his body while the clock is still at 0.4. *4)* The moment the game clock expired, it can be seen that the ball barely leaves Fisher's fingertips, thus making the shot legal.

5 Appendix (Source Code)

This section contains the important parts of the code for the trajectory extraction system.

5.1 Finding Court Model Lines

```
#   seperate function to find intersection between 2 lines
def find_intersect(l1p1,l1p2,l2p1,l2p2,l1vert=False):
   """
   Function to find intersect between lines
   """
   if l1vert:
     x = l1p1[0]
     m2 = (l2p2[1]-l2p1[1])/(l2p2[0]-l2p1[0])
     n2 = l2p1[1]-m2*l2p1[0]
     y  = m2*x+n2
     return int(x),int(y)
  m1 = (l1p2[1]-l1p1[1])/(l1p2[0]-l1p1[0])
  n1 = l1p1[1]-m1*l1p1[0]
  m2 = (l2p2[1]-l2p1[1])/(l2p2[0]-l2p1[0])
  n2 = l2p1[1]-m2*l2p1[0]
#   fix for errors
   if m1==m2:
     return (-1,-1)
  x  = (n2-n1)/(m1-m2)
  y  = m2*x+n2
  return int(x),int(y)

# Class for BBall Frames
class BBall_Frame:
  def __init__(self,frame,shot_loc_3d= (676.4,-175.6,0) ):
    self.orig_frame          = frame
    self.shot_loc_3d         = shot_loc_3d
    self.baseline_sideline = None
    self.paint_ft            = None
    self.paint_sideline     = None
    self.sideline_ft         = None
    self.paint_mid           = None
    self.court_mask          = None
    self.frame               = None
  def find_court_points(self,show_plot=False):
    # frame = cv2.imread('court.png')
    # convert color
    frame = cv2.cvtColor(self.orig_frame,cv2.COLOR_BGR2HSV)
    # mask the court colors
    lower = (5,50,140)
    upper = (15,105,250)
```

```
# Ignore the top 30% of the frame (crowd noise)
mask = cv2.inRange(frame[int(frame.shape[0]*0.3):,:],lower,upper)
mask = cv2.erode(mask,kernel=None,iterations=0)
mask = cv2.dilate(mask,kernel=None,iterations=0)
self.court_mask = mask
# Find first white pixel in each column (y_coordinate) and make array of x,y point
first_white_pix= np.array(list(zip(np.arange(mask.shape[1]),mask.argmax(0))))



#differences between consecutive columns
difs = np.diff(first_white_pix[:,1])

# Get the slopes for the baseline and sideline
(values,counts)              = np.unique(difs,return_counts=True)
ind                          = np.argsort(counts)[::-1]
most_common_difs             = np.sort(values[ind][:3])
baseline_dif,sideline_dif = most_common_difs[0],most_common_difs[-1]

# plot differences on top of first white pixel plot
# show plot
if show_plot:
  plt.scatter(first_white_pix[:,0],first_white_pix[:,1],s=.2)
  plt.plot(difs,color='orange')

# Find change points (points where the difference between that point and the previous is more th
change_points = np.append(np.nonzero(np.abs(difs)> max(np.abs(baseline_dif),np.abs(sideline_dif

# get top 4 streaks start & end indices
streak_lengths              = np.diff(change_points)
# take 6 longest streaks
longest_streaks_start_idxs = change_points[np.sort(np.argsort(streak_lengths)[:: -1][:6])]
longest_streaks_end_idxs   = change_points[np.nonzero(longest_streaks_start_idxs[:, None] == ch
longest_streaks_idxs       = np.vstack((longest_streaks_start_idxs,longest_streaks_end_idxs)).T


# sort streaks to neg/pos
signs_list = []
for streak_idx in longest_streaks_idxs:
  signs_list +=[first_white_pix[streak_idx[1]-1][1]-first_white_pix[streak_idx[0]+1][1]]


 # Only keep baseline1,baseline2, sideline
signs_list                 *= (longest_streaks_end_idxs-longest_streaks_start_idxs)
desired_streaks_idxs        = np.argsort(signs_list)[[0,1,-1]]
longest_streaks_start_idxs = longest_streaks_start_idxs[desired_streaks_idxs]
```

```
        longest_streaks_end_idxs    = longest_streaks_end_idxs[desired_streaks_idxs]

        # Extract Y coordinate of middle of baseline1 and baseline2 (paint_mid) to be used for second c
        baseline1_end    = first_white_pix[longest_streaks_end_idxs[0]-1]
        baseline2_start = first_white_pix[longest_streaks_start_idxs[1]+1]
        self.paint_mid      = int((0.4*baseline1_end+0.6*baseline2_start)[0])  ,int((0.4*baseline1_end+0.

        # First court mask
        copy          = frame.copy()
        lefty_list   = []
        righty_list = []
        for i in range(len(longest_streaks_end_idxs)):
          line    = first_white_pix[longest_streaks_start_idxs[i]+1:longest_streaks_end_idxs[i]]
          [vx,vy,x,y]= cv2.fitLine(line,cv2.DIST_FAIR,0,0.01,0.01)
        # Now find two extreme points (these are just Y values) on the line to draw line and add to com
          lefty          = int((-x*vy/vx) + y) + int(frame.shape[0]*0.3)
          righty         = int(((mask.shape[1]-x)*vy/vx)+y) + int(frame.shape[0]*0.3)
          lefty_list   += [(0,lefty)]
          righty_list += [(frame.shape[1]-1,righty)]



#     find intersect point between 2 lines
        self.baseline_sideline = find_intersect(lefty_list[1],righty_list[1], lefty_list[2],righty_lis
  #Finally draw the lines
cv2.line(copy,lefty_list[1],righty_list[1],(255,255,255),2)
cv2.line(copy,lefty_list[2],righty_list[2],(255,255,255),2)
cv2.circle(copy, (self.baseline_sideline), 3, (0, 0, 255), 2)



self.frame = copy
# SECOND MASK
# Cut the first mask to the mid paint pixel
mask2             = self.court_mask[self.paint_mid[1]:,self.paint_mid[0]:]
# Find first white pixel in each column (y_coordinate) and make array of x,y points
first_white_pix   = np.array(list(zip(np.arange(mask2.shape[1]),mask2.argmax(0))))

#differences between consecutive columns
difs              = np.diff(first_white_pix[:,1])

# Get the slopes for the baseline and sideline
(values,counts)             = np.unique(difs,return_counts=True)
ind                         = np.argsort(counts)[::-1]
most_common_difs            = np.sort(values[ind][:5])
ft_dif,paint_dif            = most_common_difs[0],most_common_difs[-1]
```

```
# plot differences on top of first white pixel plot
# show plot
if show_plot:
    plt.scatter(first_white_pix[:,0],first_white_pix[:,1],s=.2)
    plt.plot(difs,color='orange')

# Find change points (points where the difference between that point and the previous is more th
change_points = np.append(np.nonzero(np.abs(difs)> max(np.abs(ft_dif),np.abs(paint_dif)))[0],len

# get top 4 streaks start & end indices
streak_lengths             = np.diff(change_points)
# take 6 longest streaks (should be enough to get a sufficiently long positive and negative str
longest_streaks_start_idxs = np.asarray(change_points[np.sort(np.argsort(streak_lengths)[::-1][
longest_streaks_end_idxs   = np.asarray(change_points[np.nonzero(longest_streaks_start_idxs[:,
longest_streaks_end_idxs   = longest_streaks_end_idxs -0.6*(longest_streaks_end_idxs-longest_str
longest_streaks_end_idxs   = longest_streaks_end_idxs.astype(int)
longest_streaks_idxs       = np.vstack((longest_streaks_start_idxs,longest_streaks_end_idxs)).T

# sort streaks to neg/pos
signs_list = []
for streak_idx in longest_streaks_idxs:
    signs_list +=[first_white_pix[streak_idx[1]-1][1]-first_white_pix[streak_idx[0]+1][1]]
```

```
# Only keep longest positive and longest negative (paint and FT line)
signs_list                 *= (longest_streaks_end_idxs-longest_streaks_start_idxs)
desired_streaks_idxs        = np.argsort(signs_list)[[-1,0]]
longest_streaks_start_idxs = longest_streaks_start_idxs[desired_streaks_idxs]
longest_streaks_end_idxs   = longest_streaks_end_idxs[desired_streaks_idxs]

# Draw lines
copy = self.frame.copy()
for i in range(len(longest_streaks_end_idxs)):
  line    = first_white_pix[longest_streaks_start_idxs[i]+1:longest_streaks_end_idxs[i]]+(self.p
  [vx,vy,x,y]= cv2.fitLine(line,cv2.DIST_FAIR,0,0.001,0.001)

# Now find two extreme points on the line to draw line
  lefty = int((-x*vy/vx) + y) +int(frame.shape[0]*0.3)
  righty = int(((copy.shape[1]-x)*vy/vx)+y) + int(frame.shape[0]*0.3)

  lefty_list  += [(0,lefty)]
  righty_list += [(frame.shape[1]-1,righty)]

self.paint_ft          = find_intersect(lefty_list[3],righty_list[3], lefty_list[4], righty_list[4
self.paint_sideline = find_intersect(lefty_list[3],righty_list[3], lefty_list[1], righty_list[1
self.sideline_ft       = find_intersect(lefty_list[2],righty_list[2], lefty_list[4], righty_list[4
        cv2.line(copy,lefty_list[3],righty_list[3],(255,255,255),2)
        cv2.line(copy,lefty_list[4],righty_list[4],(255,255,255),2)
        cv2.circle(copy, (self.paint_ft), 3, (0, 0, 255), 2)
        cv2.circle(copy, (self.paint_sideline), 3, (0, 0, 255), 2)
        cv2.circle(copy, (self.sideline_ft), 3, (0, 0, 255), 2)

        self.frame = copy

    def find_backboard_points(self,show_plot=False):
#        frame = cv2.cvtColor(self.orig_frame,cv2.COLOR_BGR2HSV)
        frame = self.frame
        lower1 = (120,10,50)
        upper1 = (165,80,130)

        mask = cv2.inRange(frame,lower1,upper1)
        mask = cv2.erode(mask,kernel=None,iterations=3)
        mask = cv2.dilate(mask,kernel=None,iterations=3)
        cnts = cv2.findContours(mask.copy(), cv2.RETR_EXTERNAL,
        cv2.CHAIN_APPROX_SIMPLE)
        cnts = imutils.grab_contours(cnts)
    center = None

    # find the largest contour in the mask, then use
    # it to compute the minimum enclosing circle and
    # centroid
    c = max(cnts, key=cv2.contourArea)
    ((x, y), radius) = cv2.minEnclosingCircle(c)
   M = cv2.moments(c)
    center = (int(M["m10"] / M["m00"]), int(M["m01"] / M["m00"]))

# only proceed if the radius meets a minimum size
    # draw the circle and centroid on the frame,
    # then update the list of tracked points
    x,y,w,h = cv2.boundingRect(c)
# Centerize x and increase window size
    x       = int(x-0.15*w)
    w,h     = int(w*1.3), int(h*1)

    backboard_mask = mask[y:y+h,x:x+w]



#  First we get the top of the backboard
```

```
    top_white_pix   = np.array(list(zip(np.arange(backboard_mask.shape[1]),backboard_mask.argmax(0))
    difs = np.diff(top_white_pix[:,1])
    # Get the slopes for the baseline and sideline
    (values,counts)             = np.unique(difs,return_counts=True)
    ind                         = np.argsort(counts)[::-1]
    most_common_difs            = np.sort(values[ind][:2])
    change_points               = np.append(np.nonzero(np.abs(difs)> max(np.abs(most_common_difs)))[0
    # get top 4 streaks start & end indices
    streak_lengths               = np.diff(change_points)
    # take longest streak
    longest_streaks_start_idxs = change_points[np.sort(np.argsort(streak_lengths)[::-1][:1])]
    longest_streaks_end_idxs   = change_points[np.nonzero(longest_streaks_start_idxs[:, None] == ch
    longest_streaks_idxs       = np.vstack((longest_streaks_start_idxs,longest_streaks_end_idxs)).T

    copy                       = frame.copy()
    line                       = top_white_pix[longest_streaks_start_idxs[0]:longest_streaks_end_idx

    [vx,vy,x0,y0]= cv2.fitLine(line,cv2.DIST_FAIR,0,0.001,0.001)

# Now find two extreme points on the line to draw line and add to compensate for cropped mask
    lefty = int((-x0*vy/vx) + y0)
    righty = int(((copy.shape[1]-x0)*vy/vx)+y0)
    #Finally draw the line
    cv2.line(copy,(frame.shape[1],righty),(0,lefty),(255,255,255),2)
# Find left backboard value
    leftiest_white_pix          = np.array(list(zip(np.arange(backboard_mask.shape[0]),backboard_mask
    (_,counts2)                 = np.unique(leftiest_white_pix,return_counts=True)
    left_backboard_value        = np.argmax(counts2)
#   add x value to compensate for mask
    left_backboard_value       += x

# Find Right backboard value
    temp = backboard_mask*(np.arange(backboard_mask.shape[1]))
    rightest_white_pix  = np.array(list(zip(np.arange(backboard_mask.shape[0]),temp.argmax(1))))

    # Get the slopes for the baseline and sideline
    (_,counts3)                 = np.unique(rightest_white_pix,return_counts=True)
    right_backboard_value       = np.argmax(counts3)
#   add x value to compensate for mask
    right_backboard_value      += x
    cv2.line(frame,(right_backboard_value,frame.shape[1]),(right_backboard_value,0),(255,0,0),1)

# Now that we have all thew lines, we extract the points
    self.top_left_backboard        =find_intersect((left_backboard_value,0),(left_backboard_value,co
    self.top_right_backboard       =find_intersect((right_backboard_value,0),(right_backboard_value,
        cv2.circle(copy, (self.top_left_backboard), 3, (0, 0, 255), 2)
        cv2.circle(copy, (self.top_right_backboard), 3, (0, 0, 255), 2)
        if show_plot:
            plt.scatter(top_white_pix[:,0],top_white_pix[:,1],s=.2)
            plt.plot(difs,color='orange')

        self.frame = copy

# Find homography mapping in all frames
# Run operations on all frames to find the points needed for homography
bball_frames=[]
for i,frame in enumerate(frame_list[:-1]):
    bball_frame= BBall_Frame(frame)
    bball_frame.find_court_points()
    bball_frame.find_backboard_points()
    bball_frames+=[bball_frame]
```

5.2 Ball Tracker

```
#Ball Tracker

orangeLower1 = (150 ,20, 90)
orangeUpper1 = (175,160,160)
orangeLower2 = (1,20, 90)
```

```
orangeUpper2 = (17,160,160)

# if a video path was not supplied, grab the reference
vs = cv2.VideoCapture('POCvid3.mp4')
# allow the camera or video file to warm up
time.sleep(2.0)

# initialize these lists
mask_list     = []
frame_list    = []
mask_dif_list = []
# First frame
# grab the current frame
frame = vs.read()

# handle the frame from VideoCapture or VideoStream
frame = frame[1]


# set up the ROI for tracking
# setup initial location of window (roi _x,roi_y are coordinates of top left corner)
roi_x,roi_y,w,h = (1406,387,37,37) # simply hardcoded the values
```

```
# make list of ball locations (in pixels) and add initial value
ball_loc_2d    = [(roi_x ,roi_y )]

roi_cent_y ,roi_cent_x = (int(roi_y+0.5*w),int(roi_x+0.5*w))
roi                    = frame[roi_y:roi_y+h,roi_x:roi_x+w,:]

# initial roi (size 145X150)
cv2.rectangle(frame,(int(roi_x -145/2),int(roi_y -150/2)),(int(roi_x+145/2),int(roi_y+150/2)),(255,0,

hsv_roi                = cv2.cvtColor(roi, cv2.COLOR_BGR2HSV)
roi_mask1              = cv2.inRange(hsv_roi, orangeLower1,orangeUpper1)
roi_mask2              = cv2.inRange(hsv_roi, orangeLower2,orangeUpper2)
roi_mask               = cv2.bitwise_or(roi_mask1,roi_mask2)

# This is the balls color histogram, the distance from that will be the key in determining the next
roi_hist = cv2.calcHist([hsv_roi],[0],roi_mask,[180],[0,180])
cv2.normalize(roi_hist,roi_hist,0,255,cv2.NORM_MINMAX)

# construct a mask for the color "orange" (HSV), then perform
# a series of dilations and erosions to remove any small
# blobs left in the mask
hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
mask1 = cv2.inRange(hsv, orangeLower1,orangeUpper1)
```

```
mask2 = cv2.inRange(hsv, orangeLower2,orangeUpper2)
mask  = cv2.bitwise_or(mask1,mask2)

# Add to lists (for difference purposes)
frame_list += [frame]
mask_list  += [mask]

# keep looping till end of video
while True:

    if frame is None:
      break
   # grab the current frame
   frame = vs.read()

   # handle the frame from VideoCapture or VideoStream
   frame = frame[1]

#  convert to hsv
   hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)


   # construct a mask for the  ball color, then perform
```

```
    # a series of dilations and erosions to remove any small
    # blobs left in the mask
    mask1 = cv2.inRange(hsv, orangeLower1,orangeUpper1)
    mask2 = cv2.inRange(hsv, orangeLower2,orangeUpper2)
    mask  = cv2.bitwise_or(mask1,mask2)

#  add to lists
    frame_list     += [frame]
    mask_list      += [mask]
#   get the difference between frames and masks
    if len(mask_list)>1:
#       We want the pixels that weren't orange that are now orange
        mask_dif         = cv2.bitwise_and(mask_list[-1],cv2.bitwise_not(mask_list[-2]))
        mask_dif         = cv2.erode(mask_dif,None,iterations=2)
        mask_dif         = cv2.dilate(mask_dif,None,iterations=2)
        mask_dif_list += [mask_dif]

#       Define search region edges:
    search_reg_y = (max(roi_cent_y-100,0),min(roi_cent_y+45,int(frame.shape[0]/2)))
    search_reg_x = (max(roi_cent_x-100,0),min(roi_cent_x+50,frame.shape[1]-1))


    roi_edges = np.array([[search_reg_x[0],search_reg_y[0]],[search_reg_x[0],search_reg_y[1]],\
                              [search_reg_x[1],search_reg_y[1]],[search_reg_x[1],search_reg_y[0]],np.int3
    cv2.polylines(frame,[roi_edges],True,(255,0,0),3)
#   cv2.polylines(mask_dif,[roi_edges],True,(255,0,0),3)

    # find contours in the mask  that are LEFT OF THE BALL AND UPPER HALF OF SCREEN and initialize th
#   define search region and then n
    # (x, y) center of the ball
    cnts = cv2.findContours(mask_dif.copy()[search_reg_y[0]:search_reg_y[1],search_reg_x[0]:search_re
       cv2.CHAIN_APPROX_SIMPLE)
    cnts = imutils.grab_contours(cnts)
#   filter small contours out and add  search regions x,y to get cooridnates of contours in the ful
    cnts = [c+(search_reg_x[0],search_reg_y[0]) for c in cnts if c.shape[0]>2]
#   print(cnts)
    center = None

    # only proceed if at least one contour was found
    if len(cnts) > 0:
#       get the mean hue of each contour
        cont_mean_hues = np.array([hsv[cont[:,:,1],cont[:,:,0]].mean(0)[0][0] for cont in cnts])
#       get the contour with thehue that is closest to 3 (the basketballs hue in first frame)
        most_orange_idx = ((cont_mean_hues-3)**2).argmin()
        # find the contour whose distance is shortest from roi_cent
        # contour centers (with more than 2 pixels)
```

```
  cont_cents = np.array([cont.mean(axis=0) for cont in cnts])

  # update roi center

  roi_cent_x,roi_cent_y = np.array(np.round(cont_cents[most_orange_idx][:: -1]),dtype=np.int32)[0]
  c = cnts[most_orange_idx]
  ((x, y), radius) = cv2.minEnclosingCircle(c)
  M = cv2.moments(c)
  center = (int(M["m10"] / M["m00"]), int(M["m01"] / M["m00"]))

  # only proceed if the radius meets a minimum size
  if radius > 0:
    # draw the circle and centroid on the frame,
    # then update the list of tracked points
    cv2.circle(frame, (int(x), int(y)), 15,
      (0, 255, 0), 2)
# update thelocation list
ball_loc_2d  +=[center]

# loop over the set of tracked points
for i in range(1, len(ball_loc_2d)):
  # if either of the tracked points are None, ignore
  if ball_loc_2d[i - 1] is None or ball_loc_2d[i] is None:
```

```
            continue

        # otherwise, compute the thickness of the line and
        # draw the connecting lines
        thickness = int(np.sqrt(10/ float(i + 1)) * 2.5)
        cv2.line(frame, ball_loc_2d[i - 1], ball_loc_2d[i], (0, 0, 255), thickness)

cv2.destroyAllWindows()
```

5.3 Trajectory Extraction

```
# Make empty list for ball's 3D location
ball_3d_location = []
# Fixed things
size          = frame_list[0].shape
focal_length = size[1]
center        = (size[1]/2, size[0]/2)
camera_matrix= np.array(
                        [[focal_length, 0, center[0]],
                        [0, focal_length, center[1]],
                        [0, 0, 1]], dtype = "double"
                        )
dist_coeffs = np.zeros((4,1)) # Assuming no lens distortion
```

```
# Function for projecting the image plane on the court plane
def courtProject(im_point):
    z         = 0.0
    camMat    = np.asarray(camera_matrix)
    invCam    = np.linalg.inv(camMat)
    invRot    = np.linalg.inv(rotMat)

    uvPoint = np.ones(shape=(3, 1))
    # Image point
    uvPoint[0,0] = im_point[0]
    uvPoint[1,0] = im_point[1]

    temp          = np.matmul(invRot.dot(invCam), uvPoint)
    temp2         = invRot.dot(translation_vector)


    s = (z+temp2[2,0])/ temp[2, 0]
    #Court Model Point
    cmPoint = invRot.dot((np.matmul(s*invCam, uvPoint) - translation_vector))
    cmPoint[2] = z

    return cmPoint
```

```
# After using pin−pointing the point on the court where the shot was released and converting it usi
shot_loc_3d= (676.4,−175.6,0)
# The tracked points coordinates in the "Real World"
model_points = np.array([
(−159.96,1524/2,0),           #Baseline−sideline
(−159.96,−244,0),             #Paint−sideline
(419.04,−244,0),              #Paint−FT
(419.04,1524/2,0)]            #Sideline−FT
    ,dtype=np.float32
)
for i,frame in enumerate(bball_frames):
  f           =frame
  #This array has the pixel coordinates of the court & backboard points
  image_points  =np.array([f.baseline_sideline,
  f.paint_sideline,
  f.paint_ft,
  f.sideline_ft]
                        ,dtype=np.float32)

  (success, rotation_vector, translation_vector) = cv2.solvePnP(model_points, image_points, camera_

  rotMat, _ = cv2.Rodrigues(rotation_vector)
```

```
if i in [9,17]:
  (left_backboard_2d, jacobian)  = cv2.projectPoints(np.array([(-37.96, -182.9/2, 0.0)]), rotatio
  (right_backboard_2d, jacobian) = cv2.projectPoints(np.array([(-37.96, 182.9/2, 0.0)]), rotation
  (mid_basket_2d, jacobian)      = cv2.projectPoints(np.array([(0.0,0.0,0.0)]), rotation_vector,
  (shot_loc_2d, jacobian)        = cv2.projectPoints(np.array([shot_loc_3d]), rotation_vector, tr
  shot_loc_u,shot_loc_v          = shot_loc_2d[0][0][0],shot_loc_2d[0][0][1]
  shot_plane_bb_proj_int         = find_intersect(l1p1=(mid_basket_2d[0][0][0],mid_basket_2d[0][0
                  l2p1=(left_backboard_2d[0][0][0],left_backboard_2d[0][0][1]),
             l2p2=(right_backboard_2d[0][0][0],right_backboard_2d[0][0][1]))
  # finding the point on backboard (NOT THE PROJ) where the intersect happens

  backboard_intercept            = find_intersect(l1p1=(shot_plane_bb_proj_int[0],0),l1p2=shot_pl
               l2p1=f.top_left_backboard,l2p2=f.top_right_backboard,l1vert=True)
  # The difference in pixels from the shot plane and the backboard
  dif1                           = shot_plane_bb_proj_int[1]-backboard_intercept[1]
  # The intercept between the ball and the shot-plane
  ball_shotplane_int             = find_intersect(l1p1= ball_loc_2d[i],l1p2= (ball_loc_2d[i][0],0)
            l2p1=(mid_basket_2d[0][0][0],mid_basket_2d[0][0][1]),l2p2=(shot_loc_u,shot_loc_v),l1
  # get the x,y of the ball in flight
  ball_x,ball_y                  = courtProject(ball_shotplane_int)[:,0][:-1]

  #height of ball (in pixels)
  dif2                           = ball_shotplane_int[1]- ball_loc_2d[i][1]
```

```
    # Real ball height (396.32 is the height of the backboard)
    ball_z                          = (dif2/dif1)*396.32

    ball_3d_location               += [np.array([ball_x, ball_y, ball_z])]

ball_3d_location = np.array(ball_3d_location)

# Add middle of basket position to reflect made shot
ball_3d_location                   = np.append(ball_3d_location,[[0,0,304.8]],axis=0)
# Reduce x,y to distance from basket
dist_from_basket                   = (np.sqrt(ball_3d_location[:,0]**2+ball_3d_location[:,1]**2)*np.
ball_3d_location                   = np.concatenate((ball_3d_location,dist_from_basket),axis=1)

# Fitting the trajectory of the shot
x,y,z,d = ball_3d_location.T
fitdz   = np.polyfit(d, z, 2)
fitarc  = np.poly1d(fitdz)

shot_dist     = np.sqrt((np.array(shot_loc_3d)**2).sum())
dist_ar       = np.linspace(0,shot_dist,len(frame_list))[::-1]
mpl.style.use('seaborn')
plt.figure(figsize=(15,15))
plt.scatter(dist_ar, fitarc(dist_ar), s=470, alpha=0.7, edgecolors='black', color='darkorange')
plt.scatter(d, z, s=470)
# Adding basket and backboard to plot
plt.plot([-37.96, -37.96+15.1], [304.8, 304.8], 'k-', color = 'orangered', linewidth=2)
plt.plot([-37.96+15.1, -37.96+15.1+45.72], [304.8, 304.8], 'k-', color = 'orangered', linewidth=4)
plt.plot([-37.96, -37.96], [396.32-121.9, 396.32], 'k-', color = 'black', linewidth=8, alpha=0.7)
plt.xlabel('Distance From Mid-Basket Point (cm)', fontsize=20)
plt.ylabel('Height (cm)', fontsize=20)
plt.xticks(fontsize=16)
plt.yticks(fontsize=16)

plt.ylim(0, 520)
# Times by 30 to adjust for the frame rate then divide by 100 to adjust for meters instead of cm

# Find Shot Parameters
v0_x, v0_y = (dist_ar[1]-dist_ar[0])*30/100, (fitarc(dist_ar[1])-fitarc(dist_ar[0]))*30/100
v0         = np.sqrt(v0_x**2+v0_y**2)
theta      = np.rad2deg(np.arctan(v0_y/v0_x))
```

References

(2015*a*), ‘humanbenchmark’.
URL: *https://www.humanbenchmark.com/tests/reactiontime/statistics*

(2015*b*), ‘Opencv’.
URL: *https://www.pyimagesearch.com/2015/09/14/ball-tracking-with-opencv/*

(2017), ‘Basketballprotraining.com’.
URL: *https://www.basketballprotraining.com/single-post/2017/04/11/DEGREE-ELBOW-ANGLE*

(2018), ‘Basketball reference statistics database’.
URL: *http://www.basketball-reference.com*

(2019*a*), ‘Nba official website’.
URL: *http://nba.com*

(2019*b*), ‘Nba savant database’.
URL: *http://nbasavant.com*

Chantara, W., Mun, J.-H., Shin, D.-W. & Ho, Y.-S. (2015), ‘Object tracking using adaptive template matching’, *IEIE Transactions on Smart Processing and Computing* **4**, 1–9.

Chen, H.-T., Tien, M.-C., Chen, Y.-W., Tsai, W.-J. & Lee, S.-Y. (2009), ‘Physics-based ball tracking and 3d trajectory reconstruction with applications to shooting location estimation in basketball video’, *J. Vis. Comun. Image Represent.* **20**(3), 204–216.
URL: *http://dx.doi.org/10.1016/j.jvcir.2008.11.008*

Fan, H. & Ling, H. (2017), Sanet: Structure-aware network for visual tracking, *in* ‘Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops’, pp. 42–49.

Farin, D., Han, J. & de With, P. H. (2005), Fast camera calibration for the analysis of sport sequences, *in* ‘2005 IEEE International Conference on Multimedia and Expo’, IEEE, pp. 4–pp.

Feng, X., Mei, W. & Hu, D. (2016), A review of visual tracking with deep learning, *in* '2016 2nd International Conference on Artificial Intelligence and Industrial Engineering (AIIE 2016)', Atlantis Press.

Fontanella, J. (2006), *The Physics of Basketball.*

Fu, T.-S., Chen, H.-T., Chou, C.-L., Tsai, W.-J. & Lee, S.-Y. (2011), Screen-strategy analysis in broadcast basketball video using player tracking, *in* '2011 Visual Communications and Image Processing (VCIP)', IEEE, pp. 1–4.

Kirk Goldsberry (2019), 'Steph curry is still completely unfair'.
URL: $http://www.espn.co.uk/nba/story/_/id/25771897/steph - curry - unleashing - impossible - range$

Ladikos, A., Benhimane, S. & Navab, N. (2007), A real-time tracking system combining template-based and feature-based approaches., *in* 'VISAPP (2)', Citeseer, pp. 325–332.

Liu, Y., Liang, D., Huang, Q. & Gao, W. (2006), 'Extracting 3d information from broadcast soccer video', *Image Vision Comput.* **24**, 1146–1162.

Ric Bucher (2015), 'Is stephen curry the best shooter ever? yes, say many of nba's all-time marksmen'.
URL: *https://bleacherreport.com/articles/2482473-is-stephen-curry-the-best-shooter-ever-yes-say-many-of-nbas-all-time-marksmen*

Savvas Tjortjoglou (2015), 'How to create nba shot charts in python'.
URL: *http://savvastjortjoglou.com/nba-shot-sharts.html*

Scott Davis (2018), 'How stephen curry became the best shooter the nba has ever seen'.
URL: *https://www.businessinsider.com/stephen-curry-best-shooter-in-nba-2016-6?r=USIR=T*

Wang, N., Li, S., Gupta, A. & Yeung, D. (2015), 'Transferring rich feature hierarchies for robust visual tracking', *CoRR* **abs/1501.04587**.
URL: *http://arxiv.org/abs/1501.04587*

Wang, N. & Yeung, D.-Y. (2013), Learning a deep compact image representation for visual tracking, *in* 'Advances in neural information processing systems', pp. 809–817.

Wong, R. Y. & Hall, E. L. (1978), 'Sequential hierarchical scene matching', *IEEE Transactions on Computers* (4), 359–366.

Yuen, H., Princen, J., Illingworth, J. & Kittler, J. (1990), 'Comparative study of hough transform methods for circle finding', *Image and vision computing* **8**(1), 71–77.

Zhang, D., Maei, H., Wang, X. & Wang, Y.-F. (2017), 'Deep reinforcement learning for visual object tracking in videos', *arXiv preprint arXiv:1701.08936* .