AM 205 Final Project: Basketball Motion Capture

Nicholas Beasley
William C. Burke
Michael S. Emanuel

1 Introduction

1.1 Abstract

The goal of this project was to infer the position of a basketball in three dimensional space during a live game by integrating information obtained from multiple video recordings. In particular, the intuition was that with enough cameras, it should be possible to construct an over-determined system and perform a least squares or similar style estimation of the position of the ball that is in best agreement with the video frames. Steps we planned to take to achieve our goal included:

- obtaining video of a pickup game in the the Malkin Athletic Center; processing the video
- synchronizing the video based on the audio from each clip
- mapping from 3D world coordinates to 2D pixel locations on each camera
- tracking the ball in 2D space on each camera;
- inferring the position of the ball in 3D space based on its pixel location on each camera.
- simulating the path of the ball while in flight using an extended Kalman filter and the equations of motion
- predicting whether a shot would go in the basket at the moment it was released

When we articulated this project, we recognized that it was ambitious and that it might not be possible to do everything we wanted to in the time available. It has turned out that a number of the steps, while conceptually straightforward, have proven to be quite challenging and consumed far more time than we had budgeted. We present below everything that we were able to accomplish and hope for the forbearance of readers who are familiar with the slings and arrows of working on demanding problems.

In the end we were successful in capturing the 3D trajectory of the ball. We created a video showing a red circle tracking the ball: Motion Capture Video It shows where the simulation thinks the outline of the ball should on each frame. Unfortunately the version on YouTube is so blurry due to bandwidth limitations imposed by YouTube that it is difficult to see the red dots. Our Dropbox (linked below) has a much better high resolution version of the MP4 video.

1.2 Motivation

Autonomous robots including autonomous vehicles are a field of major research activity and commercial interest. One of the major challenges in designing these robots is sensing their environments; this is also an active research area. Autonomous driving systems on the road today have expensive and complex sensing systems that often include multiple cameras and LIDAR. This was one motivation behind the idea of attempting to solve a problem of perceiving the motion of objects in a complicated environment that also included people.

Separate from any directly practical applications, basketball is one of the popular sports in both the US and the world. The NBA had $14 billion in revenues in 2017. According to Wikipedia it is the second most popular sport in the US and has the highest participation. [1] A practical system that would enable accurate 3D coordinate positions of the ball during live games might attract considerable interest, both for entertainment and sports analytics. Ironically, the entertainment aspect of this project may be its most practical application. The video game NBA 2K17 sold over 8.5 million copies, making it the top sports game for Take Two Interactive with sales of $348 million. [2] What does this have to do with an overly ambitious AM 205 project? One of the top selling points of these games is verisimilitude. If a team could come up with an efficient procedure to harvest high resolution 3D coordinates of the ball and players from HDTV footage of NBA and NCAA games, this data could potentially be used to improve these video games. One immediate use case would be in training the AI agents that control the actions of players not played by people.

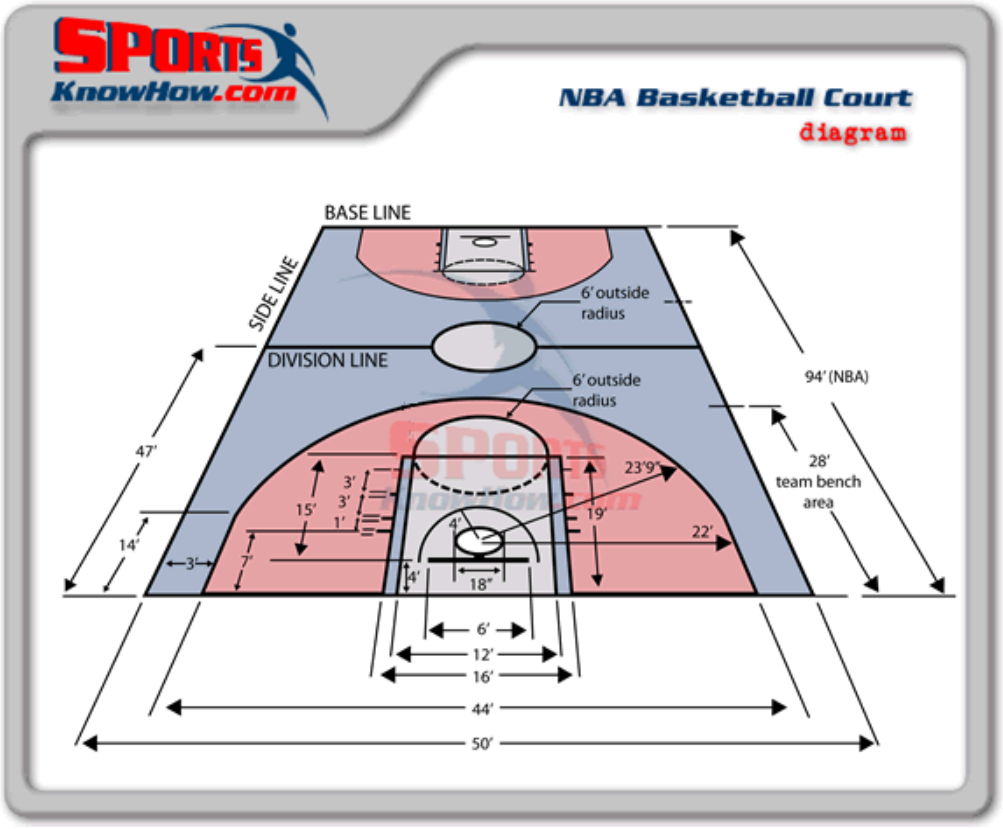
[1] sports in the united states
[2] NBA 2K17

2 Data Acquisition and Practical Details

2.1 Experimental Set-Up

We purchased eight identical video cameras from Amazon and compatible tripods. We selected the lowest cost camera we could find that supported full HD video. This camera [3] does not offer world beating performance, but at $64 it is very affordable for what it can do: either full HD video (1920x1080) at 15 FPS, or "regular" HD (1080x720) at 30 FPS. The tripod [4] is also very affordable at $21 and of decent quality. We used 64 GB SDXC cards that cost $15.50 each. [5] The relatively low cost of the equipment used makes our set-up quite accessible to e.g. high school basketball teams.

Here is a diagram of a basketball court shown in 3D perspective:



We intended to place eight cameras around the court, numbered from 1 to 8 as follows. Camera

[3] Sosun HDTV video camera
[4] Amazon Basics tripod
[5] SanDisk SD Card

1 was placed at "12 o'clock" on the diagram, i.e. behind the clear backboard at the top of the picture. Camera 2 was placed at the top right of the diagram, in the corner. Camera 3 was placed on the right side of the midcourt line. The camera numbers continue increase moving clockwise around the court, with cameras 4 through 8 at the bottom right; behind the near basket; bottom left; center left; and top left, respectively.

We performed this experiment at the Malkin Athletic Center ("MAC") on Friday, November 16. We used a ladder and electrical tape to mount Camera 1 behind one backboard. (We did this while the staff at the MAC wasn't looking... We used a buddy system to ensure ladder safety. Undergraduates were not allowed to risk standing on ladders; that activity was limited to graduate students only.) The backboard on the opposite side of the court was raised up towards the ceiling, so we were unable to obtain footage from Camera 5. We made our best effort to orient the cameras to provide useful footage for a half-court basketball game. The camera orientations would need to be different to capture a full court game. We were lucky enough that some gentlemen who were playing agreed to play a short game on the court where we set up our cameras to assist in our experiment. The game lasted approximately 2:26. This might not sound like much, but it amounts to just under 4400 frames at 30 FPS. We had about 20-25 minutes of mostly junk footage, so the pick-up game was solid gold by comparison. This data set was already at a size where it put significant strain on our computational resources, particularly those of us working on laptops.

Lesson Learned: When in doubt, take a measurement

We brought a large tape measure with us to the gym, but in the bustle of setting up all our cameras and climbing ladders, we neglected to confirm the measurements of the court, naively assuming that the might Harvard would have regulation NCAA basketball courts. Bad idea... after wasting over three hours failing to visually calibrate our cameras, we realized that the measurements we had been using were wrong. After we went back measured the court dimensions, the visual calibration was quicker and more accurate.

2.2 Video Processing

This project posed numerous practical challenges. Two of them were editing the video files we obtained from the cameras and sharing files that are far too large for GitHub with across our team. The video cameras have SD cards, which we used to transfer our video files to a PC. The cameras recorded in full HD (1920x1080) at 15 frames per second. However, while the stated resolution was full HD, the image is not nearly as sharp as a high quality camera recording at this resolution would be. We obtained AVI files from the camera, and the first step in our workflow was to edit down the 20-25 minutes of footage obtained from each camera to segments of around 2:45 containing the game with a short buffer on either side. This was done using Adobe Premier video editing software. The output of this step of the analysis was a set of 7 MPG files. These files were very roughly time synchronized, to a resolution of around 10 seconds. To complete the initial batch of processing, we extracted frames from these videos using ffmpeg, which is an excellent free open source tool.

Lesson Learned: Use ffpmg for Simple Video Manipulation

We only realized that ffmpg has all the features we needed after we had already laboriously edited our video by hand in Adobe Premier. All we really needed to do here was merge and trim feeds; convert from AVI to MP4; and extract frames and audio (WAV) from the MP4. These operations are very efficient in ffmpg once you learn how to make commands. They can also be scripted and reused. Adobe Premier or a similar tool is still excellent for reviewing the video files and determining the start and end times for desired files. Our advice for others trying this task for the first time would be to write down the times of the desired cuts on paper, then write a command line script to do the video tasks in `ffmpeg`.

2.3 Accessing our Code and Data: GitHub and DropBox

All of our code for this project is available here on a public GitHub repository:
GitHub Harvard-AM-205-Basketball. While we will follow all submission instructions and upload individual Python files with this report, the most efficient way get all of our code with the correct directory layout is to clone the repository:

```
git clone https://github.com/Harvard-AM-205-Basketball/Basketball.git
```

Our team collaborated using a combination of GitHub for source code and Dropbox for large shared files. We created a shared folder with all of our video and audio files. This got to be quite large. The AVI files are about 26 GB. The trimmed MPG files are about 8 GB. The fun really started when we extracted frames. These consumed a whopping 122 GB! (Back of the envelope: there are 5,000 frames at 1980x1020 pixels, with 3 8 bit color channels each). The movie has much more compression because most of the frames don't change that much. One technique we used to manage having different folder locations was to create links in our project directory pointing to the Dropbox folders. Dropbox no longer supports public folders, so the only way we know to share our data files is to send an invitation to access the shared folder. We are of course happy to do this. Dropbox also supports the creation of links. This link should allow any reader to download all of our data: Dropbox Harvard-AM-205-Basketball

Please be warned though that Dropbox counts the entire size of a shared folder against the space quota of everyone who has access to it. So if you are on a free Dropbox plan, sharing this folder will exhaust your space quota and leave you temporarily unable to upload files. We are still trying to figure out if there is a better way to work collaboratively on large data files and make them available to the public in the interests of replicable science. In the interim, the ability of any reader to download our data via the above link is satisfactory if not ideal.

2.4 Software Environment and Code Files

The software environment for all code on this project was Python 3.6 or 3.7. Different machines used by members of our team included a Mac laptop, a Windows Surface, a Windows PC, and a powerful server running Ubuntu Linux. Below is a list in dependency order of source code files (including scripts) used in this project.

Source Files for Basketball Tracking:

- `make_links_windows.bat` - create symbolic links in the repository to the Dropbox folder with large files
- `am205_utils.py` - utility functions for AM 205
- `image_utils.py` - utility functions used to handle image frames
- `audio_synchronize.py` - synchronize movie files (MP4, AVI, etc) by aligning their WAV audio tracks
- `synchronize_frames.py` - generate synchronized frames using the lags produced from audio synchronization
- `synchronize_video.bat` - combine the synchronized frames back into synchronized movies using `ffmpeg`
- `make_tableau.py` - combine 6 frames into one image to visualize the entire scene and demonstrate that they are synchronize; batch to process all the frames
- `basketball_court.py` – visualize and compute (x, y, z) coordinates of landmarks on basketball courts including the perimeter, key, three point line, etc.
- `camera_transform.py` – create transformation functions for a camera mapping between world coordinates of objects they see and pixels they produce
- `pixel_size.py` – numerically solve for the intrinsic pixel sizes of these cameras using landmarks on the basketball court and least squares fitting
- `camera_calibration.py` – solve for the placement and orientation of cameras by visually aligning landmarks on the basketball court with their transformed pixel locations
- `camera_calibration_num.py` – numerical calibration of camera placement and orientation by least squares solution mapping landmarks on the court to their known pixel positions
- `image_background.py` – compute the mean and median frame for each camera
- `image_foreground.py` – extract the foreground from each frame using `OpenCV`

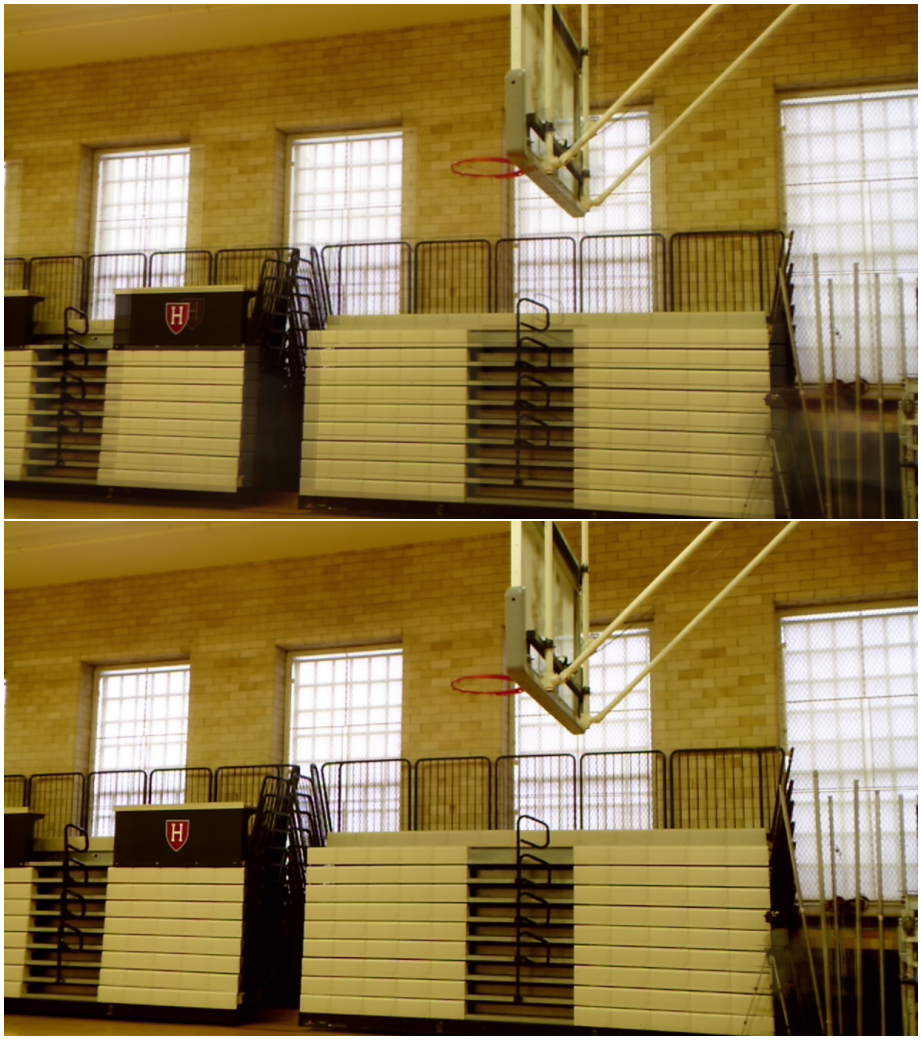
- `object_track_cv.py` – failed attempt to track the basketball's pixel location using `OpenCV`
- `ball_track.py` – track the position of the ball in world coordinates based on pixel coordinates at multiple cameras. also creates annotated images of each frame showing a red circle around the projected ball position, and an annotated tableau frame.
- `ball_stats.py` – assemble the outputs of the ball tracking into a Pandas DataFrame including frame number, time, and the inferred x, y and z coordinates of the center of the basketball in the world frame

3 Estimating the Background and Foreground Image

A common technique in image processing is Foreground Detection, which is related to estimating the background and subtracting it out. [6] We had two reasons to get the background image at each camera. First, to aid in foreground detection which we could use as input to the part of the system that would track the ball in 2D for each camera. Second, to aid us in calibrating the location of each camera (more on this below). An obvious idea that can be run quickly is to take the mean of each pixel over all the frames. This has some obvious shortcomings though, as the pixels get blurred from frames where objects are in the foreground. We hit on the idea of estimating the median value of the R, G and B color component of each pixel. This calculation was performed in the Python file `image_background.py`. There is not much going on in this file of mathematical interest. Getting the program to run to completion however was a nontrivial task! The challenge is that loading all the frames for even one camera consumes a prodigious amount of memory. While it's possible to compute the mean of all the frames by a doing a single pass through them and accumulating the sum, we are unaware of a similar trick for the median. In the end we got this clean but resource intensive implementation to run on an Ubuntu server that has 256 GB of RAM. This is a good example of the kinds of challenges that cropped up somewhat unexpectedly and consumed large amounts of time.

Here are examples of the mean and median background image:

[6] Foreground Detection



Mean and Median Frame for Camera 2

This camera gave a good example of why we went to the trouble of estimating the median. The mean is visibly blurry (especially around the "H" logo on the bleachers) and median is considerably more crisp.

We ended up devoting a substantial amount of effort to extract the foreground of each frame. One member of our team spent the better part of a week trying to get this to work. The reason we devoted so much effort to this was mostly because we thought we would need it to track the pixel location of the ball on the different cameras. We ended up with 4,391 synchronized video frames on each of our 7 cameras for a grand total of 30.737 frames, so we had a strong preference

for fully automated processing of anything pertaining to frames. The software tracking system we had hoped to use for this purpose was unable to track the ball in the original video footage. Our hope had been that if we extracted the foreground, the frames with just the foreground would allow the ball to be tracked. Unfortunately the foreground estimation proved to be a rabbit hole for us.

Initial attempts were based on comparing each frame to the median and setting a tolerance threshold. If a pixel was different from the background by more than this amount, then it was declared to be foreground. This did not work at all, and in retrospect it was too simplistic. Eventually we did some more research into this and it turns out that both foreground and background detection are challenging and well studied problems. In the end, we did succeed, after many hours of work, in automatically extracting decent quality foreground images from all the frames. It was somewhat of a Pyrrhic victory because it consumed so much of our time that we didn't really have much time left to see if we could use them for automated ball tracking. However, because the foreground detection problem is intrinsically interesting, and because we spent quite a lot of effort on this portion of the project, here are a few of the highlights. We used the `OpenCV` computer vision library. This is an excellent tool that performs many standard computer vision tasks. It is written in C++ and also available with convenient bindings to Python. OpenCV includes a handy gadget called a `BackgroundSubtractor`. The file `image_foreground.py` uses this OpenCV class to generate foregrounds for all of our frames. This job ran in approximately one hour on a high end desktop PC. Here is one example frame, taken from Camera 3 at frame 2316 (1:17 into the game):

As we can see, the OpenCV library did an excellent job of picking just a small number of pixels that were legitimately in the foreground. Crucially, it has identified the ball. Had we realized how hard it was to do this by hand, and how easy it was in OpenCV (well once you get up to speed on OpenCV, which is nontrivial...) we might have run this and then written a "basketball detector" by hand to track the ball.

Lesson Learned: Use OpenCV instead of hand-rolled functions for computer vision tasks!

4 Synchronizing the Frames Using Audio Feeds

In order to assemble a 3D trajectory of the ball by combining frames from different cameras, it is necessary first to synchronize the feeds. This is a recurring and well studied problem in videography with multiple cameras. Almost everyone faced with this synchronization problem tries to use the audio rather than the video feeds to synchronize the cameras. There are two reasons for this choice. First, the temporal resolution of sound is much sharper than the temporal resolution of video frames. These cameras were only recording at 15 FPS (we output 30 FPS to get a bit more resolution in case two cameras were separated by 0.03 seconds). Sound features on the other hand have a microstructure in the thousands of Hertz. The second reason to favor sound for synchronization is that the sound environment at the different cameras will be much more similar than the visual environments.

We had thought that, based on the frequency with which this problem arises in different contexts, that there would be a "drop in" library solution that would take two WAV files and output the lag between them. An extensive online search proved otherwise. We did identify one library called VideoSync. A half day was spent trying to get this code to run on our input files. Unfortunately we just could not get it to run to completion. The program would run without throwing an exception, but would hang. After modifying it to add progress bars and running on the monster Linux server we still had no luck. Apparently this problem is usually solved in practice by highly trained video editors using expensive software, i.e. people do it by hand.

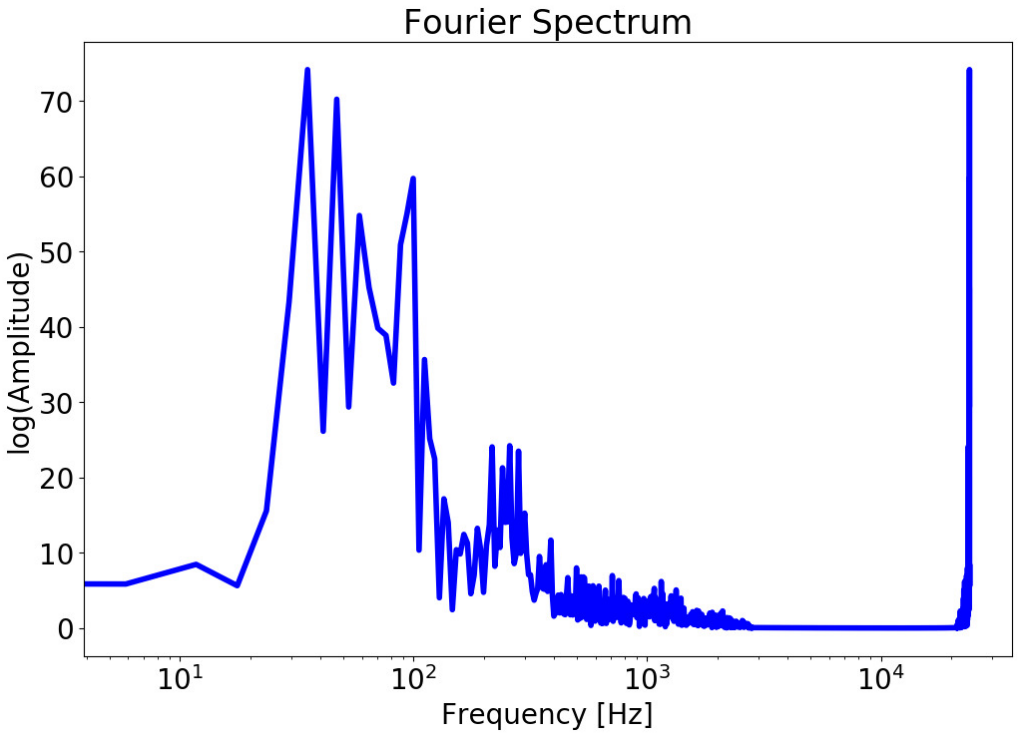
The idea behind synchronizing two audio feeds is intuitive. If $g(t)$ is similar to $f(t)$ but with a lag, then $g(t) \approx f(t+\tau)$ where τ is the time lag. So synchronizing the two feeds amounts to finding the τ that will maximize $\sum_t f(t)g(t-\tau)$. This looks very similar to a convolution between f and g, but with the roles of t and τ exchanged in the arguments of g. An introductory course in Fourier Analysis covers the fact the Fourier transform of the convolution $f * g$ is the product of the Fourier transforms of f and g. A practical application of this for discrete sampling problems such as this one is that convolution can be evaluated efficiently using the Fast Fourier Transform (FFT). One important thing to remember is that the definition of the convolution of $f * g$ is a sum over $f(x)g(t-x)$, so we need to reverse the order of the second array before running the convolution.

An initial attempt was made to find the time lag by convolving the two audio signals (i.e. sound waves loaded from WAV files). We created a simple test case by taking the audio from one camera and applying lags of 1.0 and 3.0 seconds to it. The convolution based approach (evaluate the convolution, then run `np.argmax` to find the lag with maximum overlap) worked flawlessly on this test data. But when it was tried on the actual audio feeds, it failed completely. It was way off, and the correlations between the signals was very low, in the vicinity of 0.02. The problem is that the sound waves oscillate very rapidly, and small differences in the sound environments at the cameras cause the signals to sometimes line up and other times not to line up. The test case worked well

because the dummy test data kept both the micro and macro structure of the waves. But while the macro structure of the sound at each camera (as perceived by a person) was very similar, the waves didn't line up that tightly.

The second attempt had two phases and was successful. The first phase was very low tech: by listening to the seven WAV files, we estimated the lags between them to a resolution of approximately one second. This was done by noting the times when audible phrases were spoken. ("I'm going to check the other cameras" and "oh, dude!" were uttered around the 5 and 26 second marks of the synchronized movie.). The signals were approximately aligned using these preliminary lags. We then applied a Fast Fourier Transform to the signals to estimate the sound environment during that window. The audio feed was sampled at 48,000 Hz.

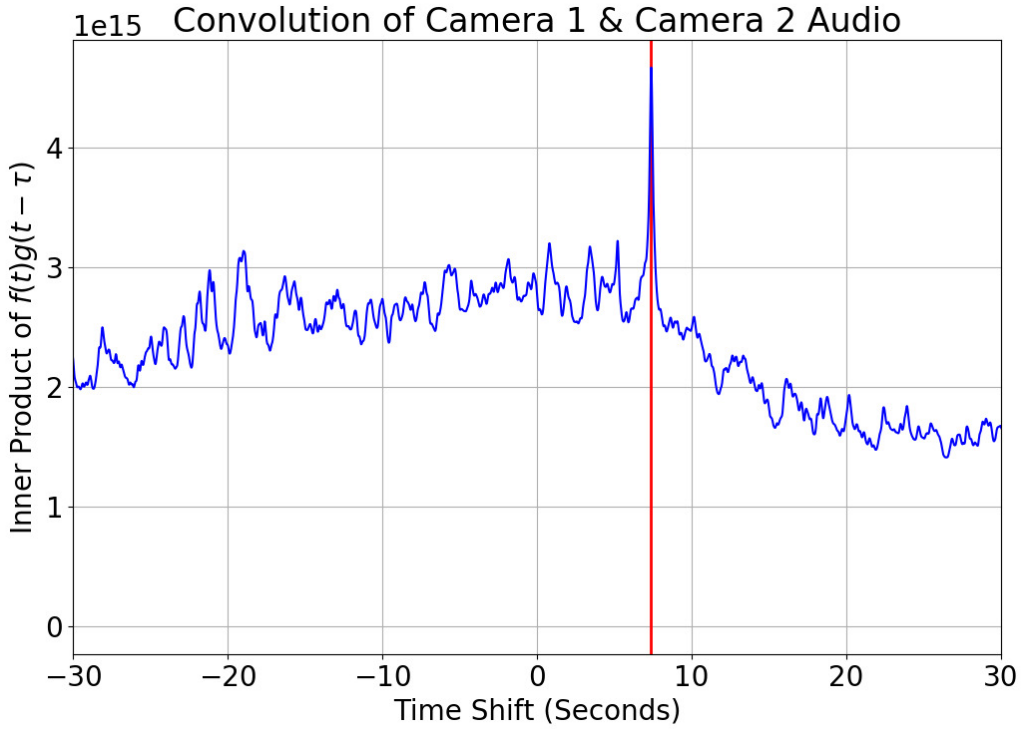
We referred to the discussion and code in audio processing in python when implementing this solution. We selected a window size of $2^{12} = 4096$ pulses, corresponding to 0.0853 seconds. (The FFT is most efficient when the window size is a power of 2; a sound window of roughly 0.1 seconds is a useful guideline.) These windows were sampled with a spacing of 1024 pulses, corresponding to a temporal resolution of 0.0213 seconds. This is small enough compared to the 0.033 seconds between frames at 30 FPS that if done correctly, it should allow us to synchronize to the nearest frame. An audio feed of 165 seconds was thus processed into approximately 7,800 overlapping windows. Each window was characterized by a vector of 4096 numbers from the FFT output. The soundscape of the basketball game had a lot of low frequency pulses from the dribbling of the basketball, and occasional high frequency squeaks of sneaker rubber on the wood floor. Here is a visualization of the sound frequency profile at camera 1 on the 100th bucket (about 2.0 seconds):



After each sound signal is converted to a Fourier spectrum, the spectra are then convolved against each other to find the lag between signals. In this case, rather than taking the dot product of two scalar functions of time, we are taking the dot product of two vectors containing the amplitude of the signal at different frequencies on the spectrum. The Fourier coefficient at each frequency is multiplied in the two signals; this product of coefficients is then summed up to produce a scalar characterized the overlap between the signals at a given time lag τ.

This approach was successful. It resolved the time delays to a resolution of 0.02 seconds, sufficient to synchronize the frames to the nearest frame number. We generated a 7x7 skew symmetric matrix of estimated time lags. The rows were averaged to form a composite estimate of the lags between cameras. As a quality metric, we generated a 7x7 correlation matrix of the spectra after they were synchronized. We then extracted the largest eigenvalue. For perfectly synchronized signals, the correlation matrix would be all 1s, and the first eigenvalue would be 7. This run achieved a first eigenvalue of 4.742.

Here is a visualization of the convolution used to determine the lag between camera 1 and camera 2.



This chart makes it clear that the signal estimating that the lag was at approximately +7 seconds was very strong. The function spikes up close to around 4.8E15 in a neighborhood where it had been below 3.2E15 . This form of the dot product tends to get smaller as the number of terms increases (i.e. both signals are padded with zeros and the inner product is not normalized to adjust for it). If the chart was continued all the way to either side, it would tail down to zero on both sides. For this reason, this particular approach will struggle to identify a large time lag. That is why the preliminary step of approximately synchronizing the feeds was performed.

All code used to synchronize the audio feeds can be found in the file `audio_synchronize.py`. The function `wav_to_freq` converts a WAV audio file to a Fourier spectrum using the FFT. The function `synchronize_spectra` finds the time lag between two spectra by convolving one against the reversal of the other and taking the argmax of the result. The function `make_synch_matrix` builds an 7x7 matrix of estimated lags. Even though there are only 6 degrees of freedom in this problem, we estimate all 42 entries and then average the rows to maximize the accuracy of the measurement. The full outputs of the synchronization calculation are in the text file `synchronization_output.txt`. in the `calculations` directory of our repository. The plot was generated with the function `plot_conv`.

Of course the proof of the pudding is in the eating. Here are images from six of the seven cameras of the first ostensibly synchronized frames:

If you pay close attention to the ball, you can see that in all six frames, the gentleman in the gray long sleeve shirt with a goatee is dribbling the ball with a high stroke, and the ball is close to its apex. This is particularly clear on the first three views if you start at the top left and read them like a page in a book (right, then down). We claim this is evidence that the audio synchronization was accurate to the nearest frame!

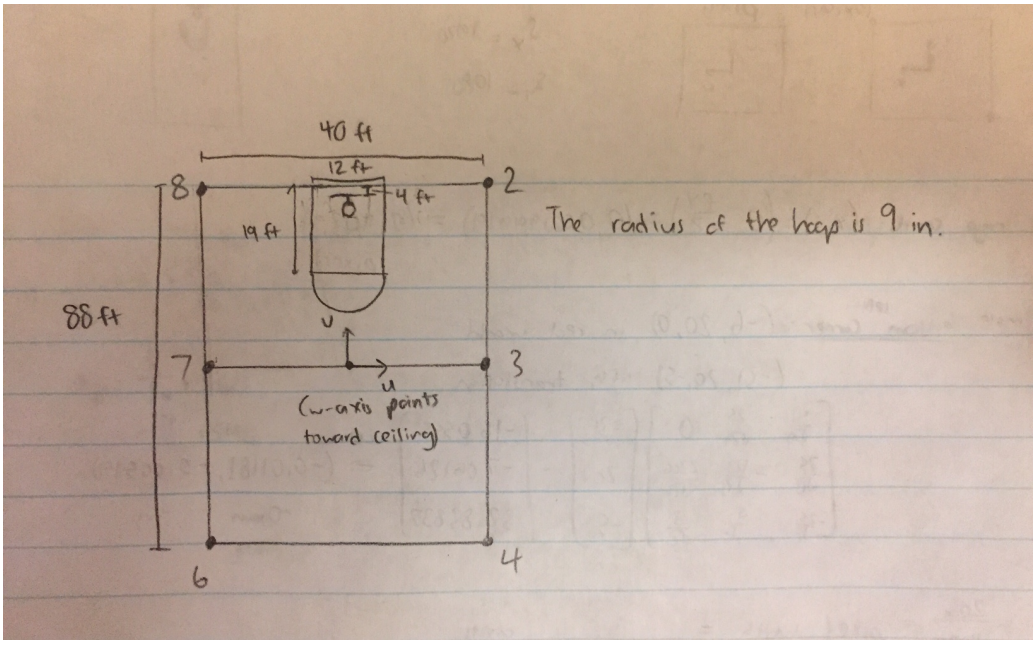
A more stringent test is to watch the movie assembled from the tableau frames. That is a movie showing the entire scene with frames like the one above. It was created using the Python script `make_tableau.py`. There isn't much mathematical content here, but it took a surprisingly long time to code it. The biggest challenge was inducing `matplotlib` to assemble the frames without padding. There was also some fun involved in parallelizing it without memory leaks. This was a big job; it ended up taking about four hours to run on the big Linux server. While there are some little hitches, if you watch this clip you should be convinced that the synchronization was accurate to within the resolution of the 15 FPS frame rate.

5 Transforming 3D World Coordinates to Pixel Locations

5.1 Conventions

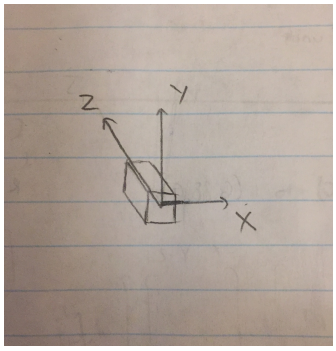
Coordinates in the world frame will be denoted as (U, V, W). Coordinates in the camera frame will be denoted as (X, Y, Z). Coordinates in the 2D image plane will be denoted as (x, y). Pixel coordinates will be denoted as (u, v).

The court has the following dimensions and coordinate system (all units are in FEET for this project because US basketball court sizes are standardized in feet, not meters). Note that we are centering the world coordinate system at the center of the court and that the w axis points towards the ceiling.



A diagram of the MAC basketball court with dimensions

For the camera frame, we will adopt the convention that the Z axis points in the direction that the camera is pointing. See the following diagram, where the rectangular prism represents the camera:



A hand sketched representation of the camera coordinate system

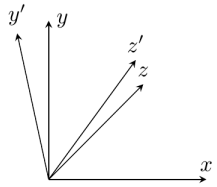
The image coordinate system has the origin at the center of the image, with the x-axis pointing to the right and the y-axis pointing up (i.e. standard conventions). The pixel coordinate system has $(0, 0)$ representing the pixel in the top left corner. Also note that it is represented as an array in Python with the first coordinate representing the row of the pixel (so the u axis points down and corresponds to the image $-y$ axis) and the second coordinate representing the column (so the v axis points to the right and corresponds to the image x axis).

5.2 World to Camera

To get from world coordinates to camera coordinates, we must perform a translation and then a rotation. Let the camera be placed at location $\boldsymbol{C}$ (denoting a vector) in the world frame, and let the rotation matrix be R. Since the rotation matrix R is designed to go from the world to the camera coordinate system, R^T is will go from the camera to the world. (The inverse of an orthogonal matrix is its transpose.) Thus, the first column of R^T should represent the camera X-axis in world coordinates, the second column should represent the Y-axis, and the third column should represent the Z-axis. In terms of R, we see that the first *row* of R must be equivalent to the camera X-axis in world coordinates, etc.

In practical terms, it is relatively easy to estimate what the camera Z-axis is in world coordinates. By looking at one frame and identifying the center pixel, we can identify where in the world coordinate system the camera is pointing. We used MS Paint in the prototyping stage to identify the center of the frame, followed by annotating plots with `matplotlib` in the production visual calibration routine presented below. The easiest way to do this is to consider where the ray pointing down the camera Z-axis hits a solid object. For example, if the center pixel is on the rim or the backboard, we can write a vector representing where that point is in relation to the camera position. We then normalize the vector to get the third row of R.

To get the X-axis, we rely on the assumption that the camera X-axis has a W-coordinate of 0 in the world frame (i.e. that it is parallel to the ground). This is a reasonable assumption because we start with the camera pointed parallel to the ground. When we tilt the camera slightly upwards, we don't rotate it in any other way, so the X-axis stays stationary while the Z and Y axes rotate (see Figure below). In fact, for the six cameras mounted on tripods, it is not an assumption at all; each tripod was set so the cameras would only tilt up and down. Interestingly, Camera 1 is not mounted on a tripod, but taped to the back of the backboard, and we can see a definite tilt in the visual calibration for that camera because this assumption *does* fail for that camera. Thus, we look at the first two world coordinates of the camera Z-axis, flip the order of the coordinates, and change one sign (we flip the sign of the U-coordinate so that the X axis points to the right). This ensures that the X and Z axes are orthogonal (have a dot product of 0).



Why the camera X axis is parallel to the ground

Finally, to get the Y-axis, we first normalize the X and Z axes, and then take the cross product of X and Z. We make sure X comes first so that the Y-axis points in the right direction by the right hand rule. We put our three unit vectors together and we have a rotation matrix R! We can now relate camera coordinates to world coordinates:

$$\boldsymbol{P}_C = R(\boldsymbol{P}_W - \boldsymbol{C})$$

5.3 Camera to Image

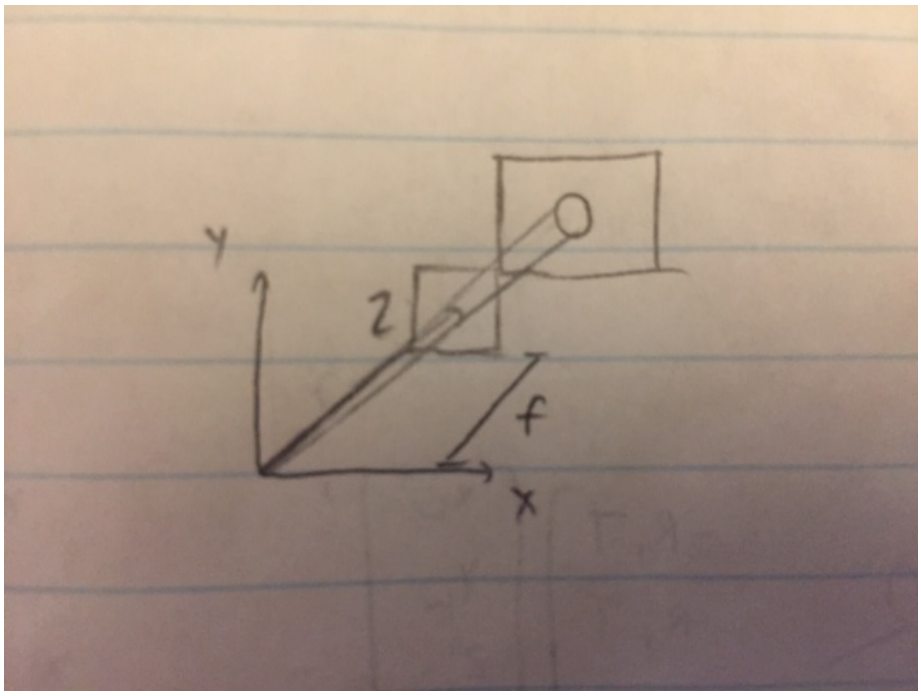
The key idea is that the "image plane" is an imaginary plane f units down the Z-axis of the camera, where f is the focal length. To get the image plane coordinates, we use similar triangles and scale

down a point that is Z units away to a plane that is f units away. We get:

$$x = \frac{f}{Z}X$$
$$y = \frac{f}{Z}Y$$

Note: Some sources like to use "homogeneous coordinates" so that we can write a matrix equation $\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$.

Then $(x', y', z') = (Xf, Yf, Z)$ and $(x, y) = (\frac{x'}{z'}, \frac{y'}{z'})$.



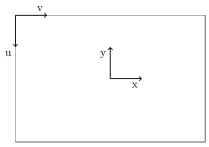
Projection onto image plane

5.4 Image to Pixel

To transform from image to pixel, we have to divide the image x coordinate by the width of a pixel and the image y coordinate by the height of a pixel. This will tell us how many pixels away a point

in the image is from the center of the image. Let the width of a pixel by s_x and the height of a pixel by s_y. Also, note that we must negate the image y coordinate because the pixel u axis points down instead of up. Right now we have $u = -\frac{y}{s_y}, v = \frac{x}{s_x}$. However, we have to add $\frac{1080}{2} = 540$ to u and $\frac{1920}{2} = 960$ to v so that the pixel $(0, 0)$ is in the top left corner. The exact numbers come from the fact that each frame is 1080 by 1920 pixels.

$$u = -\frac{y}{s_y} + 540$$
$$v = \frac{x}{s_x} + 960$$



Pixel coordinates vs. image coordinates

5.5 Extrinsic vs. Intrinsic Parameters

The rotation matrix R and the position of the camera $\boldsymbol{C}$ are considered **extrinsic parameters** in the sense that we set them. We can also obtain good estimates of what they are without doing any algebra or optimization. On the other hand, the focal length f and the pixel size s_x by s_y are **intrinsic parameters**, which means that they are properties of the camera. While the focal length is easily obtainable from the SOSUN website (7.36 mm unzoomed), the pixel size is unknown. However, we can estimate it using least squares!

We know the world coordinates of several points on the court, such as the free-throw rectangle and the white square on the backboard. Given our estimates for all of the other parameters, we can solve for the image coordinates (x, y) of these points relatively easily. We can also identify the pixel coordinates (u, v) of these points relatively accurately by examining a given frame (the camera is stationary). We have the equations:

$$v = \frac{x}{s_x} + 960$$
$$u = -\frac{y}{s_y} + 540$$

And we can write this as a matrix equation $\begin{pmatrix} v-960 \\ u-540 \end{pmatrix} = \begin{pmatrix} x & 0 \\ 0 & -y \end{pmatrix} \begin{pmatrix} \frac{1}{s_x} \\ \frac{1}{s_y} \end{pmatrix}$. We have many points where we know (x, y) and (v, u), so we can make this a least-squares problem where we solve for $\frac{1}{s_x}, \frac{1}{s_y}$!

$$\begin{pmatrix} v_1-960 \\ u_1-540 \\ v_2-960 \\ u_2-540 \\ \cdots \end{pmatrix} = \begin{pmatrix} x_1 & 0 \\ 0 & -y_1 \\ x_2 & 0 \\ 0 & -y_2 \\ \cdots & \cdots \end{pmatrix} \begin{pmatrix} \frac{1}{s_x} \\ \frac{1}{s_y} \end{pmatrix}$$

Assuming we have estimated all of the other parameters correctly, we should be able to get accurate numbers for s_x, s_y, allowing us to transform any point on the court into pixels.

5.6 An example, using Camera 3

Using Camera 1, we can see that Camera 3 is outside of the court, slightly behind and to the right of where it is supposed to be on the diagram. We also know that it is on a 5-foot high tripod. Using the diagram of the court, we estimate that its position relative to the world-frame origin is $\boldsymbol{C} = (22, -1.5, 5.2)$.

Next, we compute the rotation matrix. The center pixel of frame 20 (we could choose any frame) lies directly below the front of the rim. If we draw a line from the rim to the ground (using MS Paint), we estimate that the camera is pointed about 2 feet below the front of the rim. The point 2 feet below the front of the rim has world coordinates $(0, 41, 8)$, so the camera z-axis points in the direction $(-22, 42.5, 2.8)$. Our assumption is that the x-axis has a 3rd coordinate of 0 and points towards the northeast, so it must be of the form $(42.5, 22, 0)$ in order to be orthogonal to z (of course, it could be any scalar multiple of this). In Python, we normalize our two vectors and then take $x \times z$ to get the y-axis. Our rotation matrix is $R = \begin{pmatrix} \mathbf{x}_n \\ \mathbf{y}_n \\ \mathbf{z}_n \end{pmatrix}$ where the n subscript indicates that our vectors are normalized to have norm 1.

Note that our choice to look at the point 2 feet below the front of the rim was arbitrary, as the camera z-axis represents a ray. However, it was the easiest choice because it was a point on the ray with coordinates that were relatively easy to estimate.

From here, all we need to do is solve the least-squares point for the pixel size (s_x, s_y). For our "known" points, we take the free throw rectangle and the white box on the backboard. The four cor-

ners of the free throw rectangle are $(-6, 28, 0), (-6, 47, 0), (6, 47, 0), (6, 28, 0)$ in world coordinates, and the four corners of the box on the backboard are $(-1, 43, 10), (-1, 43, 11.5), (1, 43, 11.5), (1, 43, 10)$. We also consider some intermediate points on the free throw rectangle (I used all of the points with integer coordinates). Using Python, we can create a N by 3 matrix with the world coordinates of these points. With the parameters we already know, we can construct the 2D image plane coordinates of these points.

We can also get the pixel coordinates of all of our points with reasonable accuracy by looking at the photo in MS Paint or similar software. We did this by getting the pixel coordinates of the corners and using linspace. Now that we have our pixel coordinates and our 2D image plane coordinates for all of our points, we solve the least squares problem for s_x and s_y. Our final step in Python is to use our estimated values of s_x and s_y to complete the transformation and color the pixels corresponding to our real-world points (in this case, the free throw rectangle and the white box).

The numbers presented above were based on our initial estimates of the size of the court, which were subsequently revised with the assistance of a trusty tape measure. We took all the logic presented above and wrapped it up into one end to end calculation in the file `pixel_size.py`. This file uses the locations of features on the court that are computed in `basketball_court.py`, presented below. It also uses a handful of pixel coordinates for these landmarks. Running these calculations on the revised estimates for the size of the court, we find pixel dimensions $s_x = 1.196\text{E-}5$ and $s_y = 1.141\text{E-}5$. It is reassuring that these two estimates are within 4.8% of each other, because we expect the pixels to be square. (As an aside, one of the trickier downstream effects of our initially flawed estimates of the size of the court was that our initial pixel sizes were off by about 12%, which wrought havoc on the visual calibration.)

5.7 Python Implementation of Coordinate Transforms

The file `camera_transform.py` implements all of these formulas in a modular form.
We need both the rounded and unrounded transformation from world to pixels for different applications. The integer version is used for placing pixels on images based on an object's estimated position. The floating point version is used in numerical optimization, where it is critical that the functions we use are differentiable.
Function `cam2pix_f` converts coordinates (x, y) in the local image plane to floating point pixel values (u, v).
Function `cam2pix` is the integer version, rounding these to the nearest pixel.
Function `sph2xyz` converts spherical coordinate angles θ and ϕ to Cartesian coordinates.
Function `xyz2sph` converts Cartesian coordinates back to spherical angles θ and ϕ. These functions

are used later in an alternate characterization of the camera transforms that uses two angular inputs rather than three aiming points.

Function `rot_from_points` builds a 3x3 rotation matrix given `cam_pos`, the position of the camera in the world frame, and `cam_point`, the position of *any* object that the center of the camera is pointing at.

Function `make_transforms` takes as inputs the locations of the camera and its aim point, plus a zoom. This function is a "factory" that makes a trio of transformers returning (x, y) and (u, v) coordinates (both float and integer).

Function `make_transforms_angle` has the same outputs as `make_transforms`, but its inputs describing its orientation are θ and ϕ rather than the aiming point. This makes it better suited for the optimization used in the numerical camera calibration below.

5.8 Gratuitous Cultural Reference: Transformers

Optimus Prime

5.9 Resources

http://www.cse.psu.edu/~rtc12/CSE486/lecture12.pdf

http://www.cse.psu.edu/~rtc12/CSE486/lecture13.pdf

https://courses.cs.washington.edu/courses/cse455/09wi/Lects/lect5.pdf

https://www.cs.utah.edu/~srikumar/cv_spring2017_files/Lecture2.pdf

http://www.cs.toronto.edu/~jepson/csc420/notes/imageProjection.pdf

http://people.scs.carleton.ca/~c_shu/Courses/comp4900d/notes/camera_calibration.pdf

http://www.cs.ucf.edu/~mtappen/cap5415/lecs/lec19.pdf

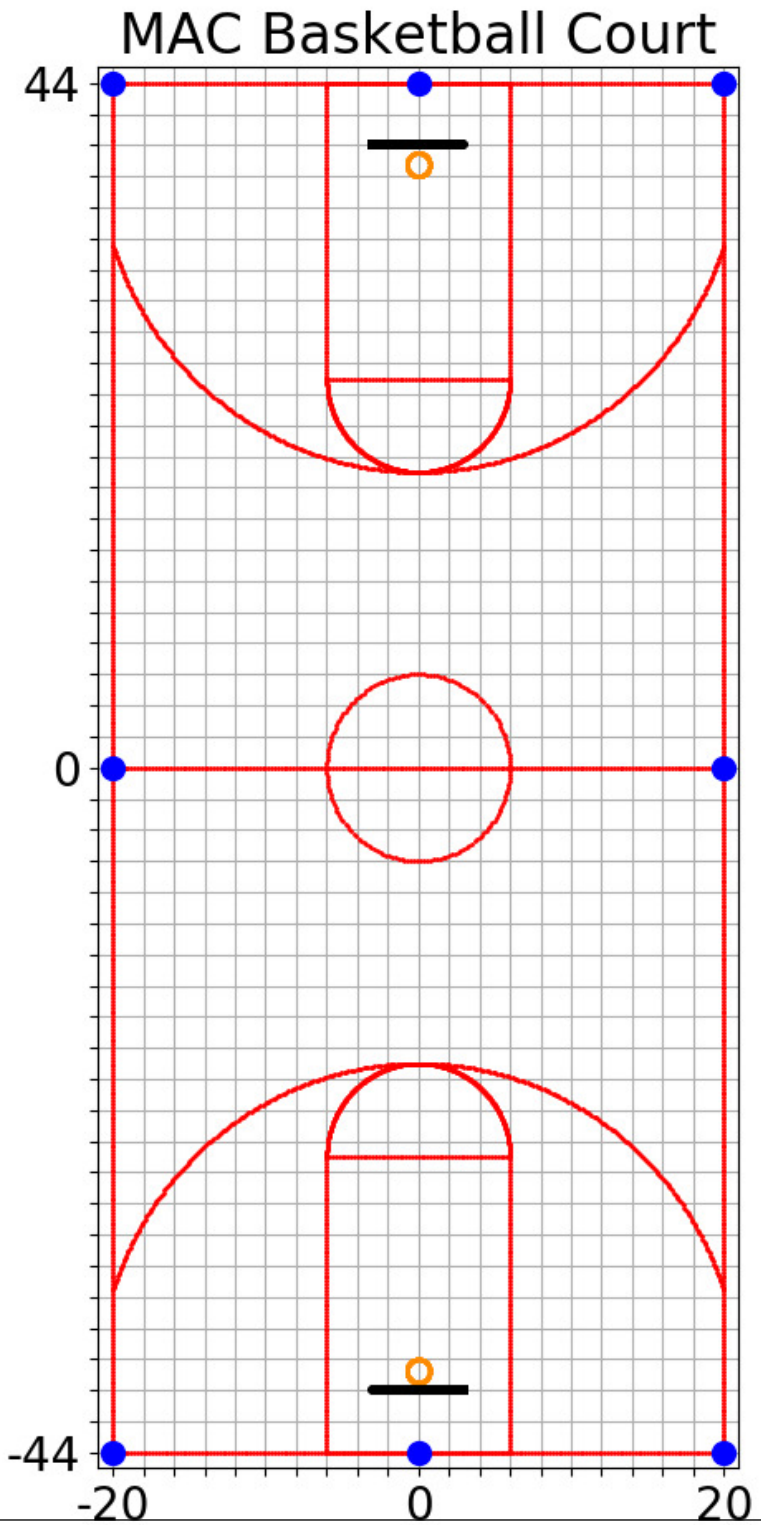
https://www.cse.unr.edu/~bebis/CS791E/Notes/CameraParameters.pdf

6 Calibration of Camera Position and Orientation

6.1 Visual Calibration

It is possible to confirm a camera calibration visually. The idea is straightforward, though as usual on this project, the devil is in the details. We know where various prominent visual features on a basketball court are located in the "world frame" coordinate system of the court. For instance, we know the court should be 88 feet long and that the rim should be exactly 10 feet over the ground. The idea is to produce arrays of 3D points on these objects, and then transform them to pixel locations. We can than "annotate" an image by overlaying the predicted pixel positions of visual landmarks on the image. If the calibration is correct, we will paint virtual lines on the perimeter, key, backboard, etc.

The Python file `basketball_court.py` computes coordinates for a series of visual landmarks on a basketball court including the perimeter, midcourt line, restraining circle at the center of the court, the "key", and the three point arc. Here is a plot of the court seen from above, i.e. of the XY plane. The backboard is shown in black and the small orange circle is the rim. The court is overlaid with a grid spaced at 2 feet. The eight target camera locations are shown as blue dots.


MAC Basketball Court
44
0
-44
-20
0
20

A careful observer will note that this figure does not quite match up with an NCAA regulation basketball court. This is not an error in the Python code! As mentioned above, it turns out that the court we used for our experiment at the MAC... is not a regulation basketball court. Not even close. It's far narrower than a regulation court. That's why the widest edge of the three point arc is practically flush against the perimeter line. With the benefit of hindsight, it would have been much easier for us if we'd located a court with standard dimensions. However we needed to do quite a bit of setup work, and it might not have been feasible to locate another court where the staff and players would be so accommodating of a science experiment encroaching on the athletic realm.

At the top of the file, global variables are set with various dimensions of the court. The functions `make_line`, `make_polygon`, `make_arc` and `make_circle` assemble arrays of 3D vectors of sample points on objects of these types. The function `make_court_lines` is the workhorse, generating both the lines plotted above which appear on the floor of the court, and also features corresponding to a vertical line from the floor to the base of the rim; the small and large white rectangle painted on the backboard; and the rim. In addition to these lines, the function generates a large collection of named landmarks (41 of them if you can believe it!). Coming up with descriptive names was about as challenging as getting their coordinates. Landmark names used were:

- `court_NW, court_NE, court_SE, court_SW` for the four corners
- `half_court_W, half_court_E` for the two sides of the half-court line
- `center, center_W, center_E, center_N, center_S` for the center and its four quadrants
- `key_N_NW, key_N_NE, key_N_SW, key_N_SE, key_top_N` for the key on the northern side of the court: its four corners plus the top
- `key_S_SW, key_S_SE, key_S_NW, key_S_NE, key_top_S` for the key on the southern side of the court: its four corners plus the top
- `backboard_N_TL, backboard_N_TR, backboard_N_BL, backboard_N_BR` for the four corners of the outer white rectangle on the northern backboard
- `backboard_S_TL, backboard_S_TR, backboard_S_BL, backboard_S_BR` for the four corners of the outer white rectangle on the southern backboard
- `backboard_in_N_TL, backboard_in_N_TR, backboard_in_N_BL, backboard_in_N_BR` for the four corners of the inner white rectangle on the northern backboard
- `backboard_in_S_TL, backboard_in_S_TR, backboard_in_S_BL, backboard_in_S_BR` for the four corners of the inner white rectangle on the southern backboard
- `rim_center_N, rim_center_S` for the center of the rim on each end of the court

Here is a visual demonstration of the calibration of Camera 3:

Camera 3 Visual Calibration

The red pixels painted in correspond to the projected locations of the key, three point arc, backboard, and a vertical line to the bottom of the hoop. The red plus sign represents the center of the camera and is intended as an aid to estimating a point z that the camera points towards. This picture gives us a sense of both what has been accomplished and what is left to do. The overlap between the projected pixels and corresponding landmarks on the photo is quite tight and a convincing demonstration that all of these calculations are basically correct. But, they're also not exactly right. We spent a lot of time manually fiddling the calibration parameters and weren't able to improve materially on this estimate. When we started the project, we had hoped we could automate this process, but doing the visual processing to extract the pixels corresponding to the landmarks has been too labor intensive for us to automate this calculation in the available time. Automating the calibration is one of our top areas where we would like to improve in the future if we continued working on this problem. Automating the camera calibration could be a good fit for machine learning or neural network techniques.

These calculations are carried out in the Python file `camera_calibration.py`. The function `plot_frame` creates a plot of a frame with suitable axes in preparation for annotation with pixels at visual landmarks. Two utility functions `annotate_frame_line` and `annotate_frame_dots` overlay

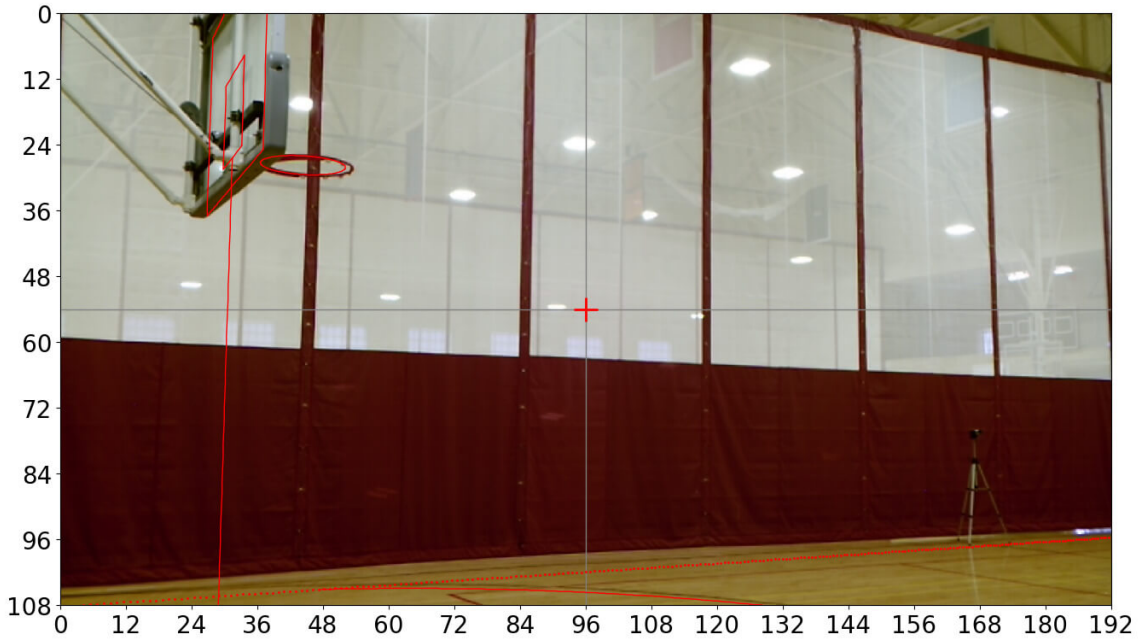
red lines on the images where the landmarks are estimated to be. The lines produce better looking charts, but plotting the perimeter can cause artificial lines to appear when matplotlib tries to connect pixel locations that are far off the screen together. All calibrations are performed against the median frame so we have a clean image without any obstructions for the calibration. Each camera has its own calibration function, e.g. `calibrate_cam3`.

Here are the visual calibrations for all the other cameras. Cameras are shown in symmetrical pairs to aid comprehension (hence the repetition of our favorite camera, Camera 3).

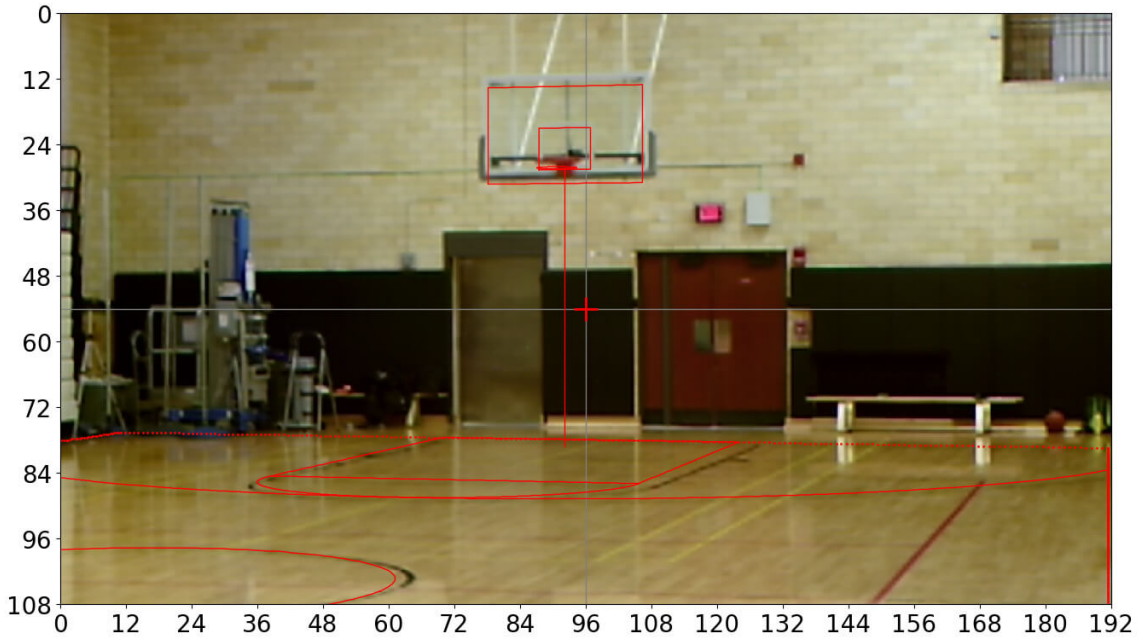
Camera 3 Visual Calibration

Camera 7 Visual Calibration

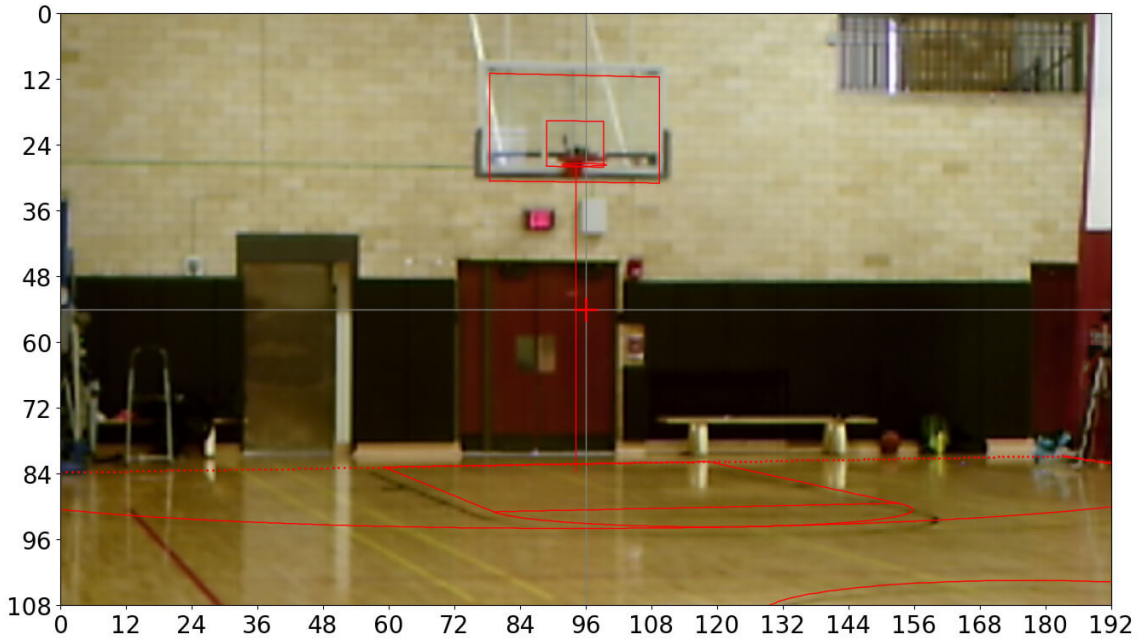
Camera 2 Visual Calibration



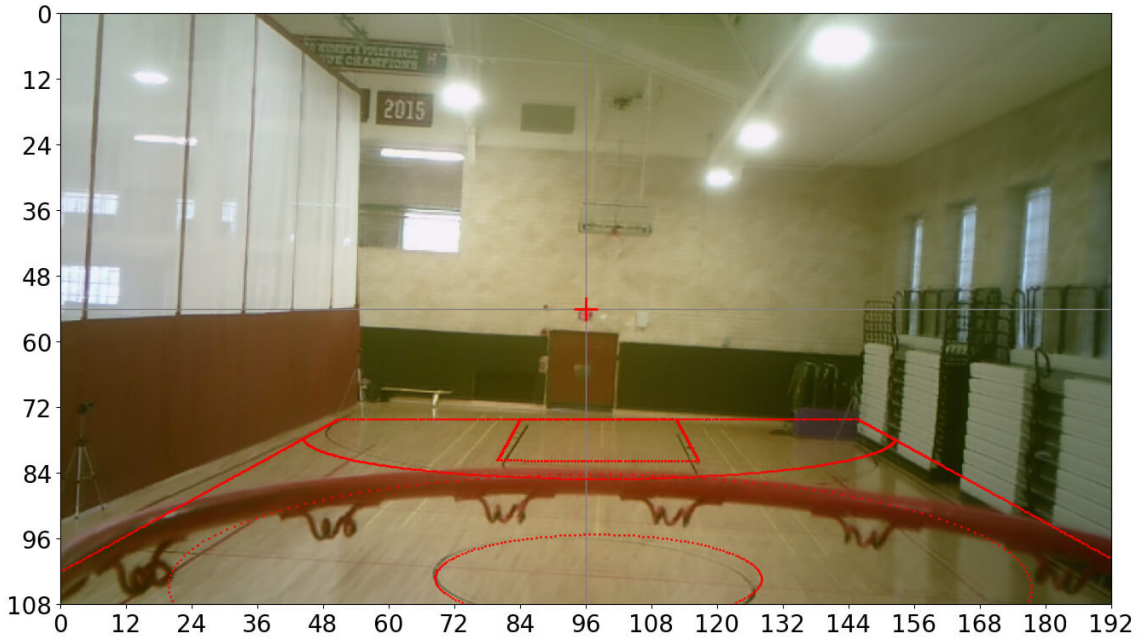
Camera 8 Visual Calibration



Camera 4 Visual Calibration



Camera 6 Visual Calibration



Camera 1 Visual Calibration

Broadly speaking, all of these except camera 1 might be termed "satisfactory" for the task of tracking a basketball in flight. As mentioned above, Camera 1 is not parallel to the XY plane in world space, so it requires an additional degree of freedom that we did not incorporate into our modeling apparatus.

6.2 Numerical Calibration

We performed an alternate camera calibration strategy of performing a numerical optimization to infer the location of each camera. Using the median frame for each camera, we identified as many visible landmarks as we could from among the 41 described above. For each of these landmarks, we used MS Paint to estimate the pixel location (u, v) corresponding to this feature. We then set up an optimization problem to estimate the camera's position, orientation, and zoom. The input parameters to the objective function were the position of the camera in world space (x, y, z); two spherical angles θ and ϕ describing where it is pointed; and the zoom. We used this alternate characterization from the aim point because the aim point has a superfluous degree of freedom. For a person looking at stills it is a useful interface, but for an optimization engine it would only add unnecessary confusion.

These calculations were carried out in the file `camera_calibration_num.py`. The function

`make_objective_func` is a factory. It takes an inputs a collection of landmarks, accepting both their world coordinates (x, y, z) and their apparent pixel coordinates (u, v). The objective function takes as inputs the six parameters $(x, y, z, \theta, \phi, \text{zoom})$. It builds a transform; applies the floating point pixel transformation to `landmark_xyz`; and compares the results `calc_uv` to the observed pixel locations `landmarks_uv`. The scalar valued objective function is just the sum of squared error on the pixel calculations.

The function `calibrate_camera` performs the calibration on a camera. It gets the visual calibration function to produce diagnostic plots similar to those used to solve for the visual calibration. It also gets the visual calibration settings. The landmark data is populated on tables below and the function looks them up. It instantiates the objective function and minimizes it using `scipy.minimize` with default settings. The initial guess is the visual calibration.

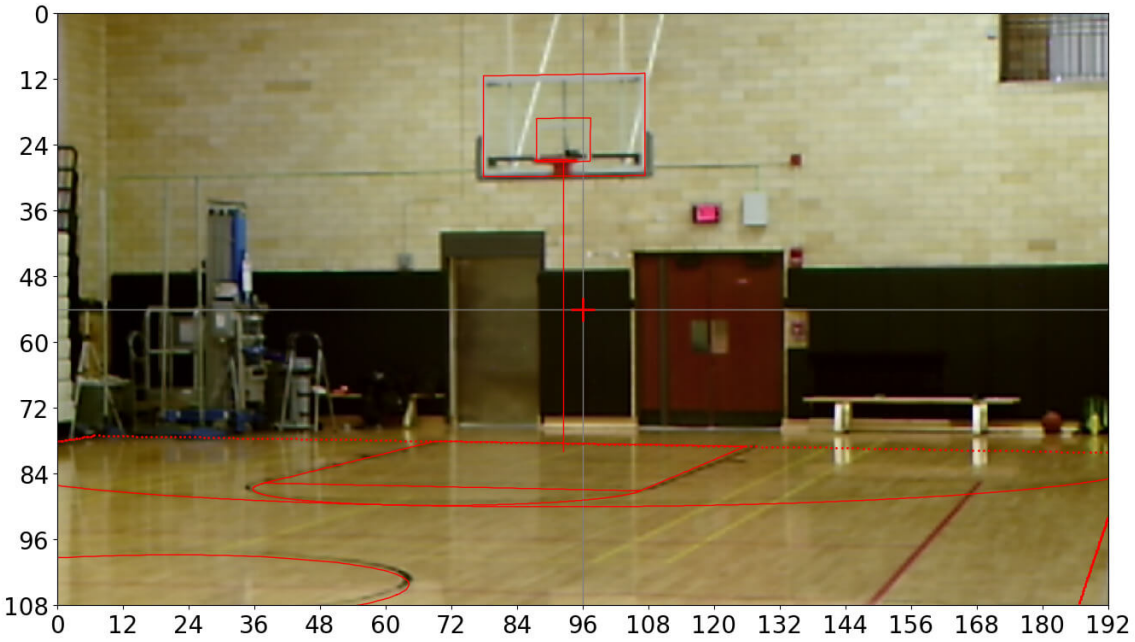
In addition to displaying annotated plots, it also outputs its score and how much it was able to improve on the visual calibration. Reported data includes the old and new camera position; the change in the implied angle θ and ϕ; and the change in score. The score is also reported as an RMS error in recovered pixel values, which are relatively easy to interpret.

At the bottom of the file, all of the landmark points are keyed in. This turned out to be quite time consuming. It's not complicated, but it takes a lot of attention to detail and it's easy to mislabel the points. This is where having a sensible naming scheme really helps.

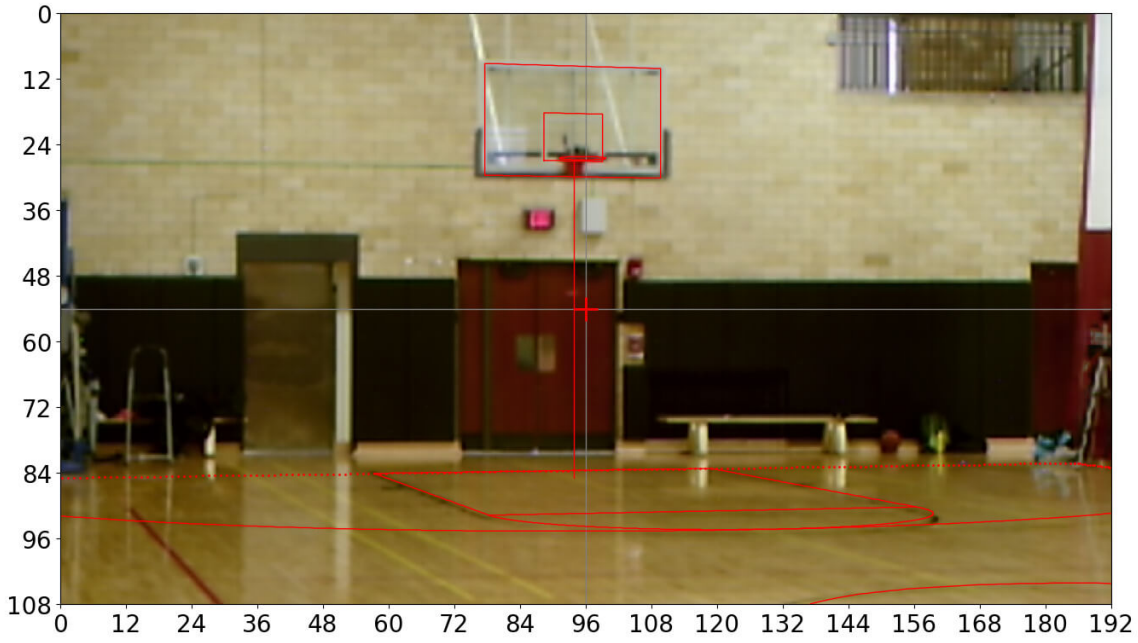
The results of the numerical calibration were very solid. Interestingly, they were only a little better than the results from the visual calibration. As an example, the numerical calibration for camera 3 moved it from (20.0, 0.0, 5.2) to (19.38, 0.15, 5.180. It reduced the RMS pixel error from 7.845 to 7.532. Our best guess is that most of these errors were some kind of misspecification (e.g. wrong measurements or pixel positions), not numerical errors.

Here are plots for three cameras:

Camera 3 Numerical Calibration



Camera 4 Numerical Calibration



Camera 6 Numerical Calibration

7 Tracking The Ball in Pixel Space

This project has proven to be considerably more difficult than we originally planned in several phases including manipulating the video, synchronizing the audio, and calibrating the camera positions and orientations. All of these obstacles proved surmountable, albeit at the cost of time that was budgeted for the later stages of the project. The one obstacle that has so far completely stumped us is **efficiently** tracking the position of the basketball in the frames where it is visible, i.e. extracting the (u, v) pixel coordinates of its center (and possibly the radius as an additional quantity). William has labored mightily trying a number of different approaches on this. The software he has used in his lab has failed completely on this problem. While we don't know exactly why, a good surmise is that the ball is moving too much between frames. These systems are typically used in conjunction with high end, laboratory photographic equipment capable of shooting video at much higher frame rates.

An ironic aspect of this challenge is that tracking the motion of the ball is a trivially easy task for a human. It's an excellent example of the kind of "natural perception" tasks at which humans excel in comparison to computers. As an aside, we are in no way claiming that a properly calibrated computer vision system cannot track the position of the ball on these frames. We're only saying that as a team of people with general scientific training and no particular background in computer vision, this was a task that looked easy from afar and proved very hard for us at least.

With the benefit of hindsight, we made a tactical error when we set the video recording mode to full HD (1920x1080). When we made this decision, we did not realize it would have the effect of reducing the frame rate from 30 FPS to 15 FPS. To add insult to injury, while the images nominally have a high resolution, the optics and sensors in this camera are not nearly good enough to render the scene crisply, so the extra pixels are effectively wasted anyway. This leads us another
Lesson Learned: Don't trade frame rate for resolution.

After we failed with the software in William's lab, we researched possible library solutions for object tracking. The strongest possibility appeared to be OpenCV, which offers 8 different algorithmic implementations of object trackers. We spent approximately four hours setting this up and experimenting with all 8 algorithms. The Python file `object_track_cv` is not a useful finished product in our code base. It's the test harness we used to interactively experiment with the different object trackers in OpenCV. The OpenCV front end is very impressive and easy to use. The program above allows you to select an initial bounding box using the mouse, and the OpenCV tries to track the object. When it notes a detection failure, it waits until a polling interval is elapsed (e.g. every 1 or 5 seconds) and then asks on the terminal whether the ball is present on the screen. If yes, the user enters a new bounding box and the process continues. This program would have offered a very handy semi-automated way to quickly annotate the ball position, if any of the OpenCV image trackers worked even a little in this scenario. Watching them flail would have been funny if it wasn't

such a hassle. The interactive experience of trying these different detectors and seeing their failure modes was useful in developing an intuition for how these systems work. They do much better with slowly moving objects, i.e. objects where most of the pixels move just a little bit from one frame to the next. Our best guess is that if we had been able to film at a high frame rate like 60 FPS, one or more of the OpenCV tools would have worked at least in semiautomatic if not in full automatic mode.

After exhausting over a week attempting to implement an automated workflow to annotate the pixel coordinates of the ball, we abandoned the effort approximately 72 hours before the project deadline and devised a new plan to prove out as much of our remaining plans as possible in the remaining time. We opted to hand annotate the center of the ball over all 4,391 frames for each of the 7 cameras. After doing this, we identified several shot attempts, to focus on with our test code before attempting to fully reconstruct the path off the ball for the entire game.

While the hand annotation was painful and laborious, the data has the potential to be very useful for future work. The annotated data could be used in future AM205 problem sets, or for training object trackers. If we had more time, we could have used the hand annotation to compare the performance of the OpenCV motion trackers, or to train our own custom motion tracker and evaluate its performance. While not fun, the hand annotation process was also a reminder that in some tasks, humans still beat machines, albeit very slowly.

8 Tracking The Ball in World Space

The technique to track the ball in world space is quite similar to the approach in numerical camera calibration. Given a guess as to the ball position, we can compute the implied position (u, v) of the ball in pixel space on each camera using the transformations fitted during camera calibration. From the pixel tracking, we have a list of cameras where the ball is visible, and for those cameras, we have the known pixel positions (u, v). That is, we sum the squares

$$(u_{\text{calc}} - u_{\text{obs}})^2 + (v_{\text{calc}} - v_{\text{obs}})^2$$

over all cameras where the ball is visible. We ignore the cameras where the ball is not visible, and set up an optimization problem in which we solve for (x, y, z) to minimize the sum of squared error in the projected pixel positions.

These calculations are carried out in the file `ball_track.py`. The top of the file sets up global variables and loads the data containing the pixel locations. The function `world2pix` takes as inputs the 3D position of the ball and a mask indicating which cameras see the ball. It returns an n x 2 matrix of the calculated pixel positions for only the visible cameras.
The function `make_difference_func` is a factory. It returns `difference_func`, which will compute the difference between the projected and actual pixel pixel values on the visible cameras.
The function `make_objective_func` is a factory function that returns an objective function ready to be optimized. The inputs are the observed pixel locations of the ball on the visible cameras and a mask indicating which cameras are presented. This function just uses the difference function computed above and then returns the sum of squared pixel error.

The function `plot_frame` makes a plot of one frame of the video that is ready for later annotation. The function `frame_overlay` overlays a red circle indicating where the simulation thinks the outline of the ball should be on that camera based on the solved position of the ball and the camera calibration. We've already recovered the estimated center of the ball. An additional calculation is done to estimate the pixel radius of the ball. This is a simple linear function depending on the radius of the basketball, the distance from the camera to the ball, and the camera parameters (focal length, zoom, and pixel size). The function `plaot_tableau` in analogous to the one used to make the unannotated tableau frames. It assembles six frames into one large frame, allowing a viewer to see all the angles at once. The function `frames_overlay` overlays all the frames for a given frame number. If the ball is not visible on that camera, the frame is left unchanged. The function `save_ball_figs` saves the individual annotated frames for each camera and the annotated tableau frame.

The function `track_frame` is a driver that does all the steps in order to track the ball and publish the results. It looks up a mask of which cameras were visible and the observed pixel positions on those cameras. It uses them to create an objective function measuring squared pixel error. It

performs an optimization using `scipy.minimize`. The initial guess was chosen to be a "neutral" position for a halfcourt basketball game. It was the top of the key at an elevation of 5 feet, which is half way between the floor and the rim. This point has coordinates (0, 26.0, 5.0) in feet.

In addition to saving the annotated frames, this function saves the data including the frame number, time, and solved for x, y, z coordinates. Each frame is saved to a separate tiny CSV file to allow for easy parallel processing.

The function `track_frames` accepts a list of frames to be processed and a flag indicating whether to display a progress bar. This is used by the main driver of the program. A good amount of the effort in writing this program was in carefully handling details so it could parallelize well. The approach taken here has a few simple ideas. First, the program does not process duplicates; if you want to re-run something, you can delete it. A centralized statistics file with solved positions at each frame is maintained. This aids coordination between multiple computers handling different batches at once. In addition, the inner functions check if an image file exists before trying to calibrate a frame and then overwrite it. The actual parallel processing is done using the `joblib` library and is quite simple. It uses the `Parallel` and `delayed` facilities in `joblib`.

The Python script `ball_stats.py` assembles all of the CSV fragments with calculated positions into one Pandas DataFrame. This data frame is also used to avoid processing duplicate frames.

Annotating all of these frames was a very large computational job. It took approximately 15 hours of total wall time, with jobs running on two computers nonstop except for pauses to check progress and correct errors. This job ran in parallel on 64 threads on both a high end deskop PC with an AMD Ryzen 2990-WX processor (32 cores, 64 threads) and the Linux server mentioned above 40 total cores (dual Intel Xeon CPUs each with 20 cores).

8.1 Visualizing the Results

Here are a series tableaux illustrating a medium range jump shot that had good flight and visibility on multiple cameras.

Frame 880: About to Shoot

Frame 890': Releasing the Ball

Frame 900: Ball in Ascent

Frame 910: Ball near Apex

Frame 920: Ball in Descent

Frame 930: Approaching the Basket

Frame 936: Basket!

Frame 940: Continuing Past the Basket

This sequence of stills shows clearly that our 3D motion capture analysis was essentially correct, albeit with some possible time lags due to the discretization of frame times. We are moderately confident that if we replicated this setup with better cameras and a higher frame rate that we could get sufficiently accurate motion tracking to predict whether a shot attempt would go in.

We thank our patient reader for tolerating so many pages. It is much easier (and more fun) to absorb this information by watching a video. Here is a link on YouTube to a video we made of the tableau: Motion Capture Video It's watchable, but the quality isn't nearly as good as watching it locally. The finished videos with the annotated circles are in the root of our Dropbox folder.

9 Acknowledgments -Michael

I would like to thank my teammates Nicholas and William for their steadfast dedication to this project even as it exploded in difficulty and complexity. I proposed the idea thinking it would be far more manageable. I am grateful for their hard work in the face of unexpected challenges.

I would also like to thank my wife Christie Lee Gibson who provided invaluable technical assistance in video editing with Adobe Premier and showed tremendous patience with my temporary abdication of family responsibilities in the pursuit of this project.