Solving the Towers of Hanoi problem using a robot arm

Tom Finet, Adam Smith and Ming Sum Sze

1 Abstract

There are many examples of solutions to the Towers of Hanoi with three blocks. However, they often do not cover the physical movement of the blocks using a robot arm. This report documents the development of a robot arm which partially solves the Towers of Hanoi problem for any number of blocks within the robots reach in C. The Towers of Hanoi problem involves moving a stack of n blocks of decreasing size from a start pole to a finish pole. The problem was solved using a simple recursive routine because the focus of this report was on the robot arm. Controlling the robot arm was done using inverse kinematics which allows the robot arm to dynamically solve the Towers of Hanoi for any number of blocks. Emphasis was placed on abstracting low level functions of the robot arm such as serial communication to servo motors, from high level functions such as moving a block from one pole to another. The robot arm performed most steps required to solve the Towers of Hanoi with three blocks, but failed to move to the correct height at the same step repeatedly; likely due to memory management issues. No tests were performed with more than 3 blocks, therefore it is unclear if our design would work for any number of blocks within the robot's reach. Further debugging of our implementation and testing of our solution needs to be done in the future to complete this report.

2 Introduction

The Towers of Hanoi is a mathematical game using three poles and n blocks of decreasing size stacked on the first pole. The blocks must be moved, in the least number of moves, one at a time, from the top of the first pole to the top of the third pole. Crucially, no larger blocks can be placed on top of smaller blocks. It demonstrates a situation where recursive problem solving is much simpler and intuitive than iterative methods.

Throughout this project, a robot arm was programmed to solve this problem using physical blocks. The robot arm had five stepper motors which could be controlled to move the robot arm's gripper to a specific height. The coordination of the motors to move the robot arm to a desired height was achieved using inverse kinematics. With inverse kinematics, the desired end location of the robot arm grabber is used to compute the stepper motor angles[6]. This allows the robot arm to solve the Towers of Hanoi for as many blocks as it can reach up to, as the desired height is proportional to the desired block number the robot arm should be positioned at.

The project was approached by designing high level functions which handle high level control of the robot arm and low level functions which handle the details of interfacing with the robot arm's stepper motors. A major aim was to abstract the details of the robot arm's hardware from the algorithm which solves the Towers of Hanoi.

In addition, another aim of the solution was to develop reusable and flexible software, improving maintainability and readability. The total number of blocks used, should be allowed to change with no impact on the correctness of the solution.

3 Requirements Analysis

The requirement specific to the Towers of Hanoi setup (A) is listed below.

1. A position, within the robot's reach, should be allocated to each of the

three poles.

The requirements specific to controlling the robot arm (B) are listed below.

1. The robot arm should be able to rotate to any of the three pole positions.
2. The robot arm should be able to move up and down to any block height in its reach.
3. The robot arm gripper should close and open to pick up and drop a block.
4. The robot arm should only hold a maximum of 1 block at a time.

The requirements specific to solving the Towers of Hanoi problem using the robot arm (C) are listed below.

1. The robot arm should be able to drop a block at the top of each pole without it falling.
2. The robot arm should be able to pick up a block at the top of each pole without moving any other blocks.
3. The robot arm should only rotate to a different pole when it is a safe height above the highest block on a pole.
4. The robot arm should only drop smaller blocks on top of larger blocks.
5. The robot arm should solve the Towers of Hanoi problem in the least number of possible moves.

4 Design

4.1 System Specification

Regarding to the requirements we listed in section Requirement Analysis, here are the list of functions that the program must include:

1. A function that takes the index of a particular pole as input, and changes the coordinates of the base step motor accordingly to reach that pole.
2. A function that takes the index of a particular block height as input, and changes the coordinates of the 3 motors (all except the base and grip motors) accordingly to reach that block height.
3. A pair of functions that change the coordinates of the grip motor to perform closing and opening of the grip.
4. A function that outputs a sequence of operation(s) that needs to be performed which solves the Tower of Hanoi problem.
5. A function that sets the coordinates of the 3 motors (all except base and grip motors) such that the robot arm will be at a safe height above the highest block on a pole.

4.2 Methods for computing angles for the robot arm

4.2.1 Method 1: Using presetted values for each height

This idea involves repeatedly, and blindly trying out different step motors' coordinates, or in other words - "Brute-forcing". The advantage of implementing this method is that this approach is straight-forward. However, there are a few disadvantages that come with it.

Firstly, the final program that implements this idea will not be able to cope with environments other than the presetted ones. Presetted environment includes the size of each block, each pole's position relative to the robot arm, and the total number of blocks initially (regarding to the problem). This program is only reliable in environments where every corresponding coordinate of the step motors have been predefined to solve the particular problem.

Secondly, implementing this idea requires us to have the robot arm and the presetted environment in-hand or otherwise no trials can be performed, meanwhile this method relies heavily on "trials and errors". Therefore, there's very limited that we can do on the program while we are not in practical sessions.

Thirdly, in the aspect of software-engineering, the final program will have little code-reusing or recycling potential as most values are "hard-coded" into the program. Also this program is definitely not future-proof and is designed to only deal with very particular cases. In addition, this program will likely include a lot of constants and variables and it will be hard to maintain.

4.2.2 Method 2: Using inverse kinematics

This idea involves creating a function in the program that is able to compute a list of coordinates for the 3 motors (all except grip and base motors), so that the robot arm reaches to a particular vertical position. This function will require a number of inputs. This includes the size of each block, the index of a particular pole, the distance between the robot arm and the pole, and the amount of blocks on that pole initially (and more). With these 4 inputs the function is able to first calculate the horizontal and vertical distances between the desired location and the robot arm such that our inverse kinematics theories can be applied. (For more details on this idea and its mechanisms, read section 4.3.1)

This is the idea that ended up being was used. Its most obvious advantage is its flexibility. Theoretically, the robot arm will be capable of reaching to any vertical position (constrained by the arm's length) without the need of using predefined values. Moreover, this program will have a high degree of reusability and extensibility, this is because any changes in the environments can be conveniently reflected by adjusting the environmental values (e.g. block size).

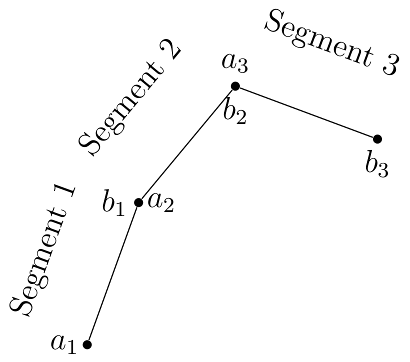
However, these advantages come at a cost - This idea is hard to implement comparably to the other idea and will require a lot of work on program designing. Also, this high level of complexity means that we are likely to encounter uncertainties and bugs. This implies that there is a higher risk of our program not being able to achieve our objectives meet the requirements under limited time we can work on this project.

4.3 Specific Program Design

This program will include several constants which together represent the environment that the robot arm is dealing with. It will include a function that takes the environment constants and an index representing the the block height as input, and converts them into 3 coordinates for the 3 motors (all except base and grip) such that robot arm reaches a particular vertical position.

4.3.1 Inverse Kinematics

Inverse kinematics is the calculation of the states of multiple segments connected together with joints with the goal of the end segment reaching a target position. This is applicable to the problem at hand as the robot arm is, in essence, 3 segments connected by joints. One can think of each segment as having a start point, a and an end point, b, making a simple line segment.



There are many algorithms for inverse kinematics. The algorithm used in this project was more of an approximation approach; with each run of the algorithm, it gets closer to the target - it never gets to the exact location. Although this may seem like a problem, the difference between getting close and being at the exact location in the real world is not going to make a difference, given it is close enough. Therefore, the algorithm needed to be run multiple times before trying to pick up a block, to ensure that the arm is close enough to it so it can pick it up. This algorithm is called the FABRIK algorithm[8]

```
Algorithm 1: FABRIK: Inverse Kinematics Algorithm
  Parameter: $target$: the position of the target
  Result: Forms a configuration of sections that gets close to a given
          target
  $currentTarget = target$
  $segment1A = segments[1].a$
  foreach Segment s (in reverse order) do
      $s.b = s.b'$ such that $s.a$, $s.b'$ and $currentTarget$ are collinear, and
        $|s.a - s.b| = |s.a - s.b'|$ and
        $|s.b - currentTarget| \geq |s.b' - currentTarget|$
      $s.a = s.a + s.b - currentTarget$
      $s.b = currentTarget$
      $currentTarget = s.a$
  end
  Translate all segments by $segments[1].a - segment1A$ (to translate
    the whole structure back to its base)
```

You may notice that the algorithm is only demonstrated in 2D: this is because this algorithm would only be applied to a particular plane at a time, and this plane would be determined by the robot's rotation about the base motor. One could then simply rotate the robot using this base motor to one of the set poles, then adjust the other motors according to this algorithm.

4.3.2 Towers of Hanoi

Solving the problem for 2 blocks allows us to find a simple algorithm for solving it[7].

1. Move the top block of the first pole to the second pole.
2. Move the largest block from the first pole to the last pole.
3. Move the block on the second pole to the last pole.

To solve this problem for 3 blocks, the same reasoning can be applied.

1. Move the top 2 blocks from the first pole to the second pole.
2. Move the largest block from the first pole to the last pole.
3. Move the 2 blocks on the second pole to the last pole.

In this way, the solution for 3 blocks uses the solution above for 2 blocks. This recursive solution can be generalised for any number of blocks n.

1. Move the top $n-1$ blocks from the first pole to the second pole.
2. Move the largest block from the first pole to the last pole.
3. Move the $n-1$ blocks on the second pole to the last pole.

The base case occurs when the number of blocks to be moved is 0, in which case nothing is done.

The following is based on a proof[5]

This algorithm solves the Towers of Hanoi in the minimum number of moves, assuming that when blocks are moved, it is done in the minimum number of moves. This is because to move the largest block from the first pole to the last pole, the $n-1$ blocks above it must all be moved to the second pole. Finally, the $n-1$ blocks must be moved from the second pole to the last pole.

In order to measure if the requirement C5 was met, a relation between the number of blocks, n, and the minimum number of moves to solve the problem must be formed.

Let M_n be the minimum number of moves. M_{n-1} moves are required in step 1 and step 3. Step 2 requires 1 move. Therefore $M_n = 2M_{n-1} + 1$ must be the minimum number of moves.

It is known that $M_0 = 0$, $M_1 = 1$, and $M_2 = 3$. In this project, testing of the algorithms correctness was done for three blocks, which requires a minimum of $M_3 = 2M_2 + 1 = 7$ moves to complete.

5 Implementation

All code was written in the C programming language. The full code repository can be found on GitHub [2]

5.1 Inverse Kinematics

5.1.1 Point Struct

To start, the Point struct was implemented; this would simply be a structure that contains an x and y location, both floating-point numbers. Some helper functions were also implemented. Some are listed here:

- `getPointMagnitude`
  Gets the Pythagorean distance of the point
- `addPoints`
  Adds two points together as vectors
- `getPointAngle`
  Gets the angle in radians the point vector makes with the positive horizontal
- `printPoint`
  Outputs a point's details to the terminal. Used for debugging

5.1.2 Servo Struct

The servo struct was a virtualized version of a segment of the robot itself. It contained two Point structs: one for the starting point of the segment, one for the ending point. There were 3 functions relating to this struct:

- `moveServoByPoint`
  Would move both servo points by a given point i.e. translating the segment by a given vector.

- `newServoLocation`
  Based on a supplied target point, it would determine a new location based on the IK algorithm described in the design section.

- `printServo`
  Outputted details of the servo struct to the terminal. Used for debugging

`moveServoByPoint` and `newServoLocation` functions:

```
// Move the whole servo by a vector
void moveServoByPoint(struct Servo *servo, struct Point *p)
{
    /* Get new positions, free the old positions from memory,
                            then set the new ones.*/
    struct Point *newA = addPoints(servo->a, p);
    free(servo->a);
    servo->a = newA;
    struct Point *newB = addPoints(servo->b, p);
    free(servo->b);
    servo->b = newB;
}
// Calculate a new servo location given a target based on algorithm.
void newServoLocation(struct Servo *servo, struct Point *target)
{
    struct Point *aToTarget = subtractPoints(target, servo->a);
    makeUnitVector(aToTarget);
    multiplyPoint(aToTarget, ARM_SERVO_LENGTH);

    struct Point *newB = addPoints(servo->a, aToTarget);
    // Point ab in the direction of the target
    free(aToTarget);
    struct Point *newBToTarget = subtractPoints(target, newB);
    // Find how far b is from the target
    struct Point *newA = addPoints(servo->a, newBToTarget);
    /* Move a by how far b needs to go to get to the target
                    i.e. move the servo to the target */
    free(newBToTarget);
    newB = copyPoint(target);
    /* Need a new point otherwise if target point supplied is changed,
                                this will also change */

    free(servo->a);
    free(servo->b);
    servo->a = newA; // Set the new values
    servo->b = newB;
}
```

5.1.3 Integrating the structs into the main code

Firstly, 3 constant values were determined for the base motor for each of the 3 different poles. Each value, if passed to the base motor, would make the whole robot 'point' in a particular pole. For each pole, the amount of blocks on it is tracked to ensure the robot arm moves to the correct height; at the start, there are n blocks on pole 1, and 0 on poles 2 and 3.

There is also an array of Servo structs which is of length 3, where each corresponds to one motor. The location of the base of the arm is also stored in a Point variable, which is used to pull the Servo structs back to the original location after performing inverse kinematics.

There is also a connection variable which allows us to communicate with the robot arm. This is only used when initialising the connection itself in the main function, then again in the `move_to_location` function which was used from the CS132 Robot Arm lab template[3]

```
struct Pole
{
    int pos;
    int index;
};

// Set pole motor positions
const int pos1 = 0x01aa;
const int pos2 = 0x01ff;
const int pos3 = 0x0254;

struct Pole pole1, pole2, pole3;

int connection;
struct Servo *servos[3]; // There are 3 segments of the robot arm, so 3 servos
struct Point *baseOfArm; // The location of the base motor of the arm
int *poleHeights;        // An array of the number of cubes on each pole
```

In the `main` function, firstly the connection is opened to the robot, then the `init` function is called. The `init` function is only ever called once, at the start of the program, and allocates various variables to memory, including the Servos and the base of arm Point.

After the `init` function, the `reset` method is called, which makes the virtualized version go to a vertical position. After this, the `drop` method is called which ensures that the robot is in a formation where it can pick up blocks. Finally, before the solving starts, the `toSafeHeight` method is called. This uses a function called `setVertical`, and makes the robot arm move to a height which is clear of all blocks (it goes to a position which is two blocks higher than the highest position possible for the particular puzzle; in the case of $n = 3$, the robot goes to a height of block 5).

The `setVertical` function makes use of two new functions: `setServoHeightsArr` and `setServoAngle`. The `setServoHeightsArr` actually performs the inverse kinematics needed for the robot arm, iterating through each servo struct in reverse order, then calling the `newServoLocation` function on each one, with the target being the previous servos' a position (for the first servo, the target is the actual desired end target). It is important to note that the `setServoHeightsArr` function only affects the virtualized version of the robot arm; the actual application of the configuration shown in the virtualized version to the real robot arm is done by the `setVertical` function. The `setServoAngle` simply sets a particular robot motor's angle to the angle supplied; it makes use of the `radsToRobotArmServoValue` function, which converts a given radian value to a value that the robot arm can understand, and the `mtl` function, which is a modified version of the `move_to_location` function[3] which automatically bit shifts the given value to get the high and low bits, which need to be separated for the logic in `move_to_location`.

```c
// Given a target height, the motor's are adjusted using the IK algorithm
void setServoHeightsArr(int blockNumber)
{
    int i;
    struct Point *target = malloc(sizeof(struct Point));
    // For the first servo, the target is the actual target wanted
    target->x = ARM_BASE_TO_CUBES_DIST;
    // The x distance is constant: the three poles are circular around the robot
    target->y = BLOCK_HEIGHT * (blockNumber - 1) + BLOCK_Y_OFFSET;
    // Based on the constants set and height given.
    // Block number 1 means the first block, not second like usual
    for (i = 2; i >= 0; i--)
    /* Go through each servo,
    starting at the servo most distant from the base motor. */
    {
        struct Servo *s = servos[i];
        newServoLocation(s, target);
        // Get the new servo location, based on the target set
        target->x = s->a->x;
        target->y = s->a->y;
        /* For all servos except the first,
        the target is the a point of the previous servo*/
    }
    free(target);
    // No longer needed
    struct Point *offset = subtractPoints(baseOfArm, servos[0]->a);
    // Get how far the base a point has moved from the origin point.
    for (i = 0; i < 3; i++)
    {
        moveServoByPoint(servos[i], offset);
        // Move all servos back by this amount
    }
    free(offset);
}
```

The `setVertical` function first calls `setServoHeightsArr` 3 times (multiple times to ensure accuracy, see design for more details). It then iterates through each servo and appropriately sets each motor's angle, based on the point structs for each corresponding servo; this makes use of the `getPointAngle` and `setServoAngle` functions. Of course, if it was to set the angle made by simply the point struct in the servo, it would be offset, as the angle, for example, the second motor is at depends on the first motor. Hence, the previous motor's angle needed to be kept track of, so that the difference between that and the `getPointAngle` return value could be used instead.

```c
void setVertical(int block)
{
    reset();
    printf("Setting vertical to: %d\n", block);
    int i;
    for (i = 0; i < 3; i++)
    { // Doing it 3 times ensures it's in the right place
        setServoHeightsArr(block);
    }
    int prevAngle = 0;
    for (i = 0; i < 3; i++)
    {
        struct Point *diff = subtractPoints(servos[i]->b, servos[i]->a);
        float angle = -abso(getPointAngle(diff));
        printf("Angle(%d): %.6f\n", i + 2,
            radsToDegrees(getPointAngle(diff) - prevAngle));

        setServoAngle(6 - (i + 2), angle - prevAngle);
        prevAngle = angle;
    }
}
```

In the project, the `wait` function was also used, which would pause code execution for a short amount of time, to allow the robot to complete the action it was sent; this function was taken from the CS132 Robot Arm lab template[3].

Finally, two simple functions were implemented to allow us to make the robot arm grip and release the blocks with a simple function. The `grab` and `drop` functions are shown below. They use values which were found simply by trial and error.

```c
// Sets the gripping mechanism to a preset which grips a block securely
void grab()
{
    mtl(5, 0x0144);
}
/* Sets the gripping mechanism to a preset
    where blocks can be freed from the grip */
void drop()
{
    mtl(5, 0x01ff);
}
```

5.2 Towers of Hanoi

The function which recursively solves the Towers of Hanoi problem is called `solveHanoi`. It takes in three arguments: the total number of blocks, first pole, second pole, and last pole.

```
void solveHanoi(int n, struct Pole from, struct Pole aux, struct Pole to)
{

    if (n > 0)
    {
        solveHanoi(n - 1, from, to, aux); // move n - 1 disks to the middle.
        moveBlock(from, to);              // move largest disk to last pole.
        solveHanoi(n - 1, aux, from, to); // move n - 1 disks to last pole.
    }
}
```

This function was simple to implement as it does not handle any of the robot arm specifics. These are abstracted away by a high level function called `moveBlock` which takes two arguments: the pole to move the block from and the pole to move the block too. The `moveBlock` function makes use of the `toSafeHeight`, `setVertical`, `setPole`, `grab`, `drop`, and `wait` functions. These were put in a simple sequence to create the motion needed.

6 Testing

In this section we will be covering how our final program performed practically.

6.1 Units Testing

- `wait`
  When executed between another two movement commands which utilised the `move_to_location` function, robot arm waited for a predefined amount of time between 2 actions. The time was set to either 2 or 4 seconds, which were sufficient enough to prevent individual movements from clashing. In the `moveBlock` function, `wait` function was inserted between any 2 other functions.

- `toSafeHeight`

When this unit is implemented alone, robot arm moved to a particular height we presetted. This behaviour matched our expectations. The extra height of 2 blocks is very sufficient to prevent robot arm from touching other poles.

- `grab` and `drop`
  These 2 functions both utilised the `move_to_location` function. When performed one after the other, robot arm expanded and contracted its grip such that a block could be gripped and released. The angle of which the grip contracts was set sensibly such that a right amount of force was applied onto the block.

- `setPole`
  This function utilises the `move_to_location` function from before. The pole's positions are passed in as parameter such that robot arm would turn to face the pole. This unit was working properly.

- `setVertical`
  This function is integrated with `setServoHeightsArr`. Testing shown the robot arm could accurately reach each vertical position.

- Helper Functions
  These functions are there to perform simple mathematics on numbers usually to convert a value's measuring unit. They worked flawlessly and were easily integrated with different units.

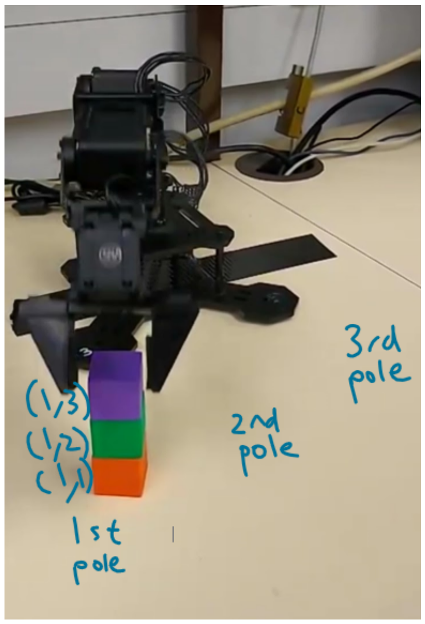
6.2 System Testing

We performed more than 3 trial runs on our finalised version of our program. It is observed that the robot arm was performing the exact same sequence of actions (as shown in the video clip below), which implies that the video clip below is an accurate presentation of our program.
The environment for testing was set as the following:

- Total of 3 blocks
- 20cm between each pole and the robot arm (center of the base)

- Block dimension is $2.5^3 cm^3$
- The relative coordinates of the poles to the robot arm are around -60,0,60 degrees

List of moves performed during test:
Goes to (1,x), where x is about 10, and grip closed (no blocks gripped)
Goes to (3,1) and grip opened (no blocks released)
Goes to (1,2) and grip closed (purple and green gripped)
Goes to (2,1) and grip opened (purple and green released)
Goes to (3,1) and grip closed (no blocks gripped)
Goes to (2,2) and grip opened (no blocks released)
Goes to (1,1) and grip closed (orange gripped)
Goes to (3,1) and grip opened (orange released)
Goes to (2,2) and grip closed (purple gripped)
Goes to (1,1) and grip opened (purple released)
Goes to (2,1) and grip closed (green gripped)
Goes to (3,2) and grip opened (green released)
Goes to (1,1) and grip closed (purple gripped)
Goes to (3,y), where y is about 10, and grip opened (purple released)



We later investigated for some possible reasons for specific behaviours in section Evaluation.

7 Evaluation

7.1 Preliminary Analysis

Steps 1, 3, 5, 14 seems odd. If in the first step the robot arm instead goes to (1,3) and closed grip to grab the purple block, step 3's odd behaviour can be explained as the purple block was not supposed to be in (1,3) during that step. (As a result two blocks were grabbed at a time.) This also applies to step 5 as the purple block would have been in position (3,1). Therefore we believe that step 1's abnormal behaviour had caused a chain of abnormal behaviours demonstrated in later steps. Behaviours seem normal afterwards until at step 14 where robot arm supposedly should go to (3,3) and open its grip.

7.2 Regarding to the requirements that we set in section Requirement Analysis

Regarding to the requirements specific to controlling the robot arm (B)

1. Our robot arm was able to rotate to any of the three pole positions after we adjusted the corresponding environment variables to suit the poles' relative coordinates to the robot arm.
2. The robot arm was able to move up and down to any block height between 1 to 3 in its reach.
3. The robot arm gripper was able to close and open to pick up and drop a block.
4. The robot arm failed to grip onto a maximum a 1 block at a time.

Regarding to the requirements specific to solving the Towers of Hanoi problem using the robot arm (C)

1. The robot arm was able to drop a block at the top of each pole without it falling throughout our tests done with 3 total blocks. This might not hold as the total number of blocks increases. However, since our main objective is to make the program works for 3 blocks therefore we would say our program met this requirement.

2. The robot arm was able to pick up a block at the top of each pole without moving any other blocks throughout our tests done with 3 total blocks. This might not hold as the total number of blocks increases.

3. The robot arm was able to rotate to a different pole when it is a safe height above the highest block on a pole throughout our tests done with 3 total blocks. However, we believe this will continue to hold for higher number of blocks because of the safe height constant we implemented.

4. The robot arm only dropped smaller blocks on top of larger blocks. We also believe this will continue to hold for higher number of blocks.

5. The robot arm failed to solve the Tower of Hanoi problem in the first place. However throughout our testing none of the moves were redundant (except for some moves stated previously) therefore whether or not our program met the requirement is ambiguous.

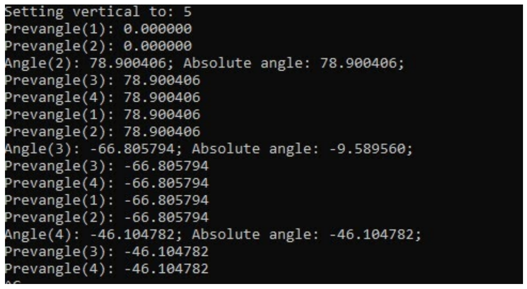
Potential cause of errors

The line inside the `setVertical` unit

```
printf("Angle(%d): %.6f\n", i + 2, radsToDegrees(getPointAngle(diff) - prevAngle));
```

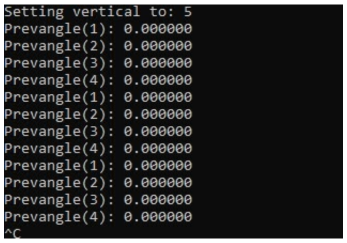
is very likely to be the cause of errors. We tested the program using the terminal instead of the robot arm, both with and without this line program of code and the results are as follows:

with the line of code:



```
Setting vertical to: 5
Prevangle(1): 0.000000
Prevangle(2): 0.000000
Angle(2): 78.900406; Absolute angle: 78.900406;
Prevangle(3): 78.900406
Prevangle(4): 78.900406
Prevangle(1): 78.900406
Prevangle(2): 78.900406
Angle(3): -66.805794; Absolute angle: -9.589560;
Prevangle(3): -66.805794
Prevangle(4): -66.805794
Prevangle(1): -66.805794
Prevangle(2): -66.805794
Angle(4): -46.104782; Absolute angle: -46.104782;
Prevangle(3): -46.104782
Prevangle(4): -46.104782
```

without the line of code:



```
Setting vertical to: 5
Prevangle(1): 0.000000
Prevangle(2): 0.000000
Prevangle(3): 0.000000
Prevangle(4): 0.000000
Prevangle(1): 0.000000
Prevangle(2): 0.000000
Prevangle(3): 0.000000
Prevangle(4): 0.000000
Prevangle(1): 0.000000
Prevangle(2): 0.000000
Prevangle(3): 0.000000
Prevangle(4): 0.000000
^C
```

It was shown that the variable `prevAngle` was dependent on this `printf` function, which was not something we expected. However we could not explain the reason why this error would only affect a certain moves.

8 Conclusion

Reflecting from the Evaluation section, although we successfully implemented our idea into the program, our program was not able to solve ultimately, the Tower of Hanoi problem. Moreover, we had not performed any tests with more than 3 blocks and therefore it is uncertain whether or not our program would continue to meet some of these requirements as the number of blocks increases. Future work would be to continue modifying our program such that it can consistently solve the problem in various different scenarios. Apart from fixing the error currently in the program, this also includes developing a way to take more accurate measurements (e.g on the size of block), variable wait time on the `wait` function depending on the distance that the robot has to travel, and a variable safe height on `toSafeHeight` function that depends on the current block height of each pole.

Overall, we did fulfil our aim of trying to maximise the flexibility and reusability of the program, and our idea of inverse kinematics was also well abstracted and implemented. Our main outcome of this project is experiencing the workflow of solving an engineering problem, including dividing a complex problem into smaller sub-problems such that problem was more easier to approach.

The final result can be found on YouTube[4]

References

[1] CS132 coursework 2 specification. https://warwick.ac.uk/fac/sci/dcs/teaching/material/cs132/cw2.pdf. Accessed: 2020-01-22.

[2] CS132 project. https://github.com/ajsmith595/CS132-Project. Accessed: 2020-01-22.

[3] CS132 robot arm lab webpage. https://warwick.ac.uk/fac/sci/dcs/teaching/material/cs132/labs/robotarm/. Accessed: 2020-01-22.

[4] CS132 robot arm video. https://youtu.be/OnYR2lT3ANQ. Accessed: 2020-01-22.

[5] Towers of hanoi proof. https://proofwiki.org/wiki/Tower_of_Hanoi. Accessed: 2020-01-22.

[6] What is Inverse Kinematics? https://uk.mathworks.com/discovery/inverse-kinematics.html. Accessed: 2020-01-22.

[7] Uroš Milutinović Ciril Petr Andreas M. Hinz, Sandi Klavžar. *The Tower of Hanoi – Myths and Maths*. 2013.

[8] Andreas Aristidou and Joan Lasenby. FABRIK: A fast, iterative solver for the inverse kinematics problem. *Graphical Models*, 73:243–260, 09 2011.