Tic-Tac-Tobot

[EECS 149/249A Class Project]

Patrick Scheffe
p.s@berkeley.edu

Nikolas Alberti
nikolas.alberti@berkeley.edu

Department of Electrical Engineering and Computer Science
University of California
Berkeley, CA

ABSTRACT

1. INTRODUCTION

Our charter defines the goal of this project as follows:

"In this project, we want to build a robot that will challenge you to play a game of tic-tac-toe. [...]The robot will control the movement of a whiteboard marker using two servos. The pen will be set up in a construction which has a shoulder and an elbow joint, so it can be moved to a defined x-y-position. The whiteboard is mounted in a fixed position next to the manipulator. First, the robot draws the playing field consisting of nine tiles. A camera will be used for monitoring the player's move (placing a circle or a cross). Autonomously, the robot will react and make its own move. In order to make space for the human player's moves, the whole construction should be able to move away from the field."

This definition specifies well what the robot should be capable of doing. Accordingly, in order to meet the specification, a robotic manipulator must be assembled and controlled. Moreover, a computer vision component must be setup. At the end, everything must work together governed by a program that keeps track of the game and is able to compute own moves.

2. HARDWARE

2.1 Arduino Uno

The Arduino Uno is a microcontroller board equipped with an Atmel ATmega328P microcontroller. Arduino is an open-source hardware and software project originated in Ivrea, Italy. The hard- software have been distributed by the founders of Arduino (known as Arduino LLC) and the producer of the official Arduino boards (Arduino S.r.L.). The goal of the Arduino project is to make the development on Embedded as accessible as possible. For instance, the Arduino IDE, completely hides technical details like header files from the user.

The Arduino has 14 GPIO pins from which 6 support pulse width modulation (PWM). Because PWM is needed for controlling servo motors, that was the reason why we chose the Arduino as our platform along with the fact that the Arduino promised to offer an uncomplicated development.

2.2 Servo Motors

For the actuation, we used three Tower Pro MG90S servo motors. To connect them with the Arduino we used ordinary jumper cables. A 5V/2A power supply has been used in order to power the servo motors. Furthermore, a breadboard was used for distributing the power. Because the power supply originally ended in a coaxial power connector, we modified it have jumper cables as output for positive and negative voltage.

2.3 Webcam

A Microsoft LifeCam HD-3000 was used for the computer vision. Any webcam would have served the purpose.

3. ROBOTIC MANIPULATOR

3.1 Choosing the dimensions

For accurate two dimensional actuation, a classical robotic manipulator that consists a chain of dependent joints is not suitable for a low budget approach. The plotclock, a project by Johannes "*Joo*" Heberlein from the Fablab Nuremberg (comparable to a makerspace) impressively proved a way to make low budget 2D actuation work. Therefore, we planed to chose a similar manipulator. However, while the plotclock needs to cover a range whose horizontal dimensions exceed the vertical, it was not possible to blindly adapt the manipulator. For our project, actuation needs to be performed in an area formed like a square. Therefore, we needed to resize the limbs.

This problem can be described as a maximization problem: The desired area of the largest possible square that fits into the reachable range of the actuator is given. Find the dimensions L_1, L_2 and L_3 that minimize the sum $L_1+L_2+L_3$.

Unfortunately, the function of the largest square $A(L_1, L_2, L_3)$ is not defined by a function that allows for partly deriving towards L_1, L_2 and L_3 and setting the derivations to zero. We therefore chose to model the reachable space in a geometry software called *Geogebra*, an open source software for supporting mathematical education in schools. Then, we empirically derived a near optimal sizing that yields a reachable square of $100cm^2$.

3.2 Manufacturing

3.3 Simplified Kinematic Model

A simplified sketch of the robotic manipulator can be seen in Figure 3.3. As a first approximation it is useful to determine the angles θ_1 to θ_4 from the given position of the joint at which the two arms coincide (x,y). The key for solving this inverse kinematics problem is to divide it into smaller subproblems that each can be solved individually. For that purpose, the line segments a, b and c are introduced.

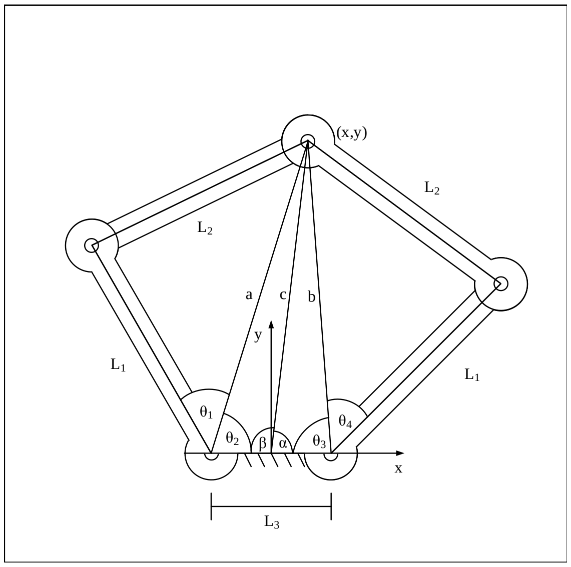

Figure 1: Simplified Version of the Manipulator

From the initial information, following values can be directly computed:

$$\alpha = arctan\left(\frac{y}{x}\right) \tag{1}$$

$$\beta = \pi - \alpha \tag{2}$$

$$c = \sqrt{x^2 + y^2} \tag{3}$$

$$a = \sqrt{\left(x + \frac{L_3}{2}\right)^2 + y^2} \tag{4}$$

$$b = \sqrt{\left(x - \frac{L_3}{2}\right)^2 + y^2} \tag{5}$$

Now, you can solve for θ_2 and θ_3 by either using sine rule or cosine rule. However, the sine rule can be ambiguous in certain setups, which makes case differentiation necessary. Although mathematically steady, in our implementation the domain crossing from one solution to the other resulted in discontinuities of the movement. Therefore, the cosine rule solution is preferred:

$$\theta_2 = arccos\left(\frac{a^2 + (\frac{L_3}{2})^2 - c^2}{aL_3}\right) \tag{6}$$

$$\theta_3 = arccos\left(\frac{b^2 + (\frac{L_3}{2})^2 - c^2}{bL_3}\right) \tag{7}$$

Using cosine rule we can also solve for θ_1 and θ_4:

$$\theta_1 = arccos\left(\frac{a^2 + {L_1}^2 - {L_2}^2}{2aL_1}\right) \tag{8}$$

$$\theta_4 = arccos\left(\frac{b^2 + {L_1}^2 - {L_2}^2}{2bL_1}\right) \tag{9}$$

3.4 Complete Kinematic Model

The simplified version of the kinematic model is good for quickly creating a working implementation. However, any movements executed by the manipulator will suffer from distortion. The pen is not mounted *exactly* at the joint's position but in a small distance. Hence, a precise solution is necessary. For that purpose, new definitions must be made (Figure 3.4).

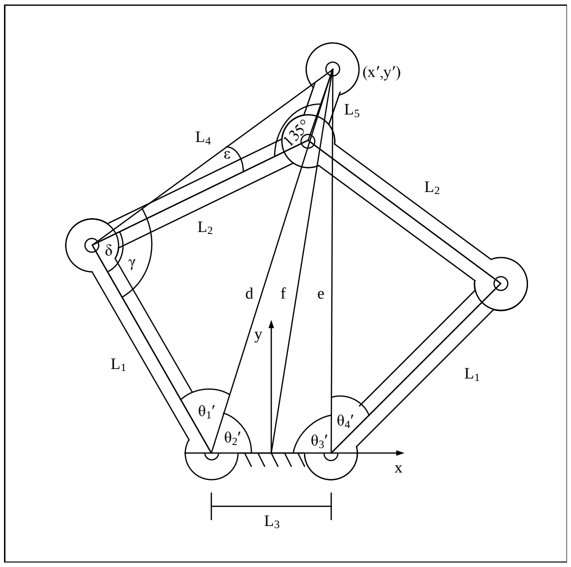

Figure 2: Complete Version of the Manipulator

L_4 and ϵ are fixed measures and not influenced by the position of the manipulator:

$$L_4 = \sqrt{{L_2}^2 + {L_5}^2 - 2L_5L_2cos\left(\frac{3\pi}{4}\right)} \tag{10}$$

$$\epsilon = arccos\left(\frac{{L_4}^2 + {L_2}^2 - {L_5}^2}{2L_4L_2}\right) \tag{11}$$

d, e and f can be yielded by the Pythagorean theorem:

$$f = \sqrt{x'^2 + y'^2} \tag{12}$$

$$d = \sqrt{\left(x' + \frac{L_3}{2}\right)^2 + y'^2} \tag{13}$$

$$e = \sqrt{\left(x' - \frac{L_3}{2}\right)^2 + y'^2} \tag{14}$$

Now, θ_2', θ_3' and δ can be computed using cosine rule:

$$\theta_2' = arccos\left(\frac{d^2 + (\frac{L_3}{2})^2 - f^2}{dL_3}\right) \tag{15}$$

$$\theta_3' = arccos\left(\frac{e^2 + (\frac{L_3}{2})^2 - f^2}{eL_3}\right) \tag{16}$$

$$\delta = arccos\left(\frac{{L_4}^2 + {L_1}^2 - d^2}{2L_4L_1}\right) \tag{17}$$

δ is the sum of ϵ and γ.

$$\gamma = \delta - \epsilon \tag{18}$$

That allows us to calculate some quantities of the simple kinematics model:

$$a = \sqrt{{L_1}^2 + {L_1}^2 - 2L_1L_2cos\left(\gamma\right)} \tag{19}$$

$$\theta_1 = arccos\left(\frac{a^2 + {L_1}^2 - {L_2}^2}{2aL_1}\right) \tag{20}$$

$$\theta_2 = \theta_1' + \theta_2' - \theta_1 \tag{21}$$

Finally, we are able to find the position of the joint at which the two arms coincide:

$$y = a\sin(\theta_2) \tag{22}$$

$$x = a\cos(\theta_2) - \frac{L_3}{2} \tag{23}$$

Now, the methods from section 3.3 can be used to solve for θ_3 and θ_4.

3.5 Modeling the Servo Motors

Servo motors are motors that do not support continuous motion but in return precisely can be driven to a desired angle. They have three external cables in different colors. The black cable should be connected to ground and the red one to V_{DD} (approximately 5V). These two cables supply the servo motor with the necessary power. The third cable (white, yellow or orange) carries the control signal. The control is done by pulse width modulation (PWM). The servo motor expects to receive a pulse every 20 ms with a pulse width between 1 ms and 2 ms. By proportional control, the servo motor assumes its most positive position at the the pulse width of 1 ms and the most negative position at a pulse width of 2 ms as defined in the mathematical direction of rotation. The range in between these extrema can be assumed to be linearly covered, i.e. a pulse width of 1.5 ms should yield the position in between these positions. A function can be derived that maps the pulse width to an angular position of the servomotor:

$$\begin{aligned}\text{Angle}(t_{\text{pulse}}) = {} & \frac{t_{\text{pulse}} - 1\,\text{ms}}{2\,\text{ms} - 1\,\text{ms}} \cdot (\text{Angle}_{max} - \text{Angle}_{min}) \\ & + \text{Angle}_{min}, \qquad 1\,\text{ms} \le t_{\text{pulse}} \le 2\,\text{ms}\end{aligned}$$

This is just a model of the movement of the servo motors. There are three causes that the actual behavior deviates from the model:

- A high torque forces the servo motors from leaving its desired position.
- The backlash of the gears in the servo motors adds an inaccuracy to the position.
- Nonlinearities make the servomotor cover the range of movement not evenly.

Furthermore, when the pulse width modulation signal is created by a digital signal, a quantization error occurs. As you can see, the model is making some approximations. The inner of the servo motor is treated as black box. However, adding the details would bloat the model and the gain is questionable. The proposed model is precise enough to be useful but not so complex that it becomes cumbersome.

4. SOFTWARE ON THE ARDUINO

4.1 Overview over the software structure

Every program written with the Arduino IDE consists of the function `void init()` and `void loop()`. The `init()` function is executed once at startup and afterwards the Arduino keeps looping through the `loop()` function. That is, because as a typical microcontroller, the Atmel ATmega328P does not have an operating system and can not just end a task. In addition, following major functions have been defined:

- `void lower()` lowers the marker.
- `void lift()` removes the marker from the surface.
- `void calculateServoMap(double x_prime,double y_prime,double* servoLsoll,double* servoRsoll)` calculates the values that should be written to the servo motors given an x-y-position.
- `void set_servos(double x, double y)` calls `calculateServoMap` and writes to the servo motors.
- `void go_to (double xto, double yto)` interpolates between the current position and the desired position of the manipulator and successively sets the position of the servo motors using `set_servos` in order to achieve a smooth movement between the origin and the destination position.
- `void draw_{cross,circle}(double x_offset, double y_offset)` draws cross/circle to any position defined by a passed x-y-position.
- `void draw_field()` draws the playing field.

The main functionality is implemented in the `loop()` function: The processor waits for an instruction coded in a byte send from the master. Once it receives the command, it is executed and following an acknowledgment is sent that it has finished the execution.

4.2 Controlling the Servo Motors

The Arduino Uno has six PWM pins available. Very conveniently, the Arduino IDE already is equipped with a library `Servo`. This library allows us to instantiate objects of the type `Servo`. The most important functions on this object are `attach()` and `write()`. The `attach()` function assigns the Servo object to a GPIO pin. The `write()` maps an angle in the range of 0° to 180° to a pulse width and makes the attached GPIO pin assume the according PWM. Inherently, a reachable range of 180° is assumed for the servo motor. However, our servo motors only are capable of spinning 150°. Signals that exceed the range of approximately 15° to 165° simply have been ignored.

Additionally,the gear that is on top of the servo motor did not allow us to attach the hat that enables us to move the limbs in any desired way. Therefore, the angle due to this offset also needed to be considered.

```
#define leftMax 161
#define rightMax 16
...
void calculateServoMap(:::){
...
*servoLsoll = leftMax - (pi -
    (theta1+theta2) ) * 180/pi;
*servoRsoll = rightMax + (pi -
    (theta3+theta4) ) * 180/pi;
}
```

Listing 1: Niko

```
void Niko void
```

5. SOFTWARE ON THE COMPUTER

5.1 Computer Vision

5.2 Tic-Tac-Toe AI

As a simple and well known game, tic-tac-toe already is fully explored and algorithms to optimally play the game are available. Also, the relatively state space of the board make exhaustive search for an optimal solution possible. The concept of our implementation in Python is nicely explained in [LINK einfÃijgen].Furthermore, [ANOTHER SOURCE] served as basis for our code. Still, severe changes had to be made, since this heavily relies on functions supplied by a GUI.

Since the detailed explanation can be read in [Link], the principle here should only summarized: A 3 × 3 2D-array of represents the nine gaps in the field. On each move, the algorithm iterates over the gaps and when it encounters an empty gap it recursively calls itself. Doing so, an altered board with the new move and another active player is passed. The break condition is either that one of the players has won or there is no free gap anymore. The original caller chooses the move that has offers the best outcome with in case of a win minimum and in case of a defeat maximum number of recursive calls.

The described algorithm plays optimally, in a meaning that it is impossible to win against it. Also, in case a win can be enforced, the computer finds this optimal path to the win. Still, it lacks an understanding for common human mistakes and assumes the opponent to play optimally himself.

In order to offer some kind of a sense of achievement to the player with a probability of 15% the algorithm is disabled and a random move is made.

5.3 The Game Program

Having described each part separately, the only thing left is combining each part to form the robot that autonomously plays Tic-Tac-Toe. A superordinate loop sets up the playing field and makes the player decide whether he wants to start or not. In order to completely decouple the game from input at the computer, we made the design choice that the first player's sign is the cross and the second player's sign is the circle. With that definition, the dialog whether the player or the computer should start the game completely can be done by drawing either a cross ("I want to start!") or a circle ("The computer should start!") to the right of the playing field.

Once the setup is done, a subordinate loop is started that breaks after nine subsequent moves or when on one of the players has won.