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

2. HARDWARE

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+L3.

Unfortunately, the function of the largest square $A(L_1, L_2, L_3)$ is not linear and therefore the problem can not be solved by partly derive towards L_1, L_2 and L_3 and set 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 **??**. 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.

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

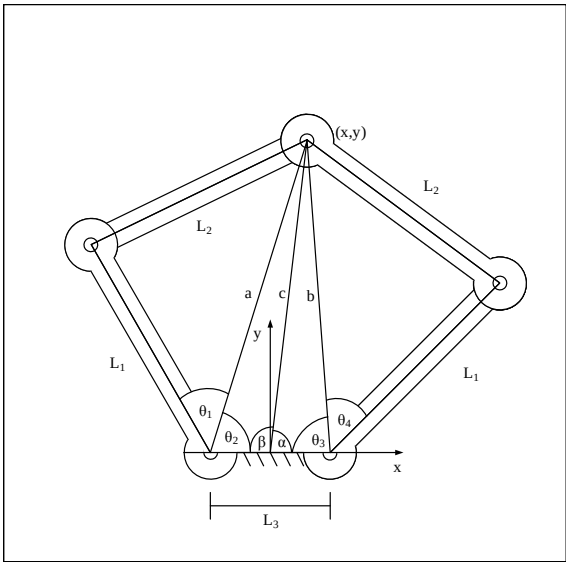

Figure 1: Simplified Version of the Manipulator

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

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

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

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

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

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 **??**).

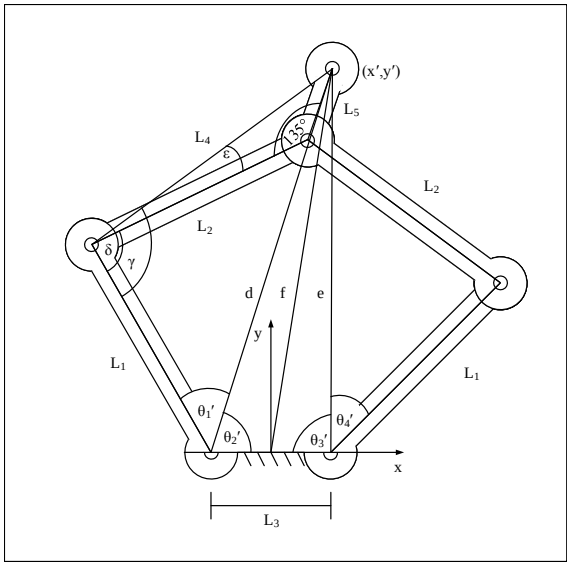

Figure 2: Complete Version of the Manipulator

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:

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

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

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

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:

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

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.

Listing 1: Angle Conversion

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

5. SOFTWARE ON THE COMPUTER

5.1 Computer Vision

5.2 Tic-Tac-Toe AI

5.3 The Game Program