ME/CS/EE 134 Final Report

Gabriel Aguiar, Naci Keskin, Andrew Pasco, Allen Yang

March 14, 2024

Task and Basic Concept

Our objective was to have our robot play a game of backgammon against a human opponent. This would involve the robot (a) using a camera to detect the state of the board (where all of the checkers are), (b) picking up and placing the checkers at the appropriate locations, (c) matching the state of the board against the robot's internal expectations of what moves are possible and where pieces should be, and (d) switching the turn marker to the opponent's side and waiting for the human to make their moves.

If, at any point, the robot encounters a situation that does not match its expectations, it will attempt to recover to a state that will allow the game to continue. This includes the following cases.

- The robot failing to grab a checker, in which case it will try again if the checker remains on the board.
- The human making an illegal move during their move, in which case the robot will return the checkers to the state they were in at the end of the robot's most recent turn.

The main challenges of this design are the coordination and game logic. There are many cases that need to be handled with care, and each of them requires a different set of actions by the robot to be properly handled. Ensuring that the robot responded appropriately to all of the situations it could encounter took up a significant portion of our time.

Mechanism and Hardware

Overall Arm Design

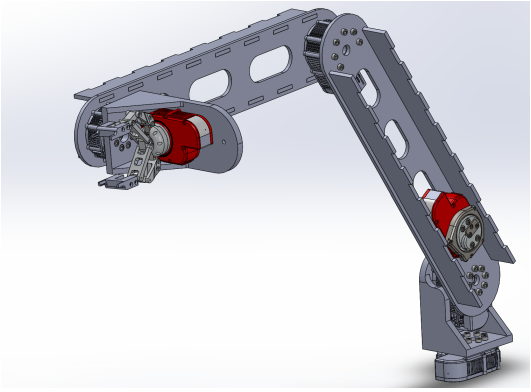
A five degree of freedom (DOF) arm was designed for the specified task of playing backgammon. We determined the need for five DOFS firstly because we wanted to manipulate objects both from a "handshaking" wrist position (for grabbing the dice cup) and a "picking" position (for grabbing the checkers and dice). Though we did not end up needing to grab the cup, we still utilized the fourth and fifth DOFs to ensure that our gripper had both the appropriate pitch position to remain parallel to the table and the appropriate roll (more like yaw when pitched down and facing the table) position to ensure we only grabbed the intended checker without conflicting against nearby checkers.

Both links were 450mm long. The equal link length was determined because we wanted to be able to utilize only a single work table and grab checkers on the near side of the board, very close to the base of the robot. Because the checkers we manipulated were of negligible weight, we decided on a laser-cut wood link design with a box structure (outer flange not pictured) for added torsional rigidity.

Motors were selected as shown in the table below, according to the computed maximum static moments seen at each of the joints.

Joint Name	Maximum Static Moment (Nm)	Chosen Motor
Base	0	X5-9
Shoulder	12.5	X8-16
Elbow	4.75	X8-9
Wrist (pitch)	0.3	X5-9
Wrist (roll)	0	X5-9

Maximum Moments and Chosen Motors for Joints



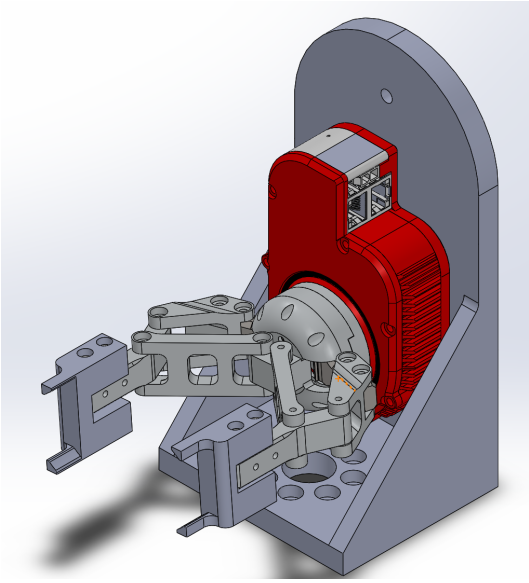
Full arm CAD model

Gripper Design

The initial gripper was designed to grab checkers, dice, and a cup. To handle all of these, it was larger and thicker, which meant that it was more likely to bump into other checkers when attempting to grab one.

Once we abandoned the dice/cup in favor of virtual dice rolls, we accordingly modified our gripper to be more targeted toward grabbing checkers.

The gripper fingers were mounted on a HEBI COTS parallel jaw gripper, screwed directly into our final wrist roll motor. The gripper is articulated with a spool motor mounted 10cm from the shoulder motor to minimize the shoulder moment.



Wrist CAD detail, final gripper design

Basic Robotics

Task Coordinates and Inverse Kinematics

Our task coordinates includes the standard x, y, z coordinates of the tip, as well as the tip's orientation. This orientation includes rotation about the wrist (relative to orientation of the base) and the pitch of the wrist (absolute, where pitch = 0 means level with the table).

There is an additional `GripperTask` which controls whether the gripper is open or closed.

Though we implemented a joint spline, we did not end up using it in our final demo.

Our inverse kinematics implementation uses a singular value decomposition with $\gamma = 0.1$. From the singular value matrix, S, we construct the following (where $s_i \in \text{diag}(S)$):

$$S_{\text{inv, i}} = \begin{cases} \frac{1}{s_i} & |s_i| \geq \gamma \\ \frac{s_i}{\gamma^2} & \text{otherwise} \end{cases}$$

We then construct $J_{\text{inv}} = V^\top S_{\text{inv}} U^\top$ and use this to find

$$\dot{q} = J_{\text{inv}}(v_d + \lambda \cdot e)$$

and

$$q = q_{\text{prev}} + \dot{q} \cdot dt.$$

Our gravity compensation used the following equations:

$$\tau_{\text{elbow}} = -6 \sin(-q_0 + q_1) - 0.3$$
$$\tau_{\text{shoulder}} = -\tau_{\text{elbow}} + 8.8 \sin(-q_0)$$

Notably, we found that including a constant term in our gravity compensation improved our robot's stability.

Trajectory Management

Our trajectory chaining was managed by a `TaskHandler` object, which stored the robot's joint and task positions and velocities for smooth transitioning between splines. We defined six different types of `TaskObject`s, each of which represented a single spline in the trajectory queue. The `InitTask` simply returned the robot to its wait and watch positon, which cleared the arm from the view of the camera and space of human players via a joint spline to known joint coordinates. The

`TaskSplineTask` represented a single task spline from the robot's current position to a requested final position in task space. This utilized the inverse kinematics implementation described above and also was parameterized by a given final direction for the velocity in the y and z directions (which was helpful for placing checkers vertically in a row and chaining trajectories together with a specified continuous velocity). The `GripperTask` was a simple joint spline for only the gripper motor, while holding the other joints stationary. It was parameterized by whether the gripper was opening or closing, as well as the type of game piece, though we only used the checker grip. The `CheckTask` was only utilized after picking up a checker, and continued the current velocity from the previous task spline. This task was distinguished to serve as a sort of flag to indicate that the effort value from the gripper should only be read at that time. The `WiggleTask` was a task trajectory which executed a wiggling movement to adjust before picking up a checker. Finally, the `WaitTask` simply held the joints stationary for a given amount of time.

While our `TaskHandler`'s "atomic units" were these six `TaskObject`s, the game of backgammon, as implemented, is played with only a single, abstracted command: pick up a checker from x and place it at y. Thus, this behavior was abstracted into a single movechecker() function, which executed a checker move from a given source position to a given destination position. From these positions, the function determined the sign of the final velocity of different segments of the spline, computed required gripper angles to ensure there would be no conflicts or collisions, and appended a chain of `TaskObject`s to the trajectory queue.

Detectors

Interactive backgammon playing required a number of detectors. These can generally be divided into board detection, which was expected to be much more stationary so only had exponential location/size filtering, and checker detection, which required temporal filtering and correspondence filtering to dynamically respond to changing positions. A dice cup detector, dice roll detector, and tip-adjustment algorithm were also conceptualized, but these were not implemented.

Board Detection

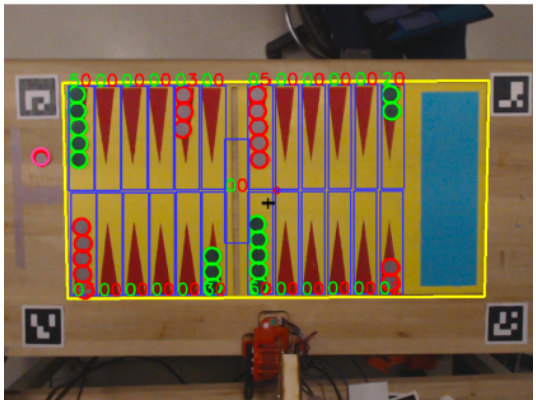
The overhead camera ran at 15Hz, meaning a new message from the raw image topic was received at the same rate. Although this refresh rate was helpful for checker detection, we only updated the board belief (and camera/world transformation from the fixed ArUco markers) at a much lower 1Hz. Basic detection was very simple, as a binary mask could be easily created from filtering out any image pixels not in the HSV range of the yellow, blue and red paint on the board. After further processing via binary pixel erosion and dilation, the largest contour was transformed into xy coordinates. Then, a minimum area RotatedRect was fit around this contour. This object held the center, dimensions, and rotation angle of the detected board as a single measurement, which was exponentially averaged into the current board belief. Notably, we rejected measurements if the area was more than 500cm^2 different, and we forced the expected aspect ratio by converting the measurement if a measurement ever contained a 90 degree offset. The area filtering was important so our board would only be altered if a clean view, unobstructed by possible hands or bodies, was present. Given the detected location and orientation of the board, locations of its relative features were known, so the positions of 25 "buckets" for each triangle and the bar were also computed given the detected board, for use in sorting checkers.

Checker Detection

Checkers needed to be detected based on their colors and features, then filtered for existence and correspondence, before finally being sorted based on their positions. Similar to the board, the first step of processing was considering only those pixels in the HSV range of the green and purple checkers. Then, only the pixels within the detected mask of the board from before were kept, which helped to remove any noise from the table or surroundings. After a few cycles of dilation and erosion, only the checker shapes remained, though when stacked in lines as they appeared in the game, they often took on a "snowman" shape. The OpenCV circle detection

algorithm HoughCircles was used to resolve the individual circles from these "snowman" stacks, with parameters for the minimum and maximum radius, circularity, and minimum distance between checkers tuned for consistent detection. Finally, any particular checker circles detected in a given frame needed to be matched against the current belief of where the checkers were on the board. Thus, existence and correspondence filtering using a log-odds representation was utilized. Each new frame of checkers would contribute to the existence belief of an already detected checker if less than 2cm from it, (and the newly detected position would be exponentially averaged in) or be initialized with 50% belief of existence if not near an existing checker. Any checkers present in a previous frame but not detected in the current frame would have their existence belief decreased, and any below a certain threshold (defined as two frames of non-existence after initialization or 15 frames / one second of non-existence if previously at the log-odds cap of 4.0) would be removed from the list. Finally, the checker positions were sorted into the previously determined board "buckets" by these filtered positions only if their log-odds were above 2 (probability above 88%).

Though not a checker on the board, the turn signal also needed to be detected so the robot could determine whether it was its turn and how to pick and place the turn signal to indicate a change of turn. The exact same detection and existence filtering algorithm was utilized, with the caveat that only one checker was allowed to exist. The turn was determined simply by the y-coordinate of the turn signal, which would be larger than 0.4 if it was the robot's turn and less than 0.4 if it was the human's turn. Usage of the turn signal made for a clear indication to a human player when it was appropriate and safe to make a move, and when the robot would be moving or fixing the board.



Board and checker detection

Software Architecture

The robotic system consists of three main nodes.

Detector Node

The Detector node was continuously subscribed to the raw camera feed from the top view camera. Each time a camera frame is received, the detector node tries to save the number of checkers of each color at each "bucket" (24 triangles and bar) as well as the number of checkers of each color that weren't assigned to a bucket as the game state. If the detected game state has the right number of checkers, then the detector nodes saves and publishes both the game state and the location of each checker organized by bucket and color. Even though the game state and checker locations are nested lists without fixed shapes, we didn't want to deal with custom ROS messages, so we dealt with this by publishing the flattened lists and reconstructing them in the other nodes. The game state message was easy to reconstruct, as it did have a fixed size (# of buckets, # of colors), but the checker locations size was dependent on the game state. Thus, for the checker location message, the detector node prepends the flattened game state list to the flattened checker locations list before publishing, making it easy to reconstruct the nested structure in other nodes. Furthermore, to aid with understanding the state of the board, the Detector node also publishes the detected position of the board so that the position of the triangles and the bar, which are known relative to the board, can be converted to a global position. We were originally planning on trying to detect who's turn it is by looking at changes in dice state, but since we didn't implement the dice detection, we used another checker as a turn signal. The position of the turn signal was sent to both the Trajectory node and Game node as well.

Game Node

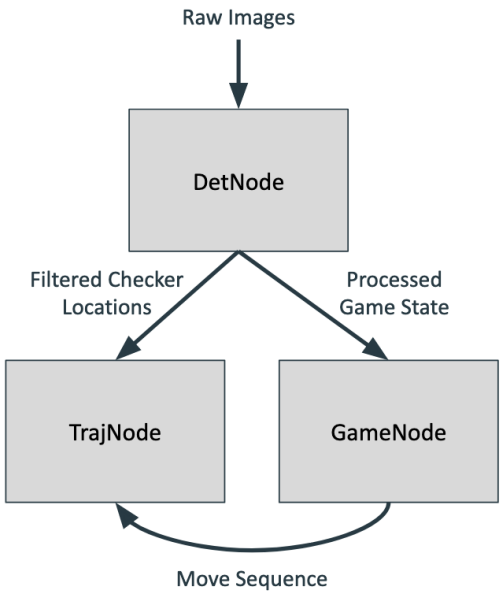
The Game node handled the high level playing state of the robot during the game. The main goal is to generate a set of moves encoded by just [source bucket, destination bucket, color]. To do so, it subscribes to the Detector node's published game states, which allows the Game node to always have a real time accurate representation of the current game state. Further, it contained a "Game object", which essentially simulated the game to give the robot an expected state to compare the actual state to. This gives us a "ground truth" to compare against that allows us to detect and alleviate possible errors that might happen during execution and the use of the Game object was the core of our failure recoveries.

Whenever a given move is completed by the robot, the Trajectory node signals to the Game node that it is ready to determine its next action. Here, the Game node determines its next course of action from a variety of factors. If the Game node hasn't received a game state in more than 3 seconds from the Detector node, the Game node determines that the game state that is currently stored is outdated and waits in a position that doesn't obstruct the board to aid the detector in sending an updated, correct game state. If it's the first turn and it detected that the turn signal was in the human turn position, the robot moves the turn signal to the robot turn's position. This is because the game defaults to having the robot move first. Then, if the robot detects it is its own turn but the expected turn is the human turn, then the robot has most likely missed a checker and the Game node compares the current state of the board to the expected state of the board, and if they're different, it computes a set of moves using a stack algorithm that "fixes" the board, including the turn signal, and publishes them for the Trajectory node to execute. If checkers need to be replaced (if there is a green checker where there should be a brown one or vice versa), the algorithm moves those checkers off the playing surface first and then executes the stack algorithm. Note that even if one checker is wrong, the robot was scan the entire board and fix all errors on the board. If the state of the board and the turn signal is correct, then the Game node raises a flag to indicate that it is ready to determine an action.

The determine action function runs on a set timer, and this function essentially runs the game. If it is the human's turn, this function doesn't do anything and just waits. If it is the robot's turn, which means the human just completed a turn, the function first determines if the current game state is legal based on the previous game state and the dice roll using a back tracking algorithm. If it isn't, the robot uses the same fixing algorithm to determine a set of moves to turn the current game state to the last expected game state (state before human move) and prompts the human to perform another move using the same dice roll. If the human move was in the set of legal states, then the Game node will re roll the dice (RNG), generates a set of valid moves from the current game state, randomly selects one, and sends the sequences of moves to the Trajectory node to be executed.

Trajectory Node

The purpose of the Trajectory node was to handle the movement of the robot. It received the location of the checkers and the location of the board from the detector node, as well as sequences of moves from the Game node. The positions of the triangles and bar were hardcoded with respect to the position of the board, so every time the Trajectory node receives a board location from the Detector node, the Trajectory node updates and saves the global positions of these buckets. The location of the buckets, as well as the location of all the checkers within these buckets gives the Trajectory node the ability to compute the positions of the checkers to pick up and where on the board to place the checkers given a set of moves. When the robot is bearing off, it chooses a random location in the free space on the board continuously until it finds a location without checkers in that region. The robot trajectories are then handled with a Task Handler object, which essentially just manages the various physical tasks the robot might perform (task splines, joint splines, gripping, waiting, etc).



Basic diagram describing the software architecture of the robot

Failure Recoveries

The main failure recoveries were already touched on in the software architecture section. The robot will correct inconsistencies between the actual and expected game states and generate fixes as well as determining invalid human moves.

Failure to Pick Up Checker

The Trajectory node was able to detect whether it had failed to pick up a checker by subscribing to the /joint_states messages from the HEBI node and checking the effort of the motor that controls the gripper. Intuitively, if the gripper has a checker, then the effort the motor has to exert is higher than the effort if the gripper doesn't have a checker. We utilized this fact and we were able to reliably detect whether it missed a checker. When a missed checker was detected, the Trajectory node would end its current task, clear the task queue, and go back to its initial position to reassess and fix the board.

Fix Board

The fix board method in the Game node compares the actual vs expected states of the board and generates a sequence of moves that turns the actual state to the expected state. If none of the moves require replacing a checker, the algorithm iterates through each bucket, comparing the actual vs expected number of checkers, and using a stack to keep track of how many extra/missing checkers there were for that bucket. If the bucket had missing/extra checkers, then the algorithm simulated a "fix" by popping the bucket number to transfer from/to from the stack and combining it with the current bucket to generate a move. If some of the moves do require replacing checkers, then the algorithm first generates a sequence of moves that move all checkers that need to be replaced off the playing surface (similar to bearing off), and then this board state then reduces to the first state and the same algorithm is applied. This works because bearing off places the checker in a random open space off the playing surface independent of the color, which makes it easier for the Trajectory node to handles moves to and from the bar, while moving checkers onto triangles or the bar has stricter requirements.

Legal States

The legal states function was a crucial part of our code in checking that whether the user played a correct move in their turn. Right before the robot turn, we wanted to check the state of the board in case the user tried to cheat or accidentally

made an incorrect move. To do so, we have implemented this function to catch if an illegal state is present after the latest turn of the robot. Initially, we take the current state of the board and the state right after the robot's last turn. The function compares the two states and takes note of the changed rows in four categories. Rows with increased purple checker count, decreased purple checker count, increased green checker count, and decreased green checker count. In the human player's turn, either the player can't move any checkers, or the player moves some purple checkers and moves some green checkers as well as a result of a hit move. Thus, we check checkers' movements from rows with decreased purple checker count to increased purple checker count. If the counts do not match, the function returns False indicating the played state in human turn was not a legal state. Similarly, this function contains more conditions like this to check when a legal move is not played. Instead of checking every possible state that can be achieved after the robot's turn, we try achieving the final state by checking conditions and moving the checkers accordingly from the previous state. So on top of the counts, we check if all the moves are possible and if we match the final state. In the end, we were efficiently able to check if the human played a correct turn among all the possible moves they could make.

Shortcomings or Outstanding Items

One of the biggest outstanding items was utlizing the tip camera. The tip camera and Detector node were set up so that it could be used, but because we didn't implement the physical dice rolling, which was the original main use of the camera, we also didn't implement the tip camera. Even without the dice roll, the tip camera could have been used for several things, including dynamically adjusting the mapping between detector coordinates and world coordinates so we could use less Aruco markers and fixing/detecting errors that are hard to detect with the over head camera (stacked checkers, etc...).

Even not considering the tip camera, our current robotic system still has several limitations. Right now, since the triangle and bar locations were hard coded with respect to the detected board, the number of checkers in each triangle and bar was limited to 6. Otherwise, if more than 6 checkers were placed in a triangle, the extraneous checkers would exceed the hard coded bounds of the triangle and bar locations and be detected as a part of the opposite triangle. Further, the board detection can have some noise, which means hard coded triangle and bar locations can cause slight inaccuracies in placing checkers in triangles. For example, the detector might not see the board as a perfect rectangle, and giving it a bounding box might shift the boards starting position, which would also shift the locations of the buckets being used by the Trajectory nodes. In the early stages of detection, we were able to develop a relatively good mask for detecting the triangles that didn't have many checkers on them, and we could also use the fact that checkers placed on the same triangle would be clustered together. Combining these two approaches, we might've been able to develop a detection algorithm (even combined with the hard coding) that dynamically detects the location of the buckets.

Also, the way that the fixing algorithm is designed right now makes it difficult to generate fixes for game states where checkers of different colors are on the same triangle. The fixing algorithm could be tweaked to take care of this, but maybe a new move type where the robot removes gaps between checkers could also be introduced in the Game and Trajectory nodes. This would be necessary, because of a triangle has 2 green checkers, then a brown checker, then 2 more green checkers, even if the algorithm detects that it has to move the brown checker away, that would leave a gap in that triangle, which is not ideal. Then, sending the Trajectory node an encoded message letting it know that it needs to remove the gap can solve this problem.

One thing that would be harder to fix is the fact that the Detector node requires itself to see exactly 30 checkers before sending an updated game state. Right now, if a checker is missing from the board for too long, the robot just waits until it sees the missing checker. If the checker is actually missing, this approach makes sense. But, if a checker is stacked, or really close to another checker of the same color, and the detector can't distinguish between the two checkers, then there exists a way for the robot to recover from this state. One possible solution without using the tip camera is just detecting where there might be missing checkers and just moving the robot gripper around in that area, hoping that it would either separate the checkers or knock the stacked checker off.

Finally, one of the minor errors we could have fixed if we had more time was the fact that the robot sometimes recognized legal human moves as illegal, especially when they were complicated and involved non-normal moves (hits and bearing off). We think the legal states method was very close to working completely, but it just needed tweaking in maybe one or two edge cases to be fully functional. We believe if we had another week, this recovery mode would have been fully functional.

Thank You and Video

Though we had to reduce from our initial scope, we are very happy with the final product and demo performance. Thank you to Günter and all of the TAs for your hard work and assistance throughout the term helping us to reach this conclusion! Please view our video at this link!