INTRODUCTION TO COMPUTER VISION

BASKETBALL TRACKER AND GAME ANALYZER

December 12, 2017

Dennis Te & Michael Cheng

Colorado School of Mines

INTRODUCTION

As engineering students that love the game of basketball, the technology aspect of the basketball game viewing experience can also be an interesting aspect other than the numbers on the scoreboard itself. Needless to say, the idea of utilizing skills learned from Computer Vision to make the game more enjoyable is an interesting topic for research and development.

During one of the 2015-2016 NBA season games, Sacramento Kings point guard Rajon Rondo made his signature "fake behind the back pass" move while driving into the paint against the Utah Jazz. Fans have seen Rondo make this move numerous times, but this time was a bit more impressive: Rondo faked out the cameraman on the baseline. The baseline camera usually captures the best views of spectacular plays near the basket, yet for this instance, fans were not able to witness the entirety of Rondo's genius. This lead to our initial inspiration for this project, ball tracking.

Basketball tracking during an NBA games is a difficult task, as their are many variables to consider. During an NBA game, the basketball can become obstructed from the camera's point of view. In addition, the basketball can move very fast, making it difficult to track between subsequent frames. For this reason, the project was shifted towards a shoot-around, which has more controlled variables than an action-packed game.

The basic objective of the project is to successfully track the basketball in a shoot-around footage at all times, displaying a square around the basketball and following it. If the tracking algorithm works robustly, it can be further utilized to perform more advanced analysis of the video. Examples of these more advanced objectives include auto-detection of the ball and rim, as well as a shot make/miss detection method.

For this assignment, various assumptions was made. The shoot-around videos used did not have too many similar colors as the desired objects of tracking. In addition, the ball was assumed to be of a certain size and color, which corresponds to the size of the template. Additionally, all of the videos tested are from the same setting, allowing for control over environmental variables, such as other players and other balls. While the code can be adjusted for other videos, it currently does not work well without other templates for the ball and hoop. However, if the code is changed so that the ball and rim are selected, (which is available within the code), then the mean-shift tracking algorithm was tested to work decently.

MATLAB was chosen as the primary coding platform for tracking algorithms over C++ along with the OpenCV packages. In comparison to OpenCV, MATLAB did not have any prebuilt mean-shift tracking functions, which forced thorough understanding of concepts used in mean-shift tracking. Additionally, the mean-shift tracking method can be modified and adjusted based on the needs of the researcher. Lastly, a portion of this code was adapted from a previous project that was available via the MATLAB exchange files program. Code was rewritten and changed significantly, but many of the mathematical functions are similar [1].

PREVIOUS WORK

The use of computer vision techniques for sports is not a novel concept. Multiple approaches have been used to track a basketball, or other similar objects (i.e. a hockey puck) during sporting events.

Sensor Based Tracking

While sensor based tracking is directly related to computer vision, it is important to understand some of the initial approaches taken to track and enhance objects during games. In the 1990s, a series of infrared LEDs were installed into a hockey puck [6], which was then tracked via an infrared camera. The infrared camera then overlaid its information with the RGB camera, meanwhile implementing a glowing effect around the puck for better visualization as well as added camera effects. While the puck was moving at a slower rate, a blue glow highlighted the location of the puck on the screen. When the puck reached speeds of above 70 mph, the blue glow turned into a red glow, which created the effect of a comet tail on the puck. Ultimately, this turned out to be a technological failure with the fans. The blue and red glow were not aesthetically pleasing, even though it occasionally enhanced the ability to track and see the puck during hockey games. The technology was scrapped, never to be brought in again.

With respect to basketball, a startup, ShotTracker, recently received funding to bring real-time analytics to basketball. The startup uses sensors attached to players'shoelaces and inside the basketball which allows for real-time tracking of both the ball and the players. The use of sensors gives a much more robust data set than the computer vision, as tracked objects would not be obstructed from the camera's field of view. In addition, changes in camera lighting, basketball colour, and other vision-related factors are ignored with the use of sensors.

Use of Multiple Cameras

For tracking of a full-court basketball game, most modern technology incorporates the use of multiple camera systems and sensors that allow for the real time tracking and accumulation of advanced analytics of a game [5]. The Hawk-eye [4] has been used in professional sports for automatic detection of the players and the ball. With the use of multiple cameras, desired objects can be mapped onto a 3D space, which allows for accurate locating of various objects. Two examples of this are in baseball and tennis. In baseball, the Hawk-eye system can detect the location of the pitch with respect to the strike-zone. In tennis, the Hawk-eye system is used to track the landing location of serves and hits, allowing for referees to accurately visualize whether or not a fault was hit.

Single Camera System

For this student project, however, the budget and time did not allow for the incorporation of multiple cameras into the system. Previous work involving computer vision and basketball incorporates the use of mean-shift tracking, Hough lines, as well as machine learning to track players and the ball in a video [2]. Mean-shift tracking is a well studied algorithm, with various improvements that has been made throughout the year. Subsequently, mean-shift tracking was chosen as the basis for the method moving forward to perform ball-tracking.

TECHNIQUES AND METHODS

Mean-Shift Tracking

Mean-shift tracking [7] is a well-known algorithm that is used for tracking an object. In summary, the mean-shift algorithm observes the colors of a selection window and then compares them to a template image. After comparing the two, it shifts the selection window towards the area that it believes the object has moved. It repeats iteratively until either a certain similarity value or the max number of iterations has been reached.

Histogram of Colors

Because the mean-shift algorithm is dependent on being able to compare colors from histograms, it is necessary to determine the best way to sort the colors into a singular histogram for processing. There is some difficulty in this, as there are 16,777,216 possible colors given the standard 256 slices of RGB (red, green, blue). It would be computationally inefficient and non-functional if every color of the images were sorted into a histogram of over 16 million bins.

To tackle this, a minimum variance quantization [3] algorithm is used. There are other methods that could be used such as color map mapping and uniform quantization, but this

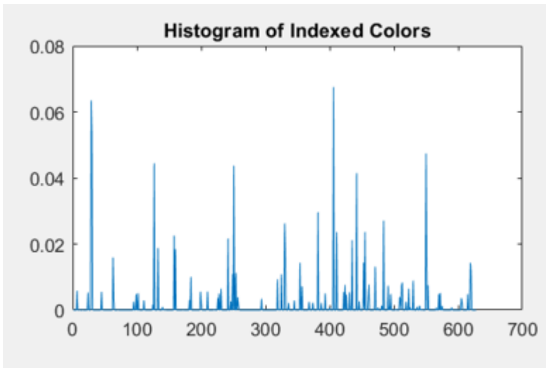

Figure 1: This is an example of the histogram of indexed colors. Note that the length of the histogram is much shorter than the possible number of colors available in a standard RGB color scheme.

was the most readily available. This method works by associating pixels into groups based on the variance between each pixel. For example, let's propose an image that was divided into three columns with three different colors (i.e. red, green, and blue) with very slight variations within each of their respective colors. All of the colors that seem red to the human eye are actually within 10 pixel colors of R256G0B0, with the same that can be said for the green and blue colors. A fully encompassing histogram would have over 16 million bins, with only around 30 bins actually having a value in them. With minimum variance quantization, the length of the color map, and therefore length of the histogram, can be cut down into three bins. Although there can be some data loss, this is computationally effective and ultimately does not have a negative effect on the mean-shift algorithm.

Within the mean-shift tracking algorithm, the first frame of the movie is processed using minimum variance quantization to create a color map of the image. Additionally, the length of the color map, LMap, is stored. Afterwards, a histogram, q, is created for the template, with the number of bins inside the histogram being equal to LMap. This allows the algorithm to reduce the size of the histogram while maintaining most of the information in the image. The histogram is created with the function below:

Histogram Creation

```
%% Function for creating a histogram from an image
```

```
% T = image input
% LMap = length of the color map
% kernel = kernel
% H = Height of the template and the image
% W = Width of the template and the image
% hist = array with the number of bins the same as LMap. Colors are put
% into each of the bins to create a histogram of the indexed colors

function hist = createHist(T, LMap, kernel, H,W)
hist = zeros(LMap,1);
for i = 1:W
    for j = 1:H
        hist(T(j,i)+1) = hist(T(j,i)+1) + kernel(j,i);
    end
end
% ---Normalize the Histogram ----
hist = hist./sum(sum(kernel));
end
```

This function observes each point inside the specified template, and returns the indexed color value for that point. It follows these procedures for every point, adding the kernel value at each point to the histogram of indexed colors. Afterwards, it normalizes the histogram by dividing by the sum of the kernel values.

The Kernel

A normal (Gaussian) kernel was used for this assignment. Other kernels were explored, but the normal kernel gave the best results for tracking initially. Other kernels that have been identified and used in the past are the Epanechnikov Kernel, a triangular kernel, and a uniform kernel.

The kernel is essentially a weighting function that allows the selection window to be weighted based on the x and y position of the pixel. In the normal kernel, the center pixel has the most weight, with decreasing weights for pixels that radiate outwards from the center. This means that when the template image and the selection window are compared, the center pixel has the highest weight on whether or not the images are deemed to be similar in colors.

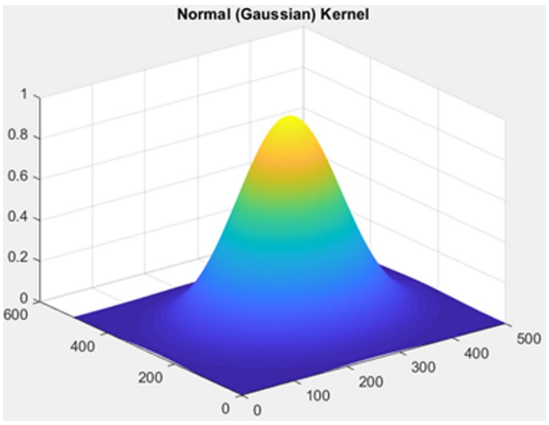

Figure 2: This is the Normal (Gaussian) kernel used in the mean-shift tracking algorithm. In simple terms, this is how the weight of each pixel is determined. Towards the center, pixels are weighted higher, whereas pixels on the edge are weighted low.

The Similarity Function

In order to compare the template image to the selection window, a function that uses both the histograms and kernel aforementioned to calculate two important values: a similarity value f, and an array of values, called the weight mask, w. The similarity value helps to define how close the template is to the selection window. If the similarity value is high, then the two images are similar. The weight mask will be used later in determining how to shift the selection window.

The function is defined as below:

Similarity Function

```
%% compareBoxes(hist, hist2, T2, kernel, H,W)
%This function compares the similarity between two histograms
% hist = Histogram from the template image
% hist2 = Histogram from the selection window
% T2 = Selection Window
% kernel = the kernel being used. In this case, a normal kernel
% H = Height of the template and the selection window
% W = Width of the template and the selection window
% f = similarity value
% w = weight array

function [f, w] = compareBoxes(hist,hist2,T2,kernel,H,W)
```

```
w = zeros(H,W);
f = 0;
for i = 1:H
    for j = 1:W
        w(i,j) = sqrt(hist(T2(i,j)+1) / hist2(T2(i,j)+1));
        f = f+w(i,j)*kernel(i,j);
    end
end

f = f/(H*W);
```

hist refers to the histogram of the template image, and hist2 refers to the histogram of the selection window. T2(i,j)+1 refers to the indexed color at the point [x,y] in the image.

A better explanation for the similarity function is provided here. The similarity function looks at each pixel from the template image, corresponding it and comparing it with the pixel from the selection window. It retrieves an indexed color for both, and then looks at the histogram for both images at that indexed color value. It then calculates the weight mask at the specific point. The similarity function sums the entire weight mask to retrieve a single number that can be used to easily understand how similar the images are.

Determining movement of the selection window

Finally, the selection window must be shifted after finding the similarity value is lower than a set threshold. A displacement vector is created, [dx, dy], which will be added to current location of the selection window [x0, y0], to create a new location [x1, y1]. The displacement vector is determined by the following function:

Finding the Displacement of the Selection Window

```
%% Finding the displacement for the selection window.
%This function finds the displacement for the selection window.
% w = weighted array
% gx = gradient of the kernel in the x direction
% gy = gradient of the kernel in the y direction
% H = Height of the template and the selection window
% W = Width of the template and the selection window

```

```
function [x,y] = findDisplacement(w, gx, gy, H, W)
num_x = 0;
num_y = 0;
den = 0;
for i = 1:H
    for j = 1:W
        num_x = num_x + i*w(i,j)*gx(i,j);
        num_y = num_y + j*w(i,j)*gy(i,j);
        den = den + w(i,j)*norm([gx(i,j), gy(i,j)]);
    end
end
%% Calculate how much to displace the new box by
if den ~= 0
    dx = round(num_x/den);
    dy = round(num_y/den);
    y = y+dy;
    x = x+dx;
end
```

In this function, the weight mask is multiplied by the gradient (gx, gy) in the x and y direction at each point. The gradient is the partial derivative of the kernel used in the x and y direction, respectively. The product of the weight mask and gradient is summed up across all of the points, and then divided by a denominator value, den, which is the sum of the products of the weight mask and magnitude of the gradients across each pixel.

Expansion Algorithm

After realizing the mean-shift tracking doesn't work well for fast moving objects and a slow frame rate, it was necessary to solve this issue. In order to do this, the window around which the object was detected before losing track was expanded to include eight other same size windows. (Figure 3). The algorithm “pretends” that the window to continue mean-shift tracking is actually in one of eight windows surrounding the original location; if the cause of losing track is due to the ball moving too fast or a low frame rate, the ball is likely to show up in nearby regions of the original window. The eight different windows are checked for their similarity values, which are stored. The algorithm then identifies the window with the

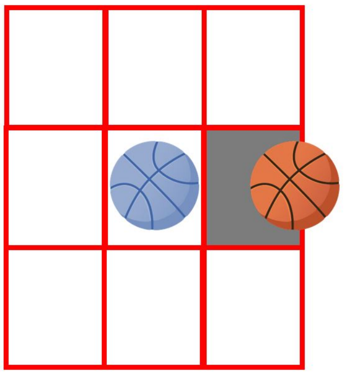

Figure 3: This diagram shows how the window is expanded to try to track the ball if it moves away from the original location too quickly. The ball is found in the right selection window and the similarity value is detected to be the highest. The algorithm continues to use mean-shift tracking in the newly found window. .

highest similarity value, replace the original with that window, and performs tracking with the window that had been selected for its highest similarity value.

Originally, the algorithm checked each of the eight windows one by one starting from the top left corner. If the similarity value of any window exceeded a threshold value, it was identified as the replacement window. However, if the expansion algorithm is triggered, the algorithm would inadvertently detect another object that was above the threshold. For example, the player's head would oftentimes be perfectly bounded within the first top-left expansion window; the algorithm would generate a similarity value that is above the threshold, immediately identifying the player's head as the ball, and replace the original with the top-left window. When this error occurs, the window will continue tracking the player's head for the rest of the video. Therefore, the algorithm was improved by checking all of the windows and using the window with the highest similarity value. The inspiration of improving this method was motivated by the problems faced while tracking around the rim, which is discussed in the following topic.

Robustness of tracking around the rim area

Most rims share the same color as the basketball, orange. Tracking failures had often occurred often when the ball was shot and traveling/bouncing near the rim because the rim and the ball were similar in color. In addition, as the ball travels in and around the rim, motion blur and object obstruction make it harder to detect the ball itself.

To solve this issue, the algorithm needed to take advanced actions when the ball location is near the rim. The rim was detected at the start of the video, along with the basketball. The algorithm then calculates the distance between the ball and the rim; if this distance becomes smaller than a threshold value, then the ball is marked in a state that is "near the rim".

While near the rim, the selection window expands to nine windows in total, including the original, for every single frame once the near rim condition is triggered. The algorithm will constantly check the similarity value to the template for each of the nine windows during every frame and locates the window with the highest score. With this, the algorithm is tracking the orange object that is the most similar to the original template, allowing for accurate tracking of the basketball while tracking the detection of the rim.

Automatic Detection of Ball and Rim

As we tested the algorithms over and over again, cropping the basketball and the rim before the tracking beings became extremely redundant and time consuming. It became of interest to attempt implementing a method for automatically detecting the ball and the rim prior to the start of tracking. This would make the results of the algorithm more consistent on a trial-to-trial basis, allowing the researchers to more easily understand any errors in their methods.

Initially, ball detection was approached by utilizing Hough based functions, such as the "imfindcircle" function that we implemented in one of the homeworks. However, after numerous attempts at finding a tolerable sensitivity value and threshold, it was concluded that it is nearly impossible to locate the ball with Hough transform due to the noise of the video. The Hough transform technique does not work well with motion blur, as the it is dependent on being able to track edges. With motion blur, the edges are not detected properly.

Subsequently, mean-shift tracking methods were used to identifying the ball. A moving window was used to cover all areas of the initial image. Similarity values were generated and stored for each window, comparing the selection window to a universal template of the ball or rim. Afterwards, the algorithm will find the window with the highest value, and the mean-shift algorithm will then maneuver the window to the targeted window. This method was implemented to automatically detect the ball and rim.

This method is not as “elegant” as initially anticipated, though it's performance is nearly perfect and the concept is straightforward and easy to implement. Hough transform based algorithms and functions can have advantages over this method in terms of processing speed. The limitations of this approach are discussed later.

Make/Miss Detection

As mentioned, a function that returned a notifying value whenever the location of the current detection window is within a close distance with the rim, signaling the “near rim” algorithm. This function was further utilized as a trigger to start the checking the ball for specific movements that would inform the outcome of the shot.

In order to detect a missed shot, the ball was tracked with its position relative to the rim. If the ball entered the “near rim” state, and subsequently left the “near rim” state via the side of the rim or the top of the rim, then the shot was detected as a miss. The x and y location of the selection window was compared to the location of the rim, outputting valuable information on whether or not the shot went in.

If the ball happens to appear at a location that is lower than the rim during the “near rim” condition, the shot is marked as a make. Regardless if the shot was a bank shot or a “swish”, the ball will go through the rim and eventually end up below the rim. Therefore, the y distance from the rim to the ball was chosen as a prime candidate for the make/miss detection algorithm.

This make/miss detection algorithm works well with the footage used, but it is not perfect. The algorithm might incorrectly classify the shot depending on the ball's movements. The limitations of the make/miss detection are discussed later.

EXPERIMENTS AND RESULTS

The best way to demonstrate the results of this experiment is via a video example. Below are three links to three different videos that were created by the algorithm. In the videos, when the ball is being tracked with normal mean-shift tracking, the square is highlighted green. When the tracking enters the expansion algorithm, the square is highlighted red. The video pauses when a make/miss detection occurs. These videos do not demonstrate the ball and rim finding algorithm used prior to creating the video. It is recommended that all videos be played at 0.5 speed, as this is the closest to real-time movements.

Video 3 is the longest video of the three. During this, there are some false positives and false negatives detected with the make/miss algorithm. However, it definitely demonstrates the full robustness of the algorithm.

One of the accidental positives of using the mean-shift algorithm is that the tracking window will generally continue to falsely highlight the player when the player occludes the ball from the videos frame. While this is inaccurate, it continues to enter the expansion algorithm because of the low similarity value produced by the mistracked object. Once the ball appears into the frame again, it can be accurately re-tracked.

Video 1 https://www.youtube.com/watch?v=NuVcV0TQ3ZM

Video 2 https://www.youtube.com/watch?v=ZHJP0bh246Y

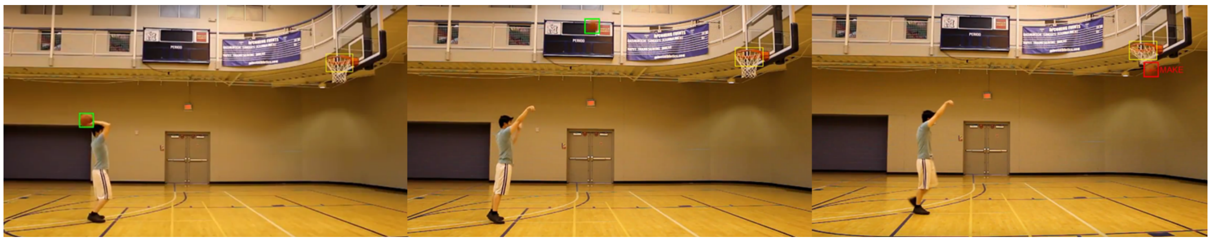

Figure 4: In this sequence, the ball is accurately tracked and a box is displayed around the tracking sequence. In addition, the shot is made and correctly detected as a make.

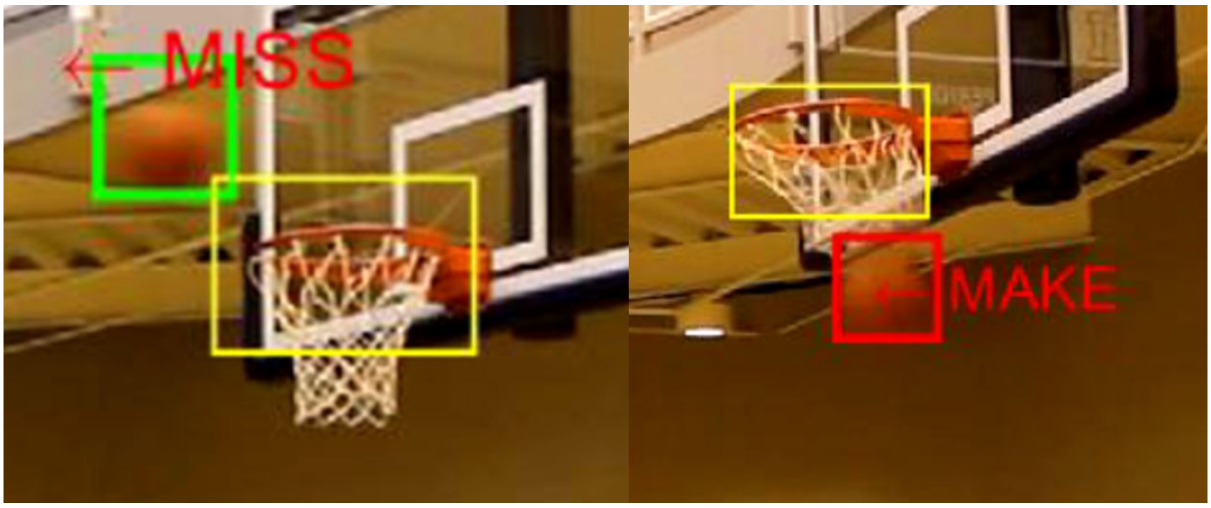

Figure 5: In this sequence, the ball is accurately tracked and a box is displayed around the tracking sequence. In addition, the shot is made and correctly detected as a make.

Video 3 https://youtu.be/NpZ6dnYymtI

Video with Working Code https://youtu.be/1mnbaWAGWjg

For ease of access, a time sequence of the video is shown in Figure 4.

For make miss detection, Figure 5 shows two examples of accurate detection of the make miss. In the frame that shows a missed shot, the ball moves too far to the left of the window of the rim. This triggers the algorithm to accurately detect that the shot was a miss. In the frame that shows a made shot, the ball moves below the window of the rim, which is then detected as a make.

Due to the nature of the assignment, there were not any meaningful available statistics that can be used to describe the code. While a table of false positives and negatives can be created for the make/miss detection, this table would not accurately describe the robustness of the algorithm.

DISCUSSIONS

Achievements

Ultimately, an algorithm was designed to robustly identify and track a basketball during a shoot-around. Additionally, the algorithm uses a simple method to employ shot detection. The program also automatically identifies the hoop within the frame. The algorithm can be easily adapted to change whether or not the ball and hoop should be identified manually or automatically.

Limitations

Automatic Identification of the ball and hoop

With the current algorithm, automatic identification of the ball and hoop works correctly for a set of videos that were recorded by the authors. The algorithm references a template for both of these. While this works well for given color of the ball and given color of the hoop, this is not the best means of identifying the ball and the hoop. Ideally, a library of images for different types of hoops and basketballs would have been used to classify and identify the two objects. Without a different approach, if the ball were to be changed for a different color ball, the current automatic detection would not work (same can be said for the hoop).

Additionally, the current method used for identifying the ball and the hoop is computationally inefficient. Using a moving window to identify both will oftentimes take upwards of thirty seconds. While it is robust given the set of videos used, this is a very brute-force approach to finding the rim and ball.

Similar Colors

One of the big issues with most mean-shift tracking algorithms is that it uses colors to identify the object. While this works in many-cases, it can become difficult if there are too many similar colors as the template. It is very possible for the algorithm to become distracted by another similarly colored object in the image. Within a shootaround, this can occur quite often. A basketball is not a unique color, and can oftentimes be confused with other brown or red objects. Within the test videos, it was seen that the the basketball is oftentimes confused with either the player's head or the player's hand. Unique colors were worn by the researcher to try to counter this issue, but ultimately this is a fundamental issue with mean-shift tracking. It might be addressed by looking at colors in different spaces, such as HSV, but that would also increase computational time.

Out of bounds

Obviously, mean shift tracking does not work if the object is not within the frame of the screen. The current algorithm does not understand that the ball has left the screen and that it should try to track ball coming into the screen. While this can be implemented using a similar method as ball identification, it is computationally draining.

False positives and negatives on make/miss detection

As described, the make/miss detection uses the location of the hoop and the location of the ball to process data. However, this is a not a robust way to perform shot make/miss detection. There are several cases where the algorithm will detect false positives (detecting a miss as a make), with the most obvious case being an airball. The algorithm would detect the shot, and then wait for the ball to either bounce up or bounce away from the rim. In an airball, neither of the bounces occur. Therefore, the shot is tracked as a make, even though the player did not make the shot.

There are also ways for false negatives to be detected (i.e. detecting a make as a miss). For example, the algorithm classifies the shot as a miss when it goes too far above the rim. It is possible that the ball can bounce above and around the rim before falling through the net. With this in mind, there would be two shots detected, labeled as a miss and a make.

Possible Future Work

Identifying the court (attempted)

One of the stretch goals of this project was to attempt to identify the basketball court. This is of interest because it allows to identify the position of the shot after it was taken and display it in an orthographic map. Several attempts were taken to identify the court, specifically the circular pattern that is around the free throw line.

The circular pattern is one of the characteristic features of a basketball court, and was identified as a unique feature that could assist in the identification of the location of the player on the court. At first, a circular Hough transform with edge detection was used to try to identify the object. However, there were several issues with this. The circle was hard to identify in grayscale space. Several filtering techniques, such as contrast adjustment and morphological opening and closing of the image, were used to try to bring out the quality of the line in the image, but they were unsuccessful. However, a larger problem became clear.

With the perspective of the video, the circular pattern on the court was not actually a circle. While there was hope that parts of the ellipse could be identified using the circular Hough transform, it proved unsuccessful. Afterwards, a parabola detection algorithm using Hough transforms was implemented onto the image. It was believed that the ellipse could be estimated using a parabola. Ultimately though, this was unsuccessful.

Future work would involve using the vanishing point of the image to understand how the court is viewed with respect to the camera. The Hough transform can be used to detect lines that are in the image (i.e. the out-of-bounds lines on the basketball court and the edge of the backboard), which can then be used to understand how to rotate and skew the image so that the court is displayed in an orthographic view. From here, the circle can be detected using Hough transform and then remapped back onto the original image. This is a hypothetical way to tackle this issue, but it has not been tested fully.

Adaptive Mean Shift through Scale-Space

One of the larger limitations of the current implementation is the inability to understand and identify changes in the size of the tracked object. With a small object, this is critical because it allows the the identification of the object whether or not it comes close to the camera or moves far from the camera. There are a number of approaches that can be used to implement this: Bradski's CAMSHIFT tracker, Camaniciu's variable bandwidth methods, and Rasmussen and Hager's method. Lindeberg's theory uses a Laplacian of Gaussian (LOG) filter to approximate the Difference of Gaussian (DOG) in scale-space, as well as a 3D spatial kernel to identify the object's size and adapt the window size appropriately. It would be very interesting to implement this on the current program to see how it would improve robustness.

Player tracking

Player tracking (single as well as multiple) is very useful information that can be used to supplement the current available data with analytics. For example, if the player is tracked during a shootaround, it should be trivial to identify when the player has the ball. Additionally, it should be simple to understand when the player has shot the ball. This can ultimately lead to an array of important analytics that include: shot angle, shot location, rebounds, assists, shot placement, etc. This would all be valuable information to someone who would want to understand what their shot looks like and how they can improve their shot.

There are multiple well-identified methods to player tracking. One method for implementing this would be to utilize machine learning to correctly identified features and players on a player. A Support Vector Machine (SVM) classifier using Histogram of gradient feature calculation has previously been used to identify basketball players in a game. This would be one of the more robust methods for identifying players, since people vary in their size, shape, and color. After the player is identified, a combination of mean-shift tracking and machine learning can be used to continuously track the player.

Out of Bounds detection

As mentioned, the mean-shift tracking does not work if the tracked object moves out of bounds. While the object cannot be tracked if it is not found in the frame, it is possible to continue searching for the object until it comes back into the frame. For example, if the ball were to leave the frame, the edges of the frame can be checked to see if any of the moving windows detected had a similarity value that was high enough to be considered the desired object. However, this is computationally intensive and could present some issues if there is an object that is similarly colored to the ball on the edge of the frame.

CONCLUSION

Overall, the opportunity to understand and modify the mean-shift algorithm was very fulfilling. It allowed the researchers to explore a topic that was touched upon in the course, giving a more in-depth understanding of the topic. Ultimately, the mean-shift algorithm was improved and adapted to track the basketball more robustly, as well as calculate some analytics on the basketball 's movement.

Bibliography

[1] Bernardt, Sylvain. *Mean-Shift Video Tracking* Project on MATLAB File Exchange. Retrieved from https://www.mathworks.com/matlabcentral/fileexchange/35520-mean-shift-video-tracking

[2] Cheshire, Halaz, and Perin. *Player Tracking and Analysis of Basketball Plays* Class Project. 2015. https://web.stanford.edu/class/ee368/Project_Spring_1415/Reports/Cheshire_Halasz_Perin.pdf

[3] MathWorks. *Working with Different Screen Bit Depths* Chapter 13 of Image Processing Toolbox - For Use with MATLAB. Version 3. 2002. https://edoras.sdsu.edu/doc/matlab/pdf_doc/images/images_tb.pdf

[4] Tepper, Fitz. *ShotTracker raises 5M in seed funding from Magic Johnson and David Stern to bring real-time analytics to NBA teams* October 19, 2006. TechCrunch. https://techcrunch.com/2016/10/19/shottracker-raises-5m-in-seed-funding-from-magic-johnson-and-david-stern-to-bring-real-time-analytics-to-nba-teams/

[5] Thomas, Gade, Moeslund, Carr and Hilton. *Computer Vision for Sports: Current applications and research topics* Elsevier. Volume 159, June 2017, pp 3-18. https://www.sciencedirect.com/science/article/pii/S1077314217300711

[6] Potel, Michael. *The FoxTrax Hockey Puck Tracking System* IEEE Computer Graphics and Applications Volume 17 Issue 2. March-April 1997. https://pdfs.semanticscholar.org/e34f/aa0f9a8ba9e84296106ff7f473599e7a1214.pdf

[7] Collins, R. *Mean-shift Tracking* Powerpoint Slides. 2006. Retrieved from http://www.cse.psu.edu/~rtc12/CSE598G/introMeanShift.pdf

APPENDIX

Main Function

```
%% Mean Shift Tracking - main.
% This performs mean shift tracking on a video file for a basketball
% shoot-around.
% It automatically detects the basketball and rim.
% Tracks the location of the basketball
% Performs automatic make/miss detection.

%% Quick check to see if we need to load new variables
% In addition, check to see if video file exists. If the file exists,
    then
% there is no need to create a new file; instead, read the file. If it
% doesn't exist, then create a new video file.
newVideo = 0;
str = 'make_01';
if newVideo == 1
    fprintf('Using previous variables...');
    close all; clc;
elseif ~exist(strcat(str, '.mat'), 'file')
    clearvars -except str; clc; close all;
    newVideo = 0;

    fprintf('Reading Video file... '); tic;
    path = strcat('videos/',str,'.mp4');
    movie = VideoReader(path);
    movH = movie.Height; movW = movie.Width;
    nFrames = movie.NumberOfFrames;
    mov(1:nFrames) = struct('color', zeros(movH, movW, 3, 'uint8'), '
        colormap', []) ;
    for i = 1: nFrames
        mov(i).color = read(movie, i);
    end
    save(strcat(str,'.mat'), '-v7.3'); toc;
else
    clearvars -except str; close all; clc;
    fprintf ('Loading Video File... ');
    tic
    load(strcat(str,'.mat'));
    toc
end
```

```
warning('off', 'Images:initSize:adjustingMag');
startFrame = 2;
%% Select Patch

T= imread('ball.png');
T = imresize(T, [59, 59]);
[H,W,~] = size(T);

Trim = imread('rim.png');
Trim = imresize(Trim, [76,114]);
[rimH,rimW,~] = size(Trim);

%% Patch Decision
%If you want to select the basketball and rim, instead of using the
% predefined templates for the rim, then change selectPatch = 1;
selectPatch = 0;
if selectPatch == 1
    f1 = figure(1);
    imshow(mov(startFrame).color); title('Select an object to track ');
    fprintf('Please highlight an object to track \n');
    rect = getrect;

    x0 = round(rect(1)); y0 = round(rect(2));
    W = round(rect(3)); H = round(rect(4));
    T = mov(startFrame).color(y0:y0+H-1, x0:x0 + W - 1,:);
    imshow(T);
    rectangle('Position', [x0, y0, W, H], 'LineWidth', 2, 'EdgeColor',
        'g');
    close(f1);

    f1 = figure(1);
    imshow(mov(startFrame).color); title('Select the rim');
    rect = getrect;
    rimX = round(rect(1)); rimY = round(rect(2));
    rimW = round(rect(3)); rimH = round(rect(4));
    rectangle('Position', [rimX, rimY, rimW, rimH], 'LineWidth', 2, '
        EdgeColor', 'g');
    pause(1);
    close(f1);
end
%% create Kernel Density Estimation (Parzen Window)
R = 1;
[kernel, gx, gy] = createKernel(H,W,R);
```

```
[rimKernel, rimGx, rimGy] = createKernel(rimH,rimW,R);

% figure(3); mesh(kernel); title('Gaussian Kernel');
%% indexing from rgb2ind
[indexedImage, colorMap] = rgb2ind(mov(startFrame).color,65536);
LMap = length(colorMap)+1;
indT = rgb2ind(T,colorMap);

indRim = rgb2ind(Trim,colorMap);
%% Create Histogram
hist = createHist(indT, LMap, kernel, H,W);
histRim = createHist(indRim, LMap, rimKernel, rimH,rimW);

%% Similarity Value
f = [];

%% Play video

f_lowThresh = 0.04;
f_highThresh = 0.20;
max_it = 5;
expansion = 0;
shotTaken = 0;
movieOutput = VideoWriter(strcat(str,'.avi'));
movieWrite = 0;
open(movieOutput)
for i = startFrame:nFrames
    Iclr = mov(i).color;
    Iind = rgb2ind(mov(i).color, colorMap);

   %% for the first frame
    if i == startFrame
        %% Find the location of the ball and rim within the first frame
            .
        if selectPatch == 0
        fprintf('Finding the ball and the rim...\n'); tic
        fcatBall = [];
        fcatRim = [];
        locCat = [];
        locCatRim = [];
        nWindX = round(movW) / W;
        nWindY = round(movH) / H;
        nWindows = nWindX*nWindY;
        %--- parse through a moving window of the entire image---
        for j = 1:nWindX-1
            x0 = 1+(W * (j-1)); %update x0 based on
            for k = 1:nWindY-1
                y0 = 1 + (H * (k - 1));

                % --- search for the ball ---
                [ballX,ballY,ballF] = MS_Tracking(hist, Iind, LMap,...
                    movH, movW, f_highThresh, 3, x0, y0, H,W,...
                    kernel, gx, gy);
                fcatBall = cat(1, fcatBall, ballF);
                locCat = cat(1, locCat, [ballX,ballY]);

                % --- search for the rim ---
                [rimX,rimY,rimF] = MS_Tracking(histRim, Iind, LMap,...
                    movH, movW, f_highThresh, 4, x0, y0, rimH,rimW,...
                    rimKernel, rimGx, rimGy);
                fcatRim = cat(1,fcatRim, rimF);
                locCatRim = cat(1, locCatRim, [rimX, rimY]);
            end
        end
        [f, indx] = max(fcatBall);
        x = locCat(indx,1);
        y = locCat(indx,2);
        imshow(mov(i).color);
        rectangle('Position', [x,y, W, H], 'LineWidth', 2, 'EdgeColor',
            'g');
        rectangle('Position', [x,y, W, H], 'LineWidth', 2, 'EdgeColor',
            'r');

        [~, indxRim] = max(fcatRim);
        rimX =  locCatRim(indxRim, 1);
        rimY = locCatRim(indxRim,2);
        rectangle('Position', [rimX,rimY, rimW, rimH], 'LineWidth', 2,
            'EdgeColor', 'r');
        toc;
        end
%% if we are still detecting the ball
    elseif f > f_lowThresh

        [x,y,f] = MS_Tracking(hist, Iind, LMap,...
            movH, movW, f_highThresh, max_it, x0, y0, H,W,...
            kernel, gx, gy);
        f
        if f < f_lowThresh
            expansion = 1;
        else
            expansion = 0;
        end
    else
        expansion = 1;
    end

    %% if we need to enter Expansion
    if expansion == 1
        fprintf('Expanding the search area...\n');

            rectangle('Position', [x0,y0,W,H], 'EdgeColor', 'r', '
               LineWidth', 2);
        %% Start Expansion Function
        [f,x,y] = expansionAlg(hist, Iind, LMap, movW, movH,
            f_highThresh, ...
    max_it, x0,y0, H,W, kernel, gx, gy);
                if f > f_lowThresh
            expansion = 0;
        end
    end
    cx = round((x0 + W/2));

    %% check if the ball is near the hoop.
    nearHoop = 0;
    cy = round((y0 + H/2));
    if cx < rimX + 2*rimW && cx > rimX
        if cy < rimY + rimH && cy + H/2> rimY
            nearHoop = 1
            if shotTaken == 0
                shotTaken = 1;
            end
            fCat = [];
            locCat = [];
        end
    end
    %% if it's near the hoop, do extra searching
    if nearHoop == 1
        rectangle('Position', [x0,y0,W,H], 'EdgeColor', 'r', 'LineWidth
            ', 2);
        [f, x,y] = expansionAlg(hist, Iind, LMap, movW, movH,
           f_highThresh, ...
    max_it, x0,y0, H,W, kernel, gx, gy);
    end

     %% Miss/Make Detection
    miss = 0;
    make = 0;

   if shotTaken == 1
       if cy + H/2 < rimY || cx < rimX || cx > rimX + rimW
           miss = 1;
           shotTaken = 2;
       end
       if cy + H/2 > rimY + rimH
           make = 1;
           shotTaken = 2;
       end
   end

   %% Display Make or Miss
   if shotTaken == 2
       if miss == 1
           txt = ['Miss at frame ',num2str(i)];
           disp(txt);
           text(x+5,y,'\leftarrow MISS','Color','red','FontSize',14);
           pause(3);
       end
       if make == 1
           txt = ['Make at frame ',num2str(i)];
           disp(txt);
           text(x+5,y,'\leftarrow MAKE','Color','red','FontSize',14);
           pause(3);
       end
       shotTaken = 3;
   end

   miss = 0;
   make = 0;


   if shotTaken == 3 && movieWrite == 0
       for z = 1:30
```

```
                newFrame = getframe;
                writeVideo(movieOutput, newFrame)
            end
            movieWrite = 1;
        else
            newFrame = getframe;
            writeVideo(movieOutput, newFrame)
        end
        pause;

        if cy > movH/2
            shotTaken = 0;
            movieWrite = 0;
        end

        x0 = x;
        y0 = y;
        imshow(mov(i).color);
        rectangle('Position', [x, y, W, H], 'LineWidth', 2, 'EdgeColor', 'g'
            );
        rectangle('Position', [rimX, rimY, rimW, rimH], 'LineWidth', 1, '
            EdgeColor', 'y');

    end

    close(movieOutput);
```

Mean Shift Tracking

```
%% Function for Mean-Shift Tracking
% Iind = indexed version of the image (instead of RGB, it's in indexed
% color)
% LMap = length of the color map
% kernel = normal kernel
% f_highThresh = threshold for when the similarity value is close
    enough to
% the template
% movH = height of the movie
% movW = width of the movie
% H = Height of the template and the image
% W = Width of the template and the image
% hist = array with the number of bins the same as LMap. Colors are put
% into each of the bins to create a histogram of the indexed colors
% gx = gradient in the x direction
% gy = gradient in the y direction
% [x0, y0] = initial location of the previous selection window
% [x,y] = final output of the location of the new window.
% f = similarity value
% max_it = maximum number of iterations
function [x,y,f] = MS_Tracking(hist, Iind, LMap,...
             movH, movW, f_highThresh, max_it, x0, y0, H,W,...
             kernel, gx, gy)
y = y0;
x = x0;
T2 = Iind(y:y+H-1, x:x+W-1);

hist2 = createHist(T2,LMap, kernel,H,W);
[f,w] = compareBoxes(hist,hist2,T2,kernel,H,W);
for iteration = 1:max_it
    if f > f_highThresh
        break;
    end

    num_x = 0;
    num_y = 0;
    den = 0;
    for i = 1:H
        for j = 1:W
            num_x = num_x + i*w(i,j)*gx(i,j);
            num_y = num_y + j*w(i,j)*gy(i,j);
            den = den + w(i,j)*norm([gx(i,j), gy(i,j)]);
        end
    end
    %% Calculate how much to displace the new box by
    if den ~= 0
        dx = round(num_x/den);
        dy = round(num_y/den);
        y = y+dy;
        x = x+dx;
    end
    if y <= 0
        y = 1;
    end
    if y+H >= movH
        y = movH - H -1;
    end
    if x <= 0
```

```
            x = 1;
        end
        if x + W >= movW
            x = movW - W - 1;
        end
        T2 = Iind(y:y+H-1, x:x+W-1);
        hist2 = createHist(T2, LMap, kernel, H, W);
        [f, w] = compareBoxes(hist,hist2,T2,kernel,H,W);
end
```

Expansion Algorithm

```

%% Function for Expansion Algorithm, this occurs when the algorithm
    loses
% track of the ball, based on the similarity value.
% Iind = indexed version of the image (instead of RGB, it's in indexed
% color)
% LMap = length of the color map
% kernel = normal kernel
% f_highThresh = threshold for when the similarity value is close
    enough to
% the template
% movH = height of the movie
% movW = width of the movie
% H = Height of the template and the image
% W = Width of the template and the image
% hist = array with the number of bins the same as LMap. Colors are put
% into each of the bins to create a histogram of the indexed colors
% gx = gradient in the x direction
% gy = gradient in the y direction
% [x0, y0] = initial location of the previous selection window
% [x,y] = final output of the location of the new window.
% f = similarity value
% max_it = maximum number of iterations
function [f,x,y] = expansionAlg(hist, Iind, LMap, movW, movH,
    f_highThresh, ...
    max_it, x0,y0, H,W, kernel, gx, gy)
fCat = [];
locCat = [];

for j = 1:8
    [x1, y1] = expandTemplate(j, x0, y0, W, H, movW, movH);
    rectangle('Position', [x1,y1, W,H], 'EdgeColor', 'g', 'LineWidth',
        2);
    [newX,newY,f] = MS_Tracking(hist, Iind, LMap,...
        movH, movW, f_highThresh, max_it, x1, y1, H,W,...
        kernel, gx, gy);
    rectangle('Position', [x1,y1, W,H], 'EdgeColor', 'b', 'LineWidth',
        2);
    fCat = cat(1, fCat, f);
    locCat = cat(1, locCat, [newX,newY]);
end
[f, index] = max(fCat);
if f > 0.02
    x = locCat(index, 1);
    y = locCat(index, 2);
else
    x = x0;
    y = y0;
end


end
```

Histogram Creation

```
%% Function for creating a histogram from an image
% T = image input
% LMap = length of the color map
% kernel = kernel
% H = Height of the template and the image
% W = Width of the template and the image
% hist = array with the number of bins the same as LMap. Colors are put
% into each of the bins to create a histogram of the indexed colors

function hist = createHist(T, LMap, kernel, H,W)
hist = zeros(LMap,1);
for i = 1:W
    for j = 1:H
        hist(T(j,i)+1) = hist(T(j,i)+1) + kernel(j,i);
    end
end
% ---Normalize the Histogram ----
hist = hist./sum(sum(kernel));
end
```

Similarity Function

```
%% compareBoxes(hist, hist2, T2, kernel, H,W)
%This function compares the similarity between two histograms
% hist = Histogram from the template image
% hist2 = Histogram from the selection window
% T2 = Selection Window
% kernel = the kernel being used. In this case, a normal kernel
% H = Height of the template and the selection window
% W = Width of the template and the selection window
% f = similarity value
% w = weight array

function [f, w] = compareBoxes(hist,hist2,T2,kernel,H,W)
w = zeros(H,W);
f = 0;
for i = 1:H
    for j = 1:W
        w(i,j) = sqrt(hist(T2(i,j)+1) / hist2(T2(i,j)+1));
        f = f+w(i,j)*kernel(i,j);
    end
end

f = f/(H*W);
```

Kernel Creation

```
%% Function for creating a the kernel
% H = Height of the template and the image
% W = Width of the template and the image
% R = Radius of the kernel

function [kernel, gx, gy ] = createKernel(H,W,R)
%There are a few different Kernels we can use
%We are using the normal (gaussian) Kernel because the shape fo the
    weight
%seems to make the most sense for a ball
kernel = zeros(H,W);

%% Normal (Gaussian)
sigmaH = (R*H/2)/3;
sigmaW = (R*W/2)/3;
for i = 1:H
    for j = 1:W
        kernel(i,j) = exp(-1/2*((i - 1/2*H)^2 / sigmaH^2+...
            (j-1/2*W)^2/sigmaW^2));
        end
end

%% Uniform
% for i=1:H
%       for j=1:W
%           if (((2*i)/H-1)/R)^2+(((2*j)/W-1)/R)^2 <= 1
%               kernel(i,j) = 1;
%           end
%       end
% end

%% Epanechnikov
% for i=1:H
%       for j=1:W
%           kernel(i,j) = (1-(2*i/(R*H)-1/R)^2-...
%               (2*j/(R*W)-1/R)^2);
%           if kernel(i,j) < 0
%               kernel(i,j) = 0;
%           end
%       end
% end
[gx, gy] = gradient(-kernel);
```

Expand the template search

```
%% Function for expanding the selection window
% j = iteration number
% x0 = initial x location
% x1 = final x location
% y0 = initial y location
% y1 = final y location
% H = Height of the template and the image
% W = Width of the template and the image
% movW = width of the movie
% movH = height of the movie.

function [x1, y1] = expandTemplate(j, x0, y0, W, H, movW, movH)
if j ==1
      x1 = x0-W;
      y1 = y0-H;
elseif j ==2
      x1 = x0;
      y1 = y0-H;
```

```
elseif j ==3
        x1 = x0+W;
        y1 = y0-H;
elseif j ==4
        x1 = x0-W;
        y1 = y0;
elseif j == 5
        x1 = x0 + W;
        y1 = y0;
elseif j ==6
        x1 = x0 - W;
        y1 = y0 + H;
elseif j == 7
        x1 = x0;
        y1 = y0 + H;
elseif j == 8
        x1 = x0 + W;
        y1 = y0 + H;
end

%% handle any negative or out of bounds
if y1 <= 0
        y1 = 1;
end
if y1+H >= movH
        y1 = movH - H -1;
end
if x1 <= 0;
        x1 = 1;
end
if x1 + W >= movW
        x1 = movW - W -  1;
end
```

Finding the Displacement of the Selection Window

```
%% Finding the displacement for the selection window.
%This function finds the displacement for the selection window.
% w = weighted array
% gx = gradient of the kernel in the x direction
% gy = gradient of the kernel in the y direction
% H = Height of the template and the selection window
% W = Width of the template and the selection window

```

```
function [x,y] = findDisplacement(w, gx, gy, H, W)
num_x = 0;
num_y = 0;
den = 0;
for i = 1:H
    for j = 1:W
        num_x = num_x + i*w(i,j)*gx(i,j);
        num_y = num_y + j*w(i,j)*gy(i,j);
        den = den + w(i,j)*norm([gx(i,j), gy(i,j)]);
    end
end
%% Calculate how much to displace the new box by
if den ~= 0
    dx = round(num_x/den);
    dy = round(num_y/den);
    y = y+dy;
    x = x+dx;
end
```