

ROBOT 1270: C PROGRAMMING

Lab #6 SCARA Robot Advanced Control

Table of Contents

[1 Task 1 3](#_Toc135081037)

[2 Task 3 7](#_Toc135081038)

[3 Appendix 8](#_Toc135081039)

Table of Figures

[Figure 1: Bad data log Part1 3](#_Toc135081040)

[Figure 2: Bad data log Part2 4](#_Toc135081041)

[Figure 3: Bad data log Part3 5](#_Toc135081042)

[Figure 4: Bad data log Part4 6](#_Toc135081043)

[Figure 5: CREATIVE task 7](#_Toc135081044)

# Task 1

A screenshot of a computer

Description automatically generated with medium confidence

Figure 1: Bad data log Part1

A screenshot of a computer

Description automatically generated with medium confidence

Figure 2: Bad data log Part2

A screenshot of a computer

Description automatically generated with medium confidence

Figure 3: Bad data log Part3

A screenshot of a computer

Description automatically generated with medium confidence

Figure 4: Bad data log Part4

# Task 3

A screenshot of a computer

Description automatically generated with medium confidence

Figure 5: CREATIVE task

# Appendix

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Course: ROBT1270 - C Programming

Program: Lab6: SCARA Robot Simulator Advanced Control

Purpose: To demonstrate advanced control over the SCARA Robot Simulator using inverse kinematic

functions. Remote commands are sent to the simulator using formatted command strings read from a file.

Programming methods: formatted I/O, conditional statements, pointers, functions, strings,

arrays, structures, file I/O, dynamic memory allocation, coordinate transformations.

Authors: Zachariah Loewen

Declaration: I, Zachariah, declare that the following program was written by me/us.

Date Created: April 19 2023

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//-------------------------- Standard library prototypes --------------------------------------------------------------

#include <stdlib.h> // standard functions and constant

#include <stdio.h> // i/o functions

#include <math.h> // math functions

#include <string.h> // string functions

#include <ctype.h> // character functions

#include <stdbool.h> // bool definitions

#include "robot.h" // robot functions

//---------------------------- Program Constants ----------------------------------------------------------------------

const double PI = 3.14159265358979323846; // the one and only

const double L1 = 350.0; // length of the inner arm

const double L2 = 250.0; // length of the outer arm

const double ABS\_THETA1\_DEG\_MAX = 150.0; // maximum magnitude of shoulder angle in degrees

const double ABS\_THETA2\_DEG\_MAX = 170.0; // maximum magnitude of elbow angle in degrees

const double LMAX = L1 + L2; // max L -> maximum reach of robot

const double LMIN = sqrt(L1 \* L1 + L2 \* L2 - 2.0 \* L1 \* L2 \* cos(PI - ABS\_THETA2\_DEG\_MAX \* PI / 180.0)); // min L

const unsigned char HL = 196; // for console (code page 437)

const unsigned char FHL = 151; // for file (code page 1252)

const unsigned char PLUSMINUS\_SYMBOL = 241; // the plus/minus ascii symbol

const unsigned char DEGREE\_SYMBOL = 248; // the degree symbol

const double ERROR\_VALUE = DBL\_MAX; // value for angles when robot can't reach

const char\* seps = "\t,\n ;:"; // for tokenizing the line string

// number of points on path for every 500 units of arc length

const int LOW\_RESOLUTION\_POINTS\_PER\_500\_UNITS = 11;

const int MEDIUM\_RESOLUTION\_POINTS\_PER\_500\_UNITS = 31;

const int HIGH\_RESOLUTION\_POINTS\_PER\_500\_UNITS = 51;

const int PRECISION = 2; // for printing values to console

const int FIELD\_WIDTH = 8; // for printing values to console

const int COMMAND\_INDEX\_NOT\_FOUND = -1; // used when command index not found

const int BLANK\_LINE = -2; // used to signal a blank line in the input file

#define COMMAND\_STRING\_ARRAY\_SIZE 502 // size of array to store commands written by sprintf\_s for robot.

// NOTE: 2 elements must be reserved for trailing '\n' and '\0'

#define MAX\_LINE\_SIZE 1002 // size of array to store a line from a file.

// NOTE: 2 elements must be reserved for trailing '\n' and '\0'

enum ARM { LEFT, RIGHT }; // left arm or right arm configuration

enum MOTOR\_SPEED { MOTOR\_SPEED\_LOW, MOTOR\_SPEED\_MEDIUM, MOTOR\_SPEED\_HIGH }; // motor speed

enum RESOLUTION { RESOLUTION\_LOW, RESOLUTION\_MEDIUM, RESOLUTION\_HIGH }; // motor speed

enum CURRENT\_ANGLES { GET\_CURRENT\_ANGLES, UPDATE\_CURRENT\_ANGLES }; // used to get/update current SCARA angles

enum COMMAND\_INDEX // list of all command indexes

{

ROTATE\_JOINT, MOTOR\_SPEED, PEN\_UP, PEN\_DOWN, CYCLE\_PEN\_COLORS, PEN\_COLOR, CLEAR\_TRACE,

CLEAR\_REMOTE\_COMMAND\_LOG, CLEAR\_POSITION\_LOG, SHUTDOWN\_SIMULATION, END, HOME, LINE, ARC, MOVE\_TO,

TRIANGLE, RECTANGLE, QUADRATIC\_BEZIER, ROTATE, TRANSLATE, SCALE, RESET\_TRANSFORM\_MATRIX, NUM\_COMMANDS

};

//---------------------------- Structure Definitions ------------------------------------------------------------------

// structure to map command keyword string to a command index

typedef struct COMMAND

{

const int index;

const char\* strCommand;

}

COMMAND;

// RGB color

typedef struct RGB

{

int r, g, b; // ranges are 0-255

}

RGB;

// SCARA tooltip coordinates

typedef struct TOOL\_POSITION

{

double x, y;

}

TOOL\_POSITION;

// SCARA joint angles (degrees)

typedef struct JOINT\_ANGLES

{

double theta1Deg, theta2Deg;

}

JOINT\_ANGLES;

// pen state

typedef struct PEN\_STATE

{

RGB penColor;

int penPos;

}

PEN\_STATE;

// forward kinematics solution data

typedef struct FORWARD\_SOLUTION

{

TOOL\_POSITION toolPos; // tool tip coordinates

bool bCanReach; // true if robot can reach, false if not

}

FORWARD\_SOLUTION;

// inverse kinematics solution data

typedef struct INVERSE\_SOLUTION

{

JOINT\_ANGLES jointAngles[2]; // joint angles (in degrees). Left and Right arm solutions

bool bCanReach[2]; // true if robot can reach, false if not. Left and right arm configurations

}

INVERSE\_SOLUTION;

typedef struct PATH\_CHECK

{

bool bCanDraw[2]; // true if robot can draw, false if not. Left and right arm configurations

double dThetaDeg[2]; // total angle changes required to draw path

}

PATH\_CHECK;

// Holds the start, end coordinates and RESOLUTION the of a line

typedef struct LINE\_NODES

{

TOOL\_POSITION start;

TOOL\_POSITION end;

size\_t RESOLUTION;

}

LINE\_NODE;

// Holds the start, end coordinates and RESOLUTION the of a line

typedef struct Arc\_NODES

{

TOOL\_POSITION cecter;

double arcStart;

double arcEnd;

double radious;

size\_t RESOLUTION;

}

Arc\_NODES;

// Holds the NODE data and the address of the points

typedef struct BEZIER\_NODES

{

TOOL\_POSITION points[3];

double RESOLUTION;

}

BEZIER\_NODES;

// Holds the NODE data and the address of the points

typedef struct LINE\_ID

{

LINE\_NODES NODES;

size\_t numPoints;

JOINT\_ANGLES\* joints[2];

TOOL\_POSITION\* Points;

}

LINE\_ID;

// Holds the NODE data and the address of the points

typedef struct Arc\_ID

{

Arc\_NODES NODES;

size\_t numPoints;

JOINT\_ANGLES\* joints[2];

TOOL\_POSITION\* Points;

}

Arc\_ID;

// Holds the NODE data and the address of the points

typedef struct BEZIER\_ID

{

BEZIER\_NODES NODES;

size\_t numPoints;

JOINT\_ANGLES\* joints[2];

TOOL\_POSITION\* Points;

}

BEZIER;

//----------------------------- Globals -------------------------------------------------------------------------------

// global array of command keyword string to command index associations

// NOTE: CYCLE\_PEN\_COLORS must preceed PEN\_COLOR

const COMMAND m\_Commands[NUM\_COMMANDS] = { {ROTATE\_JOINT, "ROTATE\_JOINT"}, {MOTOR\_SPEED, "MOTOR\_SPEED"},

{PEN\_UP, "PEN\_UP"}, {PEN\_DOWN, "PEN\_DOWN"},

{CYCLE\_PEN\_COLORS, "CYCLE\_PEN\_COLORS"}, {PEN\_COLOR, "PEN\_COLOR"},

{CLEAR\_TRACE, "CLEAR\_TRACE"},

{CLEAR\_REMOTE\_COMMAND\_LOG, "CLEAR\_REMOTE\_COMMAND\_LOG"},

{CLEAR\_POSITION\_LOG, "CLEAR\_POSITION\_LOG"},

{SHUTDOWN\_SIMULATION, "SHUTDOWN\_SIMULATION"}, {END, "END"}, {HOME, "HOME"},

{LINE, "LINE"}, {ARC, "ARC"}, {MOVE\_TO, "MOVE\_TO"},

{TRIANGLE, "TRIANGLE"},{RECTANGLE, "RECTANGLE"},

{QUADRATIC\_BEZIER, "QUADRATIC\_BEZIER"},{ROTATE, "ROTATE"},

{TRANSLATE, "TRANSLATE"},{SCALE, "SCALE"},

{RESET\_TRANSFORM\_MATRIX, "RESET\_TRANSFORM\_MATRIX"} };

CRobot robot; // the global robot Class. Can be used everywhere

FILE\* flog = NULL; // the global log file

//----------------------------- Function Prototypes -------------------------------------------------------------------

bool flushInputBuffer(); // flushes any characters left in the standard input buffer

void waitForEnterKey(); // waits for the Enter key to be pressed

int nint(double); // computes nearest integer to a double value

double degToRad(double); // returns angle in radians from input angle in degrees

double radToDeg(double); // returns angle in degrees from input angle in radians

double mapAngle(double); // make sure inverseKinematic angled are mapped in range robot understands

void pauseRobotThenClear(); // pauses the robot for screen capture, then clears everything

void printHLine(int N); // prints a solid line to the console

int dsprintf(char const\*, ...); // prints to log file and to console

void makeStringUpperCase(char\*); // makes an input string all upper case

size\_t getNumPathPoints(double, int); // gets the number of points on a path based on arc length and resolution value

void robotAngles(JOINT\_ANGLES\*, int); // gets or updates the current SCARA angles

INVERSE\_SOLUTION inverseKinematics(TOOL\_POSITION); // get left/right arm joint angles from x,y pos

PATH\_CHECK checkLinePath(LINE\_ID\*); // will check witch arm cann draw the points on the line

PATH\_CHECK checkArcPath(Arc\_ID\*); // will check witch arm cann draw the points on the arc

PATH\_CHECK checkBezierPath(BEZIER\_ID\*); // will check witch arm cann draw the points on the bezier

TOOL\_POSITION\* getLineData(LINE\_ID\*, double TM[][3]); // will return the address of the points for the line

TOOL\_POSITION\* getArcData(Arc\_ID\*, double TM[][3]); // will return the address of the points for the line

TOOL\_POSITION\* getBezierData(BEZIER\_ID\*, double TM[][3]); // will return the address of the points for the besier

bool setPenColor(char\* strCommandLine); // set the color of the pen

bool setJointAngles(char\* strCommandLine); // set the angles of the joints

bool setPenPos(char\* strCommandLine); // will move the pen to the input posishion

bool setmotorSpeed(char\* strCommandLine); // will set the motor speed

bool drawLine(char\* strCommandLine, double TM[][3]); // wiil deaw a line at the star and end pints with a set resalohion

bool drawArc(char\* strCommandLine, double TM[][3]); // will draw a line with a fiven radious , stat and end angles

bool drawTRIANGLE(char\* strCommandLine, double TM[][3]); // will draw three lines to make a trinagle

bool drawRECTANGLE(char\* strCommandLine, double TM[][3]); // will draw for lines to make a rectanfle

bool drawBEZIER(char\* strCommandLine, double TM[][3]); // will draw Bezier curve

bool rotateTm(char\* strCommandLine, double TM[][3]); // will rotate the main matrix

bool translateTm(char\* strCommandLine, double TM[][3]);// will tranlate the main matrix

bool scaleTm(char\* strCommandLine, double TM[][3]); // will scale the main matrix

void processFileCommands(); // gets commands out of a file and processes them for robot control

bool setCyclePenColors(char\* strLine); // Parses line string to send a CYCLE\_PEN\_COLORS command to robot

void processCommand(int commandIndex, char\* strLine, double TM[3][3]); // processes a command string from the file

int getCommandIndex(const char\* strLine); // gets the command keyword index from a string

double getQuadraticBezierArcLength(TOOL\_POSITION P0, TOOL\_POSITION P1, TOOL\_POSITION P2); // calc Bezier curve length

void resetTransformMatrix(double TM[][3]); // resets the transform matrix to the identity matrix

void transformMatrixMultiply(double TM[][3], const double M[][3]);// premultiplies the transform matrix TM by matrix M

TOOL\_POSITION transform(const double TM[][3], TOOL\_POSITION tp); // tranform tool position coordinates

//---------------------------------------------------------------------------------------------------------------------

// DESCRIPTION: Program to demonstrate basic control of the SCARA robot simulator

// ARGUMENTS: none

// RETURN VALUE: an int that tells the O/S how the program ended. 0 = EXIT\_SUCCESS = normal termination

int main()

{

// open connection with robot

if (!robot.Initialize()) return 0;

processFileCommands();

dsprintf("\n\nPress ENTER to end the program...\n");

waitForEnterKey();

return EXIT\_SUCCESS;

}

//---------------------------------------------------------------------------------------------------------------------

// DESCRIPTION: processes robot commands stored in a file and uses them to control the SCARA robot

// ARGUMENTS: none

// RETURN VALUE: none

void processFileCommands()

{

char strFileName[MAX\_PATH]; // stores input file name

char strLine[MAX\_LINE\_SIZE]; // stores one line out of input file

FILE\* fi = NULL; // input file handle

errno\_t err; // stores fopen\_s error value

int numChars; // used to draw dividing line

int nLine; // file line number

int commandIndex = COMMAND\_INDEX\_NOT\_FOUND; // command index

char\* token = NULL, \* next\_token = NULL;

double TM[3][3] = { {1.0, 0.0, 0.0},{0.0, 1.0, 0.0},{0.0, 0.0, 1.0} };

// open the log file (mirrors console output to log.txt if dsprintf used instead of printf)

err = fopen\_s(&flog, "log.txt", "w");

if (err != 0 || flog == NULL)

{

dsprintf("Cannot open log.txt for writing! Press ENTER to end program...");

waitForEnterKey();

exit(0);

}

// get the input file

while (true)

{

dsprintf("Please enter the name of the commands file: ");

fgets(strFileName, MAX\_PATH, stdin);

strFileName[strlen(strFileName) - 1] = '\0'; // remove newline character

err = fopen\_s(&fi, strFileName, "r");

if (err == 0 && fi != NULL) break;

dsprintf("Failed to open %s!\nError code = %d", strFileName, err);

if (err == ENOENT)

dsprintf(" (File not found! Check name/path)\n");

else if (err == EACCES)

dsprintf(" (Permission Denied! Is the file opened in another program?)\n");

else

dsprintf("\n");

}

numChars = dsprintf("Processing %s\n", strFileName);

printHLine(numChars - 1);

// get each line from the input file and process the command

nLine = 0;

while (fgets(strLine, MAX\_LINE\_SIZE, fi) != NULL)

{

if (strstr(strLine, "\n") == NULL) strcat\_s(strLine, MAX\_LINE\_SIZE, "\n"); // needed for last line

nLine++;

dsprintf("Line %02d: %s", nLine, strLine); // echo the line

//--- get the command index and process it

makeStringUpperCase(strLine); // make line string all upper case (makes commands case-insensitive)

//\*\*\*\* YOUR CODE FOR getCommandIndex and processCommand GOES HERE \*\*\*\*

commandIndex = getCommandIndex(strLine);

token = strtok\_s(strLine, seps, &next\_token);

processCommand(commandIndex, next\_token, TM);

}

fclose(fi);

fclose(flog);

}

//---------------------------------------------------------------------------------------------------------------------

// DESCRIPTION: processes a command referenced by the commandIndex. Parses the command string from the file and

// packages up the command to be sent to the robot if no errors found.

// ARGUMENTS: commandIndex: index of the command keyword string

// strCommandLine: command line from the file in the form of a string

// TM the one and only transformation matrix

// RETURN VALUE: none

void processCommand(int commandIndex, char\* strCommandLine, double TM[3][3])

{

bool bSuccess = true;

JOINT\_ANGLES homeAngles = { 0.0, 0.0 };

switch (commandIndex)

{

case PEN\_UP:

robot.Send("PEN\_UP\n");

break;

case PEN\_DOWN:

robot.Send("PEN\_DOWN\n");

break;

case CLEAR\_TRACE:

robot.Send("CLEAR\_TRACE\n");

break;

case CLEAR\_REMOTE\_COMMAND\_LOG:

robot.Send("CLEAR\_REMOTE\_COMMAND\_LOG\n");

break;

case CLEAR\_POSITION\_LOG:

robot.Send("CLEAR\_POSITION\_LOG\n");

break;

case SHUTDOWN\_SIMULATION:

robot.Send("SHUTDOWN\_SIMULATION\n");

break;

case END:

robot.Send("END\n");

break;

case HOME:

robot.Send("HOME\n");

robotAngles(&homeAngles, UPDATE\_CURRENT\_ANGLES);

break;

case PEN\_COLOR:

bSuccess = setPenColor(strCommandLine);

break;

case CYCLE\_PEN\_COLORS:

bSuccess = setCyclePenColors(strCommandLine);

break;

case ROTATE\_JOINT:

bSuccess = setJointAngles(strCommandLine);

break;

case MOVE\_TO:

bSuccess = setPenPos(strCommandLine);

break;

case MOTOR\_SPEED:

bSuccess = setmotorSpeed(strCommandLine);

break;

case LINE:

bSuccess = drawLine(strCommandLine, TM);

break;

case ARC:

bSuccess = drawArc(strCommandLine, TM);

break;

case TRIANGLE:

bSuccess = drawTRIANGLE(strCommandLine, TM);

break;

case RECTANGLE:

bSuccess = drawRECTANGLE(strCommandLine, TM);

break;

case QUADRATIC\_BEZIER:

bSuccess = drawBEZIER(strCommandLine, TM);

break;

case ROTATE:

bSuccess = rotateTm(strCommandLine, TM);

break;

case TRANSLATE:

bSuccess = translateTm(strCommandLine, TM);

break;

case SCALE:

bSuccess = scaleTm(strCommandLine, TM);

break;

case RESET\_TRANSFORM\_MATRIX:

resetTransformMatrix(TM); // all done :)

break;

default:

bSuccess = false;

dsprintf("unknown command!\n");

waitForEnterKey();

}

if (bSuccess) dsprintf("Command sent to robot!\n\n");

}

//---------------------------------------------------------------------------------------------------------------------

// DESCRIPTION: will draw a rectangle wiht the giben BL coener and TR coener

// ARGUMENTS: the string wiht the info for drawing the line

// RETURN VALUE: true if the command ran, false if not

bool drawRECTANGLE(char\* strCommandLine, double TM[][3])

{

const int paramaterNnumber = 5; // number of tokens to analize

bool goodCom = true; // is the command good or not

char RobotComand[COMMAND\_STRING\_ARRAY\_SIZE]; // Declare a char array for sending the robot commands

TOOL\_POSITION NODES[4] = { 0 }; // nodes for the 4 vertexs of the rectangle

LINE\_ID LINES[4] = { 0 }; // the lines connecting the 4 nodes

PATH\_CHECK lineCheck[4] = { 0 }; // the checks lines to see witch arm is more efeshent

JOINT\_ANGLES JointMove; // temp varible to make cose lines shorter

bool goodPath = true; // is the path valed

int armUse[4] = { LEFT }; // witch are is best to drw with

int whiteSpace = 0; // for formatting the error output

size\_t RESOLUTIONS[4] = { 0 }; // what resolution to draw each line

// the current resolution sate; H, M, L and the conectshions beatween the for nodes

int RESOLUTIONState = -1, conectshions[5] = { 0,1,2,3,0 };

double x1, x2, y1, y2, Line\_lenght; // temp varible to make code lines shorter

double theta1, theta2; // temp varible to make code lines shorter

double deltaY, deltaX; // temp varible to make code lines shorter

char\* token = NULL; // current token to analize

char\* pGarbage = NULL; // if there is any garbage

// go though and get all the paramaters

for (int i = 0; i < paramaterNnumber; i++)

{

// get the current token and put the rest in pGarbage

token = strtok\_s(NULL, seps, &strCommandLine); // get the current token

if (token != NULL)

{

// the current pos of the BL corner

if (i == 0) NODES[0].x = (double)strtod(token, &pGarbage);

if (i == 1) NODES[0].y = (double)strtod(token, &pGarbage);

// the current pos of the TR corner

if (i == 2) NODES[2].x = (double)strtod(token, &pGarbage);

if (i == 3) NODES[2].y = (double)strtod(token, &pGarbage);

if (i == 4) // get the line RESOLUTION

{

if (!strcmp(token, "HIGH")) RESOLUTIONState = RESOLUTION\_HIGH;

else if (!strcmp(token, "MEDIUM")) RESOLUTIONState = RESOLUTION\_MEDIUM;

else if (!strcmp(token, "LOW")) RESOLUTIONState = RESOLUTION\_LOW;

else

{

dsprintf("unknown command: %s\n\n", token);

goodCom = false;

}

}

if (pGarbage[0] != '\0') // theres was garbace traling the current string

{

dsprintf("Garbage found in command: %s\n", pGarbage);

dsprintf("Plese Fix command!");

goodCom = false;

waitForEnterKey();

break;

}

}

else // no token detecked

{

dsprintf("Missing data at index: %i\n", i+1);

goodCom = false;

waitForEnterKey();

break;

}

}

// the chage in x and y frome the BL and TR nodes

deltaX = NODES[2].x - NODES[0].x;

deltaY = NODES[2].y - NODES[0].y;

// make the BR node dektaX diffretn from the BL

NODES[1].x = NODES[0].x + deltaX;

NODES[1].y = NODES[0].y;

// make the BR node dektaY diffretn from the BL

NODES[3].x = NODES[0].x;

NODES[3].y = NODES[0].y + deltaY;

// loop though the conectshions array to determine the lengh of the lines beetween the nodes

for (int j = 0; j <= 3; j++)

{

x1 = NODES[conectshions[j]].x, x2 = NODES[conectshions[j + 1]].x; // get the start x and y cor

y1 = NODES[conectshions[j]].y, y2 = NODES[conectshions[j + 1]].y; // get the end c and y cor

Line\_lenght = sqrt(((x2 - x1) \* (x2 - x1)) + ((y2 - y1) \* (y2 - y1))); // calqulate the lenth of the arc

RESOLUTIONS[j] = getNumPathPoints(Line\_lenght, RESOLUTIONState);

}

// connect all the nodes acording to the conectshions array

for (int i = 0; i <= 3; i++)

{

LINES[conectshions[i]].NODES.start = NODES[conectshions[i]]; // the current node is the start of the line

LINES[conectshions[i]].NODES.end = NODES[conectshions[i + 1]]; // the nest line is the end of the line

}

// set all the paramaters of the lines, see drawLine

for (int i = 0; i < 4; i++)

{

LINES[i].NODES.RESOLUTION = RESOLUTIONS[i];

LINES[i].Points = getLineData(&LINES[i], TM);

lineCheck[i] = checkLinePath(&LINES[i]);

}

if (goodCom)

{

for (int i = 0; i < 4; i++)

{

// Determine which arm to use based on the angle differences

// if both arms can draw the line

if (lineCheck[i].bCanDraw[LEFT] && lineCheck[i].bCanDraw[RIGHT])

{

// is the left angle chage bigger then the right

if (lineCheck[i].dThetaDeg[LEFT] >= lineCheck[i].dThetaDeg[RIGHT])

{

armUse[i] = RIGHT;

}

// is the right angle chage bigger then the left

else if (lineCheck[i].dThetaDeg[RIGHT] > lineCheck[i].dThetaDeg[LEFT])

{

armUse[i] = LEFT;

}

}

// if only the left arm can draw the line

else if (lineCheck[i].bCanDraw[LEFT] && !lineCheck[i].bCanDraw[RIGHT])

{

armUse[i] = LEFT;

}

// of only the right line can draw the line

else if (!lineCheck[i].bCanDraw[LEFT] && lineCheck[i].bCanDraw[RIGHT])

{

armUse[i] = RIGHT;

}

else if ((lineCheck[i].bCanDraw[LEFT] == false) && (lineCheck[i].bCanDraw[RIGHT] == false))

{

dsprintf("RECTANGLE cant be draw\n");

dsprintf("NODES are:\n");

dsprintf("x1 = %lf\ty1 = % lf\n", LINES[0].NODES.start.x, LINES[0].NODES.start.y);

dsprintf("x2 = %lf\ty2 = %lf\n", LINES[2].NODES.end.x, LINES[2].NODES.end.y);

dsprintf("Plese Fix command!\n");

goodPath = false;

waitForEnterKey();

break;

}

}

}

if (goodCom && goodPath)

{

for (int j = 0; j < 4; j++)

{

for (int i = 0; i < LINES[j].NODES.RESOLUTION; i++)

{

JointMove = LINES[j].joints[armUse[j]][i]; // the joint moves

theta1 = JointMove.theta1Deg; // Retrieve the first theta1 value

theta2 = JointMove.theta2Deg; // Retrieve the first theta2 value

if (i == 0) robot.Send("PEN\_UP\n");

// Set the joint angles command based on the input joint move values

sprintf\_s(RobotComand, COMMAND\_STRING\_ARRAY\_SIZE, "ROTATE\_JOINT ANG1 %f ANG2 %f\n", theta1, theta2);

// Send the joint angles command to the robot

robot.Send(RobotComand);

// update the current robot angles

robotAngles(&JointMove, UPDATE\_CURRENT\_ANGLES);

if (i == 0) robot.Send("PEN\_DOWN\n");

}

}

}

return (goodPath && goodCom);

}

//---------------------------------------------------------------------------------------------------------------------

// DESCRIPTION: will draw a triangle wiht the given three vertexes

// ARGUMENTS: the string wiht the info for drawing the line

// RETURN VALUE: true if the command ran, false if not

bool drawTRIANGLE(char\* strCommandLine, double TM[][3])

{

const int paramaterNnumber = 7; // number of tokens to analize

bool goodCom = true; // is the command good or not

char RobotComand[COMMAND\_STRING\_ARRAY\_SIZE]; // Declare a char array for sending the robot commands

TOOL\_POSITION NODES[3] = { 0 }; // nodes for holding the 3 nodes

LINE\_ID LINES[3] = { 0 }; // the lines beatween the 3 node

PATH\_CHECK lineCheck[3] = { 0 }; // if the line is valled and woht line is better to deaw with

bool goodPath = true; // is the path valed

int armUse[3] = { LEFT }; // witch are is best to drw with

JOINT\_ANGLES JointMove; // temp varible to make code lines shorter

size\_t RESOLUTIONS[3] = { 0 }; // what resolution to draw each line

// the current resolution sate; H, M, L and the conectshions beatween the for nodes

int RESOLUTIONState = -1, conectshions[4] = { 0,1,2,0 };

double x1, x2, y1, y2, Line\_lenght;// temp varible to make code lines shorter

double theta1, theta2;// temp varible to make code lines shorter

char\* token = NULL; // current token to analize

char\* pGarbage = NULL; // if there is any garbage

// go though and get all the paramaters

for (int i = 0; i < paramaterNnumber; i++)

{

token = strtok\_s(NULL, seps, &strCommandLine); // get the current token

if (token != NULL)

{

// the current pos of the first node corner

if (i == 0) NODES[0].x = (double)strtod(token, &pGarbage);

if (i == 1) NODES[0].y = (double)strtod(token, &pGarbage);

// the current pos of the seond node corner

if (i == 2) NODES[1].x = (double)strtod(token, &pGarbage);

if (i == 3) NODES[1].y = (double)strtod(token, &pGarbage);

// the current pos of the therd node corner

if (i == 4) NODES[2].x = (double)strtod(token, &pGarbage);

if (i == 5) NODES[2].y = (double)strtod(token, &pGarbage);

if (i == 6) // get the line RESOLUTION

{

if (!strcmp(token, "HIGH")) RESOLUTIONState = RESOLUTION\_HIGH;

else if (!strcmp(token, "MEDIUM")) RESOLUTIONState = RESOLUTION\_MEDIUM;

else if (!strcmp(token, "LOW")) RESOLUTIONState = RESOLUTION\_LOW;

else

{

dsprintf("unknown command: %s\n\n", token);

goodCom = false;

}

}

if (pGarbage[0] != '\0') // theres was garbace traling the current string

{

dsprintf("Garbage found in command: %s\n", pGarbage);

dsprintf("Plese Fix command!\n");

goodCom = false;

waitForEnterKey();

break;

}

}

else

{

dsprintf("Missing data at index: %i\n", i+1);

goodCom = false;

waitForEnterKey();

break;

}

}

// loop though the conectshions array to determine the lengh of the lines beetween the nodes

for (int j = 0; j <= 2; j++)

{

x1 = NODES[conectshions[j]].x, x2 = NODES[conectshions[j + 1]].x; // get the start x and y cor

y1 = NODES[conectshions[j]].y, y2 = NODES[conectshions[j + 1]].y; // get the end c and y cor

Line\_lenght = sqrt(((x2 - x1) \* (x2 - x1)) + ((y2 - y1) \* (y2 - y1))); // calqulate the lenth of the arc

RESOLUTIONS[j] = getNumPathPoints(Line\_lenght, RESOLUTIONState);

}

// connect all the nodes acording to the conectshions array

for (int i = 0; i <= 2; i++)

{

LINES[conectshions[i]].NODES.start = NODES[conectshions[i]]; // the current node is the start of the line

LINES[conectshions[i]].NODES.end = NODES[conectshions[i + 1]]; // the nest line is the end of the line

}

// set all the paramaters of the lines, see drawLine

for (int i = 0; i < 3; i++)

{

LINES[i].NODES.RESOLUTION = RESOLUTIONS[i];

LINES[i].Points = getLineData(&LINES[i], TM);

lineCheck[i] = checkLinePath(&LINES[i]);

}

if (goodCom)

{

for (int i = 0; i < 3; i++)

{

// Determine which arm to use based on the angle differences

// if both arms can draw the line

if (lineCheck[i].bCanDraw[LEFT] && lineCheck[i].bCanDraw[RIGHT])

{

// is the left angle chage bigger then the right

if (lineCheck[i].dThetaDeg[LEFT] >= lineCheck[i].dThetaDeg[RIGHT])

{

armUse[i] = RIGHT;

}

// is the right angle chage bigger then the left

else if (lineCheck[i].dThetaDeg[RIGHT] > lineCheck[i].dThetaDeg[LEFT])

{

armUse[i] = LEFT;

}

}

// if only the left arm can draw the line

else if (lineCheck[i].bCanDraw[LEFT] && !lineCheck[i].bCanDraw[RIGHT])

{

armUse[i] = LEFT;

}

// of only the right line can draw the line

else if (!lineCheck[i].bCanDraw[LEFT] && lineCheck[i].bCanDraw[RIGHT])

{

armUse[i] = RIGHT;

}

else if ((lineCheck[i].bCanDraw[LEFT] == false) && (lineCheck[i].bCanDraw[RIGHT] == false))

{

dsprintf("TRIANGLE cant be draw\n");

dsprintf("NODES are:\nx1 = %lf\ty1 = %lf\n", LINES[0].NODES.start.x, LINES[0].NODES.start.y);

dsprintf("x2 = %lf\ty2 = %lf\n", LINES[1].NODES.start.x, LINES[1].NODES.end.y);

dsprintf("x2 = %lf\ty2 = %lf\n", LINES[2].NODES.start.x, LINES[2].NODES.end.y);

dsprintf("Plese Fix command!\n");

goodPath = false;

waitForEnterKey();

break;

}

}

}

if (goodCom && goodPath)

{

for (int j = 0; j < 3; j++)

{

for (int i = 0; i < LINES[j].NODES.RESOLUTION; i++)

{

JointMove = LINES[j].joints[armUse[j]][i]; // the joint moves

theta1 = JointMove.theta1Deg; // Retrieve the first theta1 value

theta2 = JointMove.theta2Deg; // Retrieve the first theta2 value

if (i == 0) robot.Send("PEN\_UP\n");

// Set the joint angles command based on the input joint move values

sprintf\_s(RobotComand, COMMAND\_STRING\_ARRAY\_SIZE, "ROTATE\_JOINT ANG1 %f ANG2 %f\n", theta1, theta2);

// Send the joint angles command to the robot

robot.Send(RobotComand);

// update the current robot angles

robotAngles(&JointMove, UPDATE\_CURRENT\_ANGLES);

if (i == 0) robot.Send("PEN\_DOWN\n");

}

}

}

return (goodPath && goodCom);

}

//---------------------------------------------------------------------------------------------------------------------

// DESCRIPTION: will draw a bezier curve wiht the given three points

// ARGUMENTS: the string wiht the info for drawing the line

// RETURN VALUE: true if the command ran, false if not

bool drawBEZIER(char\* strCommandLine, double TM[][3])

{

const int paramaterNnumber = 7; // number of tokens to analize

bool goodCom = true; // is the command good or not

char RobotComand[COMMAND\_STRING\_ARRAY\_SIZE]; // Declare a char array for sending the robot commands

BEZIER\_ID bezier = {}; // the current bezier strucked

PATH\_CHECK bezierCheck; // if the bezier is valled and woht line is better to deaw with

JOINT\_ANGLES JointMove; // temp varible to make code lines shorter

size\_t RESOLUTION = -1; // what resolution to draw each bezier

size\_t NP; // number of points in the line

int armUse = LEFT; // witch arem to use when drawing the line

bool goodPath = true;// did/can the robot run the command

double theta1, theta2; // temp varible to make code lines shorter

char\* token = NULL; // current token to analize

char\* pGarbage = NULL; // if there is any garbage

for (int i = 0; i < paramaterNnumber; i++)

{

token = strtok\_s(NULL, seps, &strCommandLine); // get the current token

if (token != NULL)

{

// the current pos of the start node corner

if (i == 0) bezier.NODES.points[0].x = (double)strtod(token, &pGarbage);

if (i == 1) bezier.NODES.points[0].y = (double)strtod(token, &pGarbage);

// the current pos of the conroll node corner

if (i == 2) bezier.NODES.points[1].x = (double)strtod(token, &pGarbage);

if (i == 3) bezier.NODES.points[1].y = (double)strtod(token, &pGarbage);

// the current pos of the end node corner

if (i == 4) bezier.NODES.points[2].x = (double)strtod(token, &pGarbage);

if (i == 5) bezier.NODES.points[2].y = (double)strtod(token, &pGarbage);

if (i == 6) // get the line RESOLUTION

{

if (!strcmp(token, "HIGH")) RESOLUTION = RESOLUTION\_HIGH;

else if (!strcmp(token, "MEDIUM")) RESOLUTION = RESOLUTION\_MEDIUM;

else if (!strcmp(token, "LOW")) RESOLUTION = RESOLUTION\_LOW;

else

{

dsprintf("unknown command: %s\n\n", token);

goodCom = false;

}

}

if (pGarbage[0] != '\0') // theres was garbace traling the current string

{

dsprintf("Garbage found in command: %s\n", pGarbage);

dsprintf("Plese Fix command!\n");

goodCom = false;

waitForEnterKey();

break;

}

}

else

{

dsprintf("Missing data at index: %i\n", i + 1);

goodCom = false;

waitForEnterKey();

break;

}

}

if (goodCom)

{

// get the lenth of the bezier curve

bezier.NODES.RESOLUTION = getNumPathPoints(getQuadraticBezierArcLength(bezier.NODES.points[0],

bezier.NODES.points[1], bezier.NODES.points[2]), RESOLUTION);

// get all the points for the bezier curve

bezier.Points = getBezierData(&bezier, TM);

// check witch arm is best to draw the like

bezierCheck = checkBezierPath(&bezier);

// get the numer of points to daw the bezier

NP = bezier.NODES.RESOLUTION;

// Determine which arm to use based on the angle differences

// if both carm can draw the line

if (bezierCheck.bCanDraw[LEFT] && bezierCheck.bCanDraw[RIGHT])

{

// is the left angle chage bigger then the right

if (bezierCheck.dThetaDeg[LEFT] >= bezierCheck.dThetaDeg[RIGHT])

{

armUse = RIGHT;

}

// is the right angle chage bigger then the left

else if (bezierCheck.dThetaDeg[RIGHT] > bezierCheck.dThetaDeg[LEFT])

{

armUse = LEFT;

}

}

// if only the left arm can draw

else if (bezierCheck.bCanDraw[LEFT] && !bezierCheck.bCanDraw[RIGHT])

{

armUse = LEFT;

}

// if only the right arm can draw

else if (!bezierCheck.bCanDraw[LEFT] && bezierCheck.bCanDraw[RIGHT])

{

armUse = RIGHT;

}

else // non of the arms can daw the line

{

goodPath = false;

dsprintf("BEZIER is not able to be drawn!\n\n"); // Notify the user that the line cannot be drawn

}

}

if (goodPath && goodCom)

{

for (int i = 0; i < NP; i++)

{

JointMove = bezier.joints[armUse][i]; // get the useable arm joint moves

theta1 = JointMove.theta1Deg; // Retrieve the first theta1 value

theta2 = JointMove.theta2Deg; // Retrieve the first theta2 value

if (i == 0) robot.Send("PEN\_UP\n");

// Set the joint angles command based on the input joint move values

sprintf\_s(RobotComand, COMMAND\_STRING\_ARRAY\_SIZE, "ROTATE\_JOINT ANG1 %f ANG2 %f\n", theta1, theta2);

// Send the joint angles command to the robot

robot.Send(RobotComand);

// update the current joint angles

robotAngles(&JointMove, UPDATE\_CURRENT\_ANGLES);

if (i == 0) robot.Send("PEN\_DOWN\n");

}

}

free(bezier.joints[LEFT]); // wil free the no long neede space

free(bezier.joints[RIGHT]);

return (goodPath && goodCom);

}

//---------------------------------------------------------------------------------------------------------------------

// DESCRIPTION: will rotate the main matrix with eveything else

// ARGUMENTS: the string wiht the info for drawing the line

// RETURN VALUE: true if the command ran, false if not

bool rotateTm(char\* strCommandLine, double TM[][3])

{

const int paramaterNnumber = 1; // how many tokes to be prossesd

bool goodCom = true; // is the command good or not

double rm[3][3] = { 0.0 }; // the rotations matrix

resetTransformMatrix(rm);

double angle = 0; // what angle to implnt the rotashion

char\* token = NULL;

char\* pGarbage = NULL; // if there is any garbage

for (int i = 0; i < paramaterNnumber; i++)

{

token = strtok\_s(NULL, seps, &strCommandLine); // get the current token

if (token != NULL)

{

// get the current angle and convert it to radions

if (i == 0) angle = degToRad((double)strtod(token, &pGarbage));

if (pGarbage[0] != '\0') // theres was garbace traling the current string

{

dsprintf("Garbage found in command: %s\n", pGarbage);

dsprintf("Plese Fix command!");

goodCom = false;

waitForEnterKey();

break;

}

}

else

{

dsprintf("Missing data at index: %i\n", i + 1);

goodCom = false;

waitForEnterKey();

break;

}

}

// set the rotations matrix

rm[0][0] = cos(angle);

rm[0][1] = -sin(angle);

rm[1][0] = sin(angle);

rm[1][1] = cos(angle);

// Multiply the rotations matrix with the current globle matrix

transformMatrixMultiply(TM, rm);

return goodCom;

}

//---------------------------------------------------------------------------------------------------------------------

// DESCRIPTION: will translete the main matrix with eveything else

// ARGUMENTS: the string wiht the info for drawing the line

// RETURN VALUE: true if the command ran, false if not

bool translateTm(char\* strCommandLine, double TM[][3])

{

const int paramaterNnumber = 2; // how many tokes to be prossesd

bool goodCom = true; // is the command good or not

double tm[3][3] = { 0.0 };

resetTransformMatrix(tm);

char\* token = NULL;

char\* pGarbage = NULL; // if there is any garbage

for (int i = 0; i < paramaterNnumber; i++)

{

token = strtok\_s(NULL, seps, &strCommandLine); // get the current token

if (token != NULL)

{

// assigne the set displatement to the translate matrix

if (i == 0) tm[0][2] = (double)strtod(token, &pGarbage);

if (i == 1) tm[1][2] = (double)strtod(token, &pGarbage);

if (pGarbage[0] != '\0') // theres was garbace traling the current string

{

dsprintf("Garbage found in command: %s\n", pGarbage);

dsprintf("Plese Fix command!");

goodCom = false;

waitForEnterKey();

break;

}

}

else

{

dsprintf("Missing data at index: %i\n", i + 1);

goodCom = false;

waitForEnterKey();

break;

}

}

// Multiply the translate matrix with the current globle matrix

transformMatrixMultiply(TM, tm);

return goodCom;

}

//---------------------------------------------------------------------------------------------------------------------

// DESCRIPTION: will scale the main matrix with eveything else

// ARGUMENTS: the string wiht the info for drawing the line

// RETURN VALUE: true if the command ran, false if not

bool scaleTm(char\* strCommandLine, double TM[][3])

{

const int paramaterNnumber = 2; // how many tokes to be prossesd

bool goodCom = true; // is the command good or not

double sm[3][3] = { 0.0 }; // the scale matrix

resetTransformMatrix(sm);

char\* token = NULL;

char\* pGarbage = NULL; // if there is any garbage

for (int i = 0; i < paramaterNnumber; i++)

{

token = strtok\_s(NULL, seps, &strCommandLine); // get the current token

if (token != NULL)

{

// assigne the set scaling to the scale matrix

if (i == 0) sm[0][0] = (double)strtod(token, &pGarbage);

if (i == 1) sm[1][1] = (double)strtod(token, &pGarbage);

if (pGarbage[0] != '\0') // theres was garbace traling the current string

{

dsprintf("Garbage found in command: %s\n", pGarbage);

dsprintf("Plese Fix command!");

goodCom = false;

waitForEnterKey();

break;

}

}

else

{

dsprintf("Missing data at index: %i\n", i + 1);

goodCom = false;

waitForEnterKey();

break;

}

}

// Multiply the scale matrix with the current globle matrix

transformMatrixMultiply(TM, sm);

return goodCom;

}

//---------------------------------------------------------------------------------------------------------------------

// DESCRIPTION: will draw the the arc wiht the given poins and angles

// ARGUMENTS: the string wiht the info for drawing the line

// RETURN VALUE: true if the command ran, false if not

bool drawArc(char\* strCommandLine, double TM[][3])

{

const int paramaterNnumber = 6; // number of tokens to analize

bool goodCom = true; // is the command good or not

char RobotComand[COMMAND\_STRING\_ARRAY\_SIZE]; // Declare a char array for sending the robot commands

// temp varibles to make code line shorter

double theta1, theta2;

double arcStart;

double arcEnd;

double radious;

// number of points to draw the arc

size\_t NP;

int armUse = LEFT; // witch arm to draw the line

bool goodPath = true; // did/can the arm be drawn

Arc\_ID arc = {};

PATH\_CHECK arcCheck;

JOINT\_ANGLES JointMove;

char\* token = NULL;

char\* pGarbage = NULL; // if there is any garbage

for (int i = 0; i < paramaterNnumber; i++)

{

token = strtok\_s(NULL, seps, &strCommandLine); // get the current token

if (token != NULL)

{

// set the firsat token to the the x start pos

if (i == 0) arc.NODES.cecter.x = (double)strtod(token, &pGarbage);

// set the secod token to the y start pos

if (i == 1) arc.NODES.cecter.y = (double)strtod(token, &pGarbage);

// set the therd token to the radious of the arc

if (i == 2) arc.NODES.radious = (double)strtod(token, &pGarbage);

// set the forth token to the x end pos

if (i == 3) arc.NODES.arcStart = (double)strtod(token, &pGarbage);

// set the fith token to the y end pos

if (i == 4) arc.NODES.arcEnd = (double)strtod(token, &pGarbage);

if (i == 5)

{

arcStart = arc.NODES.arcStart; // get the start of the arc

arcEnd = arc.NODES.arcEnd; // get the end of the arc

radious = arc.NODES.radious; // get the radious

// get the lenghth of arc

double arc\_lenght = fabs(radious \* (degToRad(arcEnd) - degToRad(arcStart)));

// set the RESOLUTION of the line

if (!strcmp(token, "HIGH")) arc.NODES.RESOLUTION = getNumPathPoints(arc\_lenght, RESOLUTION\_HIGH);

else if (!strcmp(token, "MEDIUM")) arc.NODES.RESOLUTION = getNumPathPoints(arc\_lenght, RESOLUTION\_MEDIUM);

else if (!strcmp(token, "LOW")) arc.NODES.RESOLUTION = getNumPathPoints(arc\_lenght, RESOLUTION\_LOW);

else

{

dsprintf("unknown command: %s\n\n", token);

goodCom = false;

}

}

if (pGarbage[0] != '\0') // theres was garbace traling the current string

{

dsprintf("Garbage found in command: %s\n", pGarbage);

dsprintf("Plese Fix command!\n ");

goodCom = false;

waitForEnterKey();

break;

}

}

else

{

dsprintf("Missing data at index: %i\n", i + 1);

goodCom = false;

waitForEnterKey();

break;

}

}

if (goodCom)

{

arc.Points = getArcData(&arc, TM);

arcCheck = checkArcPath(&arc);

NP = arc.NODES.RESOLUTION;

// Determine which arm to use based on the angle differences

// if both arms can draw the line

if (arcCheck.bCanDraw[LEFT] && arcCheck.bCanDraw[RIGHT])

{

// is the left angle chage bigger then the right

if (arcCheck.dThetaDeg[LEFT] >= arcCheck.dThetaDeg[RIGHT])

{

armUse = RIGHT;

}

// is the right angle chage bigger then the left

else if (arcCheck.dThetaDeg[RIGHT] > arcCheck.dThetaDeg[LEFT])

{

armUse = LEFT;

}

}

// if only the left arm can draw the line

else if (arcCheck.bCanDraw[LEFT] && !arcCheck.bCanDraw[RIGHT])

{

armUse = LEFT;

}

// of only the right line can draw the line

else if (!arcCheck.bCanDraw[LEFT] && arcCheck.bCanDraw[RIGHT])

{

armUse = RIGHT;

}

else // non of the arcs can draw the line

{

goodPath = false;

dsprintf("ARC is not able to be drawn!\n\n"); // Notify the user that the line cannot be drawn

}

}

if (goodPath && goodCom)

{

for (int i = 0; i < NP; i++)

{

JointMove = arc.joints[armUse][i]; // the joint moves

theta1 = JointMove.theta1Deg; // Retrieve the first theta1 value

theta2 = JointMove.theta2Deg; // Retrieve the first theta2 value

if (i == 0) robot.Send("PEN\_UP\n");

// Set the joint angles command based on the input joint move values

sprintf\_s(RobotComand, COMMAND\_STRING\_ARRAY\_SIZE, "ROTATE\_JOINT ANG1 %f ANG2 %f\n", theta1, theta2);

// Send the joint angles command to the robot

robot.Send(RobotComand);

// update the current robot angles

robotAngles(&JointMove, UPDATE\_CURRENT\_ANGLES);

if (i == 0) robot.Send("PEN\_DOWN\n");

}

}

free(arc.joints[LEFT]); // wil free the no long neede space

free(arc.joints[RIGHT]);

return (goodPath && goodCom);

}

//---------------------------------------------------------------------------------------------------------------------

// DESCRIPTION: will draw the the line wiht the given poins and angles

// ARGUMENTS: the string wiht the info for drawing the line

// RETURN VALUE: true if the command ran, false if not

bool drawLine(char\* strCommandLine, double TM[][3])

{

const int paramaterNnumber = 5; // how many tokes to be prossesd

bool goodCom = true; // is the command good or not

char RobotComand[COMMAND\_STRING\_ARRAY\_SIZE]; // Declare a char array for sending the robot commands

// tem varibles to make code lines shorter

double theta1, theta2;

double x1, x2, y1, y2, Line\_lenght;

size\_t NP; // number of points in the line

int armUse = LEFT; // witch arem to use when drawing the line

bool goodPath = true;// did/can the robot run the command

LINE\_ID line1 = {};

PATH\_CHECK lineCheck;

JOINT\_ANGLES JointMove;

char\* token = NULL;

char\* pGarbage = NULL; // if there is any garbage

for (int i = 0; i < paramaterNnumber; i++)

{

token = strtok\_s(NULL, seps, &strCommandLine); // get the current token

if (token != NULL)

{

// set the firsat token to the the x start pos

if (i == 0) line1.NODES.start.x = (double)strtod(token, &pGarbage);

// set the secod token to the y start pos

if (i == 1) line1.NODES.start.y = (double)strtod(token, &pGarbage);

// set the therd token to the x end pos

if (i == 2) line1.NODES.end.x = (double)strtod(token, &pGarbage);

// set the therd token to the y end pos

if (i == 3) line1.NODES.end.y = (double)strtod(token, &pGarbage);

if (i == 4) // get the line RESOLUTION

{

x1 = line1.NODES.start.x, x2 = line1.NODES.end.x; // get the start x and y cor

y1 = line1.NODES.start.y, y2 = line1.NODES.end.y; // get the end c and y cor

Line\_lenght = sqrt(((x2 - x1) \* (x2 - x1)) + ((y2 - y1) \* (y2 - y1))); // calqulate the lenth of the arc

if (!strcmp(token, "HIGH")) line1.NODES.RESOLUTION = getNumPathPoints(Line\_lenght, RESOLUTION\_HIGH);

else if (!strcmp(token, "MEDIUM"))line1.NODES.RESOLUTION = getNumPathPoints(Line\_lenght, RESOLUTION\_MEDIUM);

else if (!strcmp(token, "LPW")) line1.NODES.RESOLUTION = getNumPathPoints(Line\_lenght, RESOLUTION\_LOW);

else

{

dsprintf("unknown command: %s\n\n", token);

goodCom = false;

}

}

if (pGarbage[0] != '\0') // theres was garbace traling the current string

{

dsprintf("Garbage found in command: %s\n", pGarbage);

dsprintf("Plese Fix command!\n");

goodCom = false;

waitForEnterKey();

break;

}

}

else

{

dsprintf("Missing data at index: %i\n", i + 1);

goodCom = false;

waitForEnterKey();

break;

}

}

if (goodCom == true)

{

line1.Points = getLineData(&line1, TM);

lineCheck = checkLinePath(&line1);

NP = line1.NODES.RESOLUTION;

// Determine which arm to use based on the angle differences

// if both carm can draw the line

if (lineCheck.bCanDraw[LEFT] && lineCheck.bCanDraw[RIGHT])

{

// is the left angle chage bigger then the right

if (lineCheck.dThetaDeg[LEFT] >= lineCheck.dThetaDeg[RIGHT])

{

armUse = RIGHT;

}

// is the right angle chage bigger then the left

else if (lineCheck.dThetaDeg[RIGHT] > lineCheck.dThetaDeg[LEFT])

{

armUse = LEFT;

}

}

// if only the left arm can draw

else if (lineCheck.bCanDraw[LEFT] && !lineCheck.bCanDraw[RIGHT])

{

armUse = LEFT;

}

// if only the right arm can draw

else if (!lineCheck.bCanDraw[LEFT] && lineCheck.bCanDraw[RIGHT])

{

armUse = RIGHT;

}

else // non of the arms can daw the line

{

goodPath = false;

dsprintf("line is not able to be drawn!\n\n"); // Notify the user that the line cannot be drawn

}

}

if (goodPath && goodCom)

{

for (int i = 0; i < NP; i++)

{

JointMove = line1.joints[armUse][i]; // get the useable arm joint moves

theta1 = JointMove.theta1Deg; // Retrieve the first theta1 value

theta2 = JointMove.theta2Deg; // Retrieve the first theta2 value

if (i == 0) robot.Send("PEN\_UP\n");

// Set the joint angles command based on the input joint move values

sprintf\_s(RobotComand, COMMAND\_STRING\_ARRAY\_SIZE, "ROTATE\_JOINT ANG1 %f ANG2 %f\n", theta1, theta2);

// Send the joint angles command to the robot

robot.Send(RobotComand);

// update the current joint angles

robotAngles(&JointMove, UPDATE\_CURRENT\_ANGLES);

if (i == 0) robot.Send("PEN\_DOWN\n");

}

}

free(line1.joints[LEFT]); // wil free the no long neede space

free(line1.joints[RIGHT]);

return (goodPath && goodCom);

}

//---------------------------------------------------------------------------------------------------------------------

// DESCRIPTION: will set the motor speed of the robot

// ARGUMENTS: the string wiht the info for setting the speed

// RETURN VALUE: true if the command ran, false if not

bool setmotorSpeed(char\* strCommandLine)

{

bool goodCom = true; // did/can the robot run the command

char\* token = NULL; // the current token of the string

char\* pGarbage = NULL; // if there is any garbage

char RobotComand[COMMAND\_STRING\_ARRAY\_SIZE]; // Declare a char array for sending the robot commands

token = strtok\_s(NULL, seps, &strCommandLine);

if (token != NULL)

{

if (!strcmp(token, "HIGH")) sprintf\_s(RobotComand, COMMAND\_STRING\_ARRAY\_SIZE, "MOTOR\_SPEED HIGH\n");

else if (!strcmp(token, "MEDIUM")) sprintf\_s(RobotComand, COMMAND\_STRING\_ARRAY\_SIZE, "MOTOR\_SPEED MEDIUM\n");

else if (!strcmp(token, "LOW")) sprintf\_s(RobotComand, COMMAND\_STRING\_ARRAY\_SIZE, "MOTOR\_SPEED LOW\n");

else // theres was garbace traling the current string

{

dsprintf("Garbage found in command: %s\n", token);

dsprintf("Plese Fix command!\n");

goodCom = false;

waitForEnterKey();

}

}

else // theres was nothing in the token string

{

dsprintf("Missing data\n");

goodCom = false;

waitForEnterKey();

}

// Send the motor speed command to the robot

robot.Send(RobotComand);

return goodCom;

}

//---------------------------------------------------------------------------------------------------------------------

// DESCRIPTION: will set the joints to the given angles

// ARGUMENTS: the string wiht the info for seting the angles

// RETURN VALUE: true if the command ran, false if not

bool setJointAngles(char\* strCommandLine)

{

const int paramaterNnumber = 2; // the number of paramaters to deal with

bool goodCom = true; // is the command good or not

JOINT\_ANGLES angles = { 0,0 }; // holds the angles of the command

char\* token = NULL; // if there is any garbage

char\* pGarbage = NULL; // if there is any garbage

char RobotComand[COMMAND\_STRING\_ARRAY\_SIZE]; // Declare a char array for sending the robot commands

for (int i = 0; i < paramaterNnumber; i++)

{

token = strtok\_s(NULL, seps, &strCommandLine); // get the nest string of tect frome the input

if (token != NULL)

{

if (i == 0) angles.theta1Deg = radToDeg(mapAngle(degToRad((double)strtod(token, &pGarbage)))); // set the first token to the theta1Deg

if (i == 1) angles.theta2Deg = radToDeg(mapAngle(degToRad((double)strtod(token, &pGarbage)))); // set the second token to the theta2Deg

if (pGarbage[0] != '\0') // theres was garbace traling the current string

{

dsprintf("Garbage found in command: %s\n", pGarbage);

dsprintf("Plese Fix command!\n");

goodCom = false;

waitForEnterKey();

break;

}

}

else // theres was nothing in the token string

{

dsprintf("Missing data at index: %i\n", i + 1);

goodCom = false;

waitForEnterKey();

break;

}

}

if (goodCom && ((ABS\_THETA1\_DEG\_MAX < fabs(angles.theta1Deg) || (ABS\_THETA2\_DEG\_MAX < fabs(angles.theta2Deg)))))

{

goodCom = false;

printf("Joint angle is out of range!\n");

waitForEnterKey();

}

if (goodCom) // run if the command was good, can be run

{

// Set the joint angles command based on the input joint move values

sprintf\_s(RobotComand, COMMAND\_STRING\_ARRAY\_SIZE, "ROTATE\_JOINT ANG1 %lf ANG2 %lf\n", angles.theta1Deg, angles.theta2Deg);

// Send the joint angles command to the robot

robot.Send(RobotComand);

robotAngles(&angles, UPDATE\_CURRENT\_ANGLES);

}

return goodCom;

}

//---------------------------------------------------------------------------------------------------------------------

// DESCRIPTION: will set the joint to one given x,y cor

// ARGUMENTS: the string wiht the info for moveing the pen

// RETURN VALUE: true if the command ran, false if not

bool setPenPos(char\* strCommandLine)

{

const int paramaterNnumber = 2; // how many tokes to be prossesd

bool goodCom = true; // did/can the robot run the command

TOOL\_POSITION toolPos = { 0, 0 }; // the tool possion of the robot

JOINT\_ANGLES Ltheta = { 0,0 }; // the left arm angles

JOINT\_ANGLES Rtheta = { 0,0 }; // the right arm angles

char\* token = NULL; // the current token being analized

char\* pGarbage = NULL; // if there is any garbage

char RobotComand[COMMAND\_STRING\_ARRAY\_SIZE]; // Declare a char array for sending the robot commands

INVERSE\_SOLUTION isol; // holds the IK solution

// tem varibles to make lines of code shorter

double theta1Deg, theta2Deg, Lcom, Rcom = -1;

// loop though all the paramaters in the string

for (int i = 0; i < paramaterNnumber; i++)

{

// get the nect string to tokenize zeparated by the steps

token = strtok\_s(NULL, seps, &strCommandLine);

if (token != NULL)

{

if (i == 0) toolPos.x = (double)strtod(token, &pGarbage); // set the firsat token to the the x pos

if (i == 1) toolPos.y = (double)strtod(token, &pGarbage); // set the secod token to the y pos

if (pGarbage[0] != '\0') // theres was garbace traling the current string

{

dsprintf("Garbage found in command: %s\n", pGarbage);

dsprintf("Plese Fix command!");

goodCom = false; // the move cant be donre

waitForEnterKey();

break;

}

}

else

{

dsprintf("Missing data at index: %i\n", i + 1);

goodCom = false; // the move cant be donre

waitForEnterKey();

break;

}

}

if (goodCom == true)

{

isol = inverseKinematics(toolPos);

if (isol.bCanReach[LEFT] || isol.bCanReach[RIGHT])

{

// Check if the robot arm can reach the left point and leftWork flag is true

if (isol.bCanReach[LEFT] && isol.bCanReach[RIGHT])

{

// get the leaft and right arm joints

Ltheta = isol.jointAngles[LEFT];

Rtheta = isol.jointAngles[RIGHT];

// witch joins move is bigger, has the grates effect on effe

if (Ltheta.theta1Deg >= Ltheta.theta2Deg) Lcom = Ltheta.theta1Deg;

else Lcom = Ltheta.theta2Deg;

// witch joins move is bigger, has the grates effect on effe

if (Rtheta.theta1Deg >= Rtheta.theta2Deg) Lcom = Rtheta.theta1Deg;

else Lcom = Rtheta.theta2Deg;

// douse the left arm move more then the right

if (Lcom > Rcom)

{

// get the joint angles

theta1Deg = Rtheta.theta1Deg;

theta2Deg = Rtheta.theta2Deg;

}

// use the keft are

else

{

// get the joint angles

theta1Deg = Ltheta.theta1Deg;

theta2Deg = Ltheta.theta2Deg;

}

}

else if (isol.bCanReach[LEFT])

{

// get the IK info for sending thr joint angles

Ltheta = isol.jointAngles[LEFT];

// get the joint angles

theta1Deg = Ltheta.theta1Deg;

theta2Deg = Ltheta.theta2Deg;

}

else if (isol.bCanReach[RIGHT])

{

// get the IK info for sending thr joint angles

Rtheta = isol.jointAngles[RIGHT];

// get the joint angles

theta1Deg = Rtheta.theta1Deg;

theta2Deg = Rtheta.theta2Deg;

}

// Set the joint angles command based on the input joint move values

sprintf\_s(RobotComand, COMMAND\_STRING\_ARRAY\_SIZE, "ROTATE\_JOINT ANG1 %f ANG2 %f\n", theta1Deg, theta2Deg);

// Send the joint angles command to the robot

robot.Send(RobotComand);

// update the robot state

robotAngles(&Rtheta, UPDATE\_CURRENT\_ANGLES);

}

else // noe of the joints can reach the point

{

printf("point is unreachable: %lf, %lf\n", toolPos.x, toolPos.y);

dsprintf("Plese Fix command!\n");

waitForEnterKey();

goodCom = false; // the move cant be donre

}

}

return goodCom;

}

//---------------------------------------------------------------------------------------------------------------------

// DESCRIPTION: will tokenixe the given command and set the color of the pen

// ARGUMENTS: strCommandLine is a string only containg the pen color info

// RETURN VALUE: true if the command was able to be run, false if there was a problem

bool setPenColor(char\* strCommandLine)

{

const int paramaterNnumber = 3; // the number of paramaters to deal with

RGB colors = { 0,0,0 }; // holds the color

bool goodCom = true; // is the command good or not

char\* token = NULL; // the current token of the string

char\* pGarbage = NULL; // if there is any garbage

char RobotComand[COMMAND\_STRING\_ARRAY\_SIZE]; // Declare a char array for sending the robot commands

for (int i = 0; i < paramaterNnumber; i++)

{

token = strtok\_s(NULL, seps, &strCommandLine); // get the nest string of tect frome the input

if (token != NULL)

{

if (i == 0) colors.r = (int)strtol(token, &pGarbage, 10); // set the first token to red

if (i == 1) colors.g = (int)strtol(token, &pGarbage, 10);// set the second token to green

if (i == 2) colors.b = (int)strtol(token, &pGarbage, 10); // set the therid token to blue

if (pGarbage[0] != '\0') // theres was garbace traling the current string

{

dsprintf("Garbage found in command: %s\n", pGarbage);

dsprintf("Plese Fix command!\n");

goodCom = false;

waitForEnterKey();

break;

}

}

else // theres was nothing in the token string

{

dsprintf("Missing data at index: %i\n", i + 1);

goodCom = false;

waitForEnterKey();

break;

}

}

if ((colors.r < 0) || (colors.g < 0) || (colors.r < 0))

{

goodCom = false;

dsprintf("All RGB values must be >= 0!\n");

dsprintf("Plese Fix command!\n");

waitForEnterKey();

return goodCom;

}

if (goodCom) // run if the command was good, can be run

{

// Set the pen color command based on the color values

sprintf\_s(RobotComand, COMMAND\_STRING\_ARRAY\_SIZE, "PEN\_COLOR %d %d %d\n", colors.r, colors.g, colors.b);

// Send the pen color command to the robot

robot.Send(RobotComand);

}

return goodCom;

}

//--------------------------------------------------------------------------------------------------------------------

// DESCRIPTION: Find shoulder and elbow angles for both left/right arm configurations for a x,y coordinate

// ARGUMENTS: tp: the x,y coordinates of a tool tip position

// RETURN VALUE: a structure that contains the left/right arm joint angles corresponding to tp and a true/false

// value for each arm indicating if the coordinate is reachable.

INVERSE\_SOLUTION inverseKinematics(TOOL\_POSITION tp)

{

INVERSE\_SOLUTION isol = { ERROR\_VALUE, ERROR\_VALUE, ERROR\_VALUE, ERROR\_VALUE, false, false }; // solution values

double thata1R = -1.0, thata2R = -1.0, thata2aR = -1.0; // Declare theta1, theta2, RIGHT

double thata1L = -1.0, thata2L = -1.0, thata2aL = -1.0; // Declare theta1, theta2, RIGHT

double X = tp.x; // get the X pos

double Y = tp.y; // get the Y pos

double magnitude = sqrt(X \* X + Y \* Y); // Calculate the magnitude of the pos

double bata = atan2(Y, X);

double alpha = acos((pow(L2, 2) - pow(magnitude, 2) - pow(L1, 2)) / (-2.0 \* magnitude \* L1));

// Calculate theta1

thata1R = bata - alpha;

thata1L = bata + alpha;

// Calculate theta2a

thata2aR = atan2((Y - L1 \* sin(thata1R)), (X - L1 \* cos(thata1R)));

thata2aL = atan2((Y - L1 \* sin(thata1L)), (X - L1 \* cos(thata1L)));

//Calculate theta2

thata2R = thata2aR - thata1R;

thata2L = thata2aL - thata1L;

// Convert joint angles from radians to degrees and map them to the range [-180, 180] degrees

thata1R = radToDeg(mapAngle(thata1R));

thata2R = radToDeg(mapAngle(thata2R));

thata1L = radToDeg(mapAngle(thata1L));

thata2L = radToDeg(mapAngle(thata2L));

// assine the tem varibles to the solushion strut

isol.jointAngles[RIGHT].theta1Deg = thata1R;

isol.jointAngles[RIGHT].theta2Deg = thata2R;

isol.jointAngles[LEFT].theta1Deg = thata1L;

isol.jointAngles[LEFT].theta2Deg = thata2L;

// Check if the calculated joint angles are within the allowable range for the RIGHT arm

isol.bCanReach[RIGHT] = true;

if (fabs(thata1R) > ABS\_THETA1\_DEG\_MAX || fabs(thata2R) >= ABS\_THETA2\_DEG\_MAX)

{

isol.bCanReach[RIGHT] = false;

}

// Check if the calculated joint angles are within the allowable range for the LEFT arm

isol.bCanReach[LEFT] = true;

if (fabs(thata1L) > ABS\_THETA1\_DEG\_MAX || fabs(thata2L) >= ABS\_THETA2\_DEG\_MAX)

{

isol.bCanReach[LEFT] = false;

}

//Check if the calculated magnitude is within the allowable range

if (magnitude > LMAX || magnitude < LMIN)

{

isol.bCanReach[RIGHT] = false;

isol.bCanReach[LEFT] = false;

}

return isol;

}

//---------------------------------------------------------------------------------------------------------------------

// DESCRIPTION: will check witch arm cann draw the line

// ARGUMENTS: the line\_ID with all the line info

// RETURN VALUE: PATH\_CHECK wil the info on witch line will be dest to draw the line

PATH\_CHECK checkLinePath(LINE\_ID\* line)

{

// Get the number of points in the line

size\_t NP = line->numPoints;

// Declare variables to store joint angles

JOINT\_ANGLES\* joinsL, \* joinsR;

JOINT\_ANGLES PreAngle;

double theta1Deg = 0, theta1DegPre = 0, theta2Deg = 0, theta2DegPre = 0;

robotAngles(&PreAngle, GET\_CURRENT\_ANGLES);

theta1DegPre = PreAngle.theta1Deg;

theta2DegPre = PreAngle.theta2Deg;

// Allocate memory for the joint angles

joinsL = (JOINT\_ANGLES\*)calloc(NP, sizeof(JOINT\_ANGLES));

joinsR = (JOINT\_ANGLES\*)calloc(NP, sizeof(JOINT\_ANGLES));

// If memory allocation fails, print an error message and exit the program

if ((joinsL == NULL) || (joinsR == NULL))

{

dsprintf("Memory allocation problem, checkPath()\n");

waitForEnterKey();

exit(0);

}

// Initialize path check struct

PATH\_CHECK path = { false, false, 0, 0 };

INVERSE\_SOLUTION isol;

// Initialize flags to track whether the robot arm can reach the left and right points

bool leftWork = true, rightWork = true;

// Loop through each point in the line

for (int i = 0; i < NP; i++)

{

// Get inverse kinematics solution for the current point

isol = inverseKinematics(line->Points[i]);

// Check if the robot arm can reach the left point and leftWork flag is true

if (isol.bCanReach[LEFT] && leftWork)

{

// Set flag to indicate that robot arm can draw the path using the left arm

path.bCanDraw[LEFT] = true;

// Get joint angles for the left arm

theta1Deg = isol.jointAngles[LEFT].theta1Deg;

theta2Deg = isol.jointAngles[LEFT].theta2Deg;

// Calculate the change in joint angles since the previous point

double change = max(fabs(theta1Deg - theta1DegPre), fabs(theta2Deg - theta2DegPre));

path.dThetaDeg[LEFT] += max(fabs(theta1Deg - theta1DegPre), fabs(theta2Deg - theta2DegPre));

// Store joint angles for the left arm

joinsL[i] = isol.jointAngles[LEFT];

// Update previous joint angles

theta1DegPre = theta1Deg;

theta2DegPre = theta2Deg;

}

else

{

// Set flag to indicate that robot arm cannot draw the path using the left arm

path.bCanDraw[LEFT] = false;

// Set leftWork flag to false to indicate that the left arm cannot reach the current point

leftWork = false;

}

}

robotAngles(&PreAngle, GET\_CURRENT\_ANGLES);

theta1DegPre = PreAngle.theta1Deg;

theta2DegPre = PreAngle.theta2Deg;

// Loop through each point in the line

for (int i = 0; i < NP; i++)

{

// Get inverse kinematics solution for the current point

isol = inverseKinematics(line->Points[i]);

// Check if the robot arm can reach the right point and rightWork flag is true

if (isol.bCanReach[RIGHT] && rightWork)

{

// Set flag to indicate that robot arm can draw the path using the right arm

path.bCanDraw[RIGHT] = true;

// Get joint angles for the right arm

theta1Deg = isol.jointAngles[RIGHT].theta1Deg;

theta2Deg = isol.jointAngles[RIGHT].theta2Deg;

// Calculate the change in joint angles since the previous point

path.dThetaDeg[RIGHT] += max(fabs(theta1Deg - theta1DegPre), fabs(theta2Deg - theta2DegPre));

// Store joint angles for the RIGH arm

joinsR[i] = isol.jointAngles[RIGHT];

// Update previous joint angles

theta1DegPre = theta1Deg;

theta2DegPre = theta2Deg;

}

else

{

// Set flag to indicate that robot arm cannot draw the path using the left arm

path.bCanDraw[RIGHT] = false;

// Set leftWork flag to false to indicate that the left arm cannot reach the current point

rightWork = false;

}

}

// writes the joint angles to the line

line->joints[LEFT] = joinsL;

line->joints[RIGHT] = joinsR;

// return the checksed path info of witch arm shouls sraw the line

return path;

}

//---------------------------------------------------------------------------------------------------------------------

// DESCRIPTION: will check witch arm cann draw the arc

// ARGUMENTS: the arc\_ID with all the line info

// RETURN VALUE: PATH\_CHECK wil the info on witch line will be dest to draw the arc

PATH\_CHECK checkArcPath(Arc\_ID\* arc)

{

// Get the number of points in the line

size\_t NP = arc->numPoints;

// Declare variables to store joint angles

JOINT\_ANGLES\* joinsL, \* joinsR;

double theta1Deg, theta1DegPre = 0, theta2Deg, theta2DegPre = 0;

// Allocate memory for the joint angles

joinsL = (JOINT\_ANGLES\*)calloc(NP, sizeof(JOINT\_ANGLES));

joinsR = (JOINT\_ANGLES\*)calloc(NP, sizeof(JOINT\_ANGLES));

// If memory allocation fails, print an error message and exit the program

if ((joinsL == NULL) || (joinsR == NULL))

{

dsprintf("Memory allocation problem, checkPath()\n");

waitForEnterKey();

exit(0);

}

// Initialize path check struct

PATH\_CHECK path = { false, false, 0, 0 };

INVERSE\_SOLUTION isol;

// Initialize flags to track whether the robot arm can reach the left and right points

bool leftWork = true, rightWork = true;

// Loop through each point in the line

for (int i = 0; i < NP; i++)

{

// Get inverse kinematics solution for the current point

isol = inverseKinematics(arc->Points[i]);

// Check if the robot arm can reach the left point and leftWork flag is true

if (isol.bCanReach[LEFT] && leftWork)

{

// Set flag to indicate that robot arm can draw the path using the left arm

path.bCanDraw[LEFT] = true;

// Get joint angles for the left arm

theta1Deg = isol.jointAngles[LEFT].theta1Deg;

theta2Deg = isol.jointAngles[LEFT].theta2Deg;

// Calculate the change in joint angles since the previous point

if (theta1Deg >= theta2Deg)

{

path.dThetaDeg[LEFT] += (theta1Deg - theta1DegPre);

}

else if (theta2Deg > theta1Deg)

{

path.dThetaDeg[LEFT] += (theta2Deg - theta2DegPre);

}

// Store joint angles for the left arm

joinsL[i] = isol.jointAngles[LEFT];

// Update previous joint angles

theta1DegPre = theta1Deg;

theta2DegPre = theta2Deg;

}

else

{

// Set flag to indicate that robot arm cannot draw the path using the left arm

path.bCanDraw[LEFT] = false;

// Set leftWork flag to false to indicate that the left arm cannot reach the current point

leftWork = false;

}

// Check if the robot arm can reach the right point and rightWork flag is true

if (isol.bCanReach[RIGHT] && rightWork)

{

// Set flag to indicate that robot arm can draw the path using the right arm

path.bCanDraw[RIGHT] = true;

// Get joint angles for the right arm

theta1Deg = isol.jointAngles[RIGHT].theta1Deg;

theta2Deg = isol.jointAngles[RIGHT].theta2Deg;

// Calculate the change in joint angles since the previous point

if (theta1Deg >= theta2Deg)

{

path.dThetaDeg[RIGHT] += (theta1Deg - theta1DegPre);

}

else if (theta2Deg > theta1Deg)

{

path.dThetaDeg[RIGHT] += (theta2Deg - theta2DegPre);

}

// Store joint angles for the RIGH arm

joinsR[i] = isol.jointAngles[RIGHT];

// Update previous joint angles

theta1DegPre = theta1Deg;

theta2DegPre = theta2Deg;

}

else

{

// Set flag to indicate that robot arm cannot draw the path using the left arm

path.bCanDraw[RIGHT] = false;

// Set leftWork flag to false to indicate that the left arm cannot reach the current point

rightWork = false;

}

}

// writes the joint angles to the line

arc->joints[LEFT] = joinsL;

arc->joints[RIGHT] = joinsR;

// return the checksed path info of witch arm shouls sraw the line

return path;

}

//---------------------------------------------------------------------------------------------------------------------

// DESCRIPTION: will check witch arm cann draw the bezier

// ARGUMENTS: the bezier\_ID with all the bezier info

// RETURN VALUE: PATH\_CHECK wil the info on witch line will be dest to draw the bezier

PATH\_CHECK checkBezierPath(BEZIER\_ID\* bezier)

{

// Get the number of points in the line

size\_t NP = bezier->numPoints;

// Declare variables to store joint angles

JOINT\_ANGLES\* joinsL, \* joinsR;

double theta1Deg, theta1DegPre = 0, theta2Deg, theta2DegPre = 0;

// Allocate memory for the joint angles

joinsL = (JOINT\_ANGLES\*)calloc(NP, sizeof(JOINT\_ANGLES));

joinsR = (JOINT\_ANGLES\*)calloc(NP, sizeof(JOINT\_ANGLES));

// If memory allocation fails, print an error message and exit the program

if ((joinsL == NULL) || (joinsR == NULL))

{

dsprintf("Memory allocation problem, checkPath()\n");

waitForEnterKey();

exit(0);

}

// Initialize path check struct

PATH\_CHECK path = { false, false, 0, 0 };

INVERSE\_SOLUTION isol;

// Initialize flags to track whether the robot arm can reach the left and right points

bool leftWork = true, rightWork = true;

// Loop through each point in the line

for (int i = 0; i < NP; i++)

{

// Get inverse kinematics solution for the current point

isol = inverseKinematics(bezier->Points[i]);

// Check if the robot arm can reach the left point and leftWork flag is true

if (isol.bCanReach[LEFT] && leftWork)

{

// Set flag to indicate that robot arm can draw the path using the left arm

path.bCanDraw[LEFT] = true;

// Get joint angles for the left arm

theta1Deg = isol.jointAngles[LEFT].theta1Deg;

theta2Deg = isol.jointAngles[LEFT].theta2Deg;

// Calculate the change in joint angles since the previous point

if (theta1Deg >= theta2Deg)

{

path.dThetaDeg[LEFT] += (theta1Deg - theta1DegPre);

}

else if (theta2Deg > theta1Deg)

{

path.dThetaDeg[LEFT] += (theta2Deg - theta2DegPre);

}

// Store joint angles for the left arm

joinsL[i] = isol.jointAngles[LEFT];

// Update previous joint angles

theta1DegPre = theta1Deg;

theta2DegPre = theta2Deg;

}

else

{

// Set flag to indicate that robot arm cannot draw the path using the left arm

path.bCanDraw[LEFT] = false;

// Set leftWork flag to false to indicate that the left arm cannot reach the current point

leftWork = false;

}

// Check if the robot arm can reach the right point and rightWork flag is true

if (isol.bCanReach[RIGHT] && rightWork)

{

// Set flag to indicate that robot arm can draw the path using the right arm

path.bCanDraw[RIGHT] = true;

// Get joint angles for the right arm

theta1Deg = isol.jointAngles[RIGHT].theta1Deg;

theta2Deg = isol.jointAngles[RIGHT].theta2Deg;

// Calculate the change in joint angles since the previous point

if (theta1Deg >= theta2Deg)

{

path.dThetaDeg[RIGHT] += (theta1Deg - theta1DegPre);

}

else if (theta2Deg > theta1Deg)

{

path.dThetaDeg[RIGHT] += (theta2Deg - theta2DegPre);

}

// Store joint angles for the RIGH arm

joinsR[i] = isol.jointAngles[RIGHT];

// Update previous joint angles

theta1DegPre = theta1Deg;

theta2DegPre = theta2Deg;

}

else

{

// Set flag to indicate that robot arm cannot draw the path using the left arm

path.bCanDraw[RIGHT] = false;

// Set leftWork flag to false to indicate that the left arm cannot reach the current point

rightWork = false;

}

}

// writes the joint angles to the line

bezier->joints[LEFT] = joinsL;

bezier->joints[RIGHT] = joinsR;

// return the checksed path info of witch arm shouls sraw the line

return path;

}

//---------------------------------------------------------------------------------------------------------------------

// DESCRIPTION: will get all the intermideate points beewten the mains line nodes

// ARGUMENTS: the line ID and the main transform matrix

// RETURN VALUE: an address of where the points are

TOOL\_POSITION\* getLineData(LINE\_ID\* line, double TM[3][3])

{

double x1 = line->NODES.start.x, x2 = line->NODES.end.x;

double y1 = line->NODES.start.y, y2 = line->NODES.end.y;

size\_t NP = line->NODES.RESOLUTION;

TOOL\_POSITION\* Line\_cords\_FT;

TOOL\_POSITION newData = {0};

TOOL\_POSITION Line\_cords\_BT = { 0 };

Line\_cords\_FT = (TOOL\_POSITION\*)calloc(NP, sizeof(TOOL\_POSITION));

if (Line\_cords\_FT == NULL)

{

dsprintf("Memmory allocashion problem, getLineData()");

exit(0);

}

for (int i = 0; i < NP; i++)

{

// Compute the current x and y point value

Line\_cords\_BT.x = x1 + ((x2 - x1) \* ((double)i) / ((double)NP - 1));

Line\_cords\_BT.y = y1 + ((y2 - y1) \* ((double)i) / ((double)NP - 1));

newData = transform(TM, Line\_cords\_BT);

Line\_cords\_FT[i] = newData;

}

line->numPoints = NP;

return Line\_cords\_FT;

}

//---------------------------------------------------------------------------------------------------------------------

// DESCRIPTION: will get all the intermideate points beewten the main arc nodes

// ARGUMENTS: the arc ID and the main transform matrix

// RETURN VALUE: an address of where the points are

TOOL\_POSITION\* getArcData(Arc\_ID\* arc, double TM[][3])

{

// temp varibles to shorten the cose lenght

double xCen = arc->NODES.cecter.x, yCen = arc->NODES.cecter.y;

double thetaN = -1;

double arcStart = arc->NODES.arcStart;

double arcEnd = arc->NODES.arcEnd;

double radious = arc->NODES.radious;

// the lenth of the arc

//double arc\_lenght = radious \* (fabs(arcEnd) - fabs(arcStart));

// number of points to draw the arc

size\_t NP = arc->NODES.RESOLUTION;

TOOL\_POSITION\* arc\_cords\_AT;

TOOL\_POSITION arc\_cords\_BT = { 0 };

// allowcant the memory for the points

arc\_cords\_AT = (TOOL\_POSITION\*)calloc(NP, sizeof(TOOL\_POSITION));

if (arc\_cords\_AT == NULL) // coud not allocare the memory

{

dsprintf("Memmory allocashion problem, getLineData()");

waitForEnterKey();

exit(0);

}

for (int i = 0; i < NP; i++)

{

// Compute the current x and y point value

thetaN = arcStart + (arcEnd - arcStart) \* ((double)i) / ((double)NP - 1);

arc\_cords\_BT.x = xCen + radious \* cos(degToRad(thetaN));

arc\_cords\_BT.y = yCen + radious \* sin(degToRad(thetaN));

arc\_cords\_AT[i] = transform(TM, arc\_cords\_BT);

}

arc->numPoints = NP;

return arc\_cords\_AT;

}

//---------------------------------------------------------------------------------------------------------------------

// DESCRIPTION: will get all the intermideate points beewten the main bezier nodes

// ARGUMENTS: the bezier ID and the main transform matrix

// RETURN VALUE: an address of where the points are

TOOL\_POSITION\* getBezierData(BEZIER\_ID\* bezier, double TM[][3])

{

double x1, x2, x3, y1, y2, y3;

double t;

size\_t NP = bezier->NODES.RESOLUTION;

TOOL\_POSITION\* bezier\_cords\_FT;

TOOL\_POSITION bezier\_cords\_BT = { 0 };

x1 = bezier->NODES.points[0].x;

y1 = bezier->NODES.points[0].y;

x2 = bezier->NODES.points[1].x;

y2 = bezier->NODES.points[1].y;

x3 = bezier->NODES.points[2].x;

y3 = bezier->NODES.points[2].y;

bezier\_cords\_FT = (TOOL\_POSITION\*)calloc(NP, sizeof(TOOL\_POSITION));

if (bezier\_cords\_FT == NULL)

{

dsprintf("Memmory allocashion problem, getLineData()");

exit(0);

}

for (int i = 0; i < NP; i++)

{

t = (double)i / (double)(NP - 1);

bezier\_cords\_BT.x = ((1.0 - t) \* (1.0 - t)) \* x1 + 2.0 \* (1.0 - t)\*t \* x2 + (t \* t) \* x3;

bezier\_cords\_BT.y = ((1.0 - t) \* (1.0 - t)) \* y1 + 2.0 \* (1.0 - t)\*t \* y2 + (t \* t) \* y3;

bezier\_cords\_FT[i] = transform(TM, bezier\_cords\_BT);

}

bezier->numPoints = NP;

return bezier\_cords\_FT;

}

//---------------------------------------------------------------------------------------------------------------------

// DESCRIPTION: Calculates the length of a quadratic Bezier Curve

// ARGUMENTS: P0: coordinates of start of curve.

// P2: coordinates of end of curve

// P1: control point coordinates

// RETURN VALUE: length of the quadratic Bezier Curve

double getQuadraticBezierArcLength(TOOL\_POSITION P0, TOOL\_POSITION P1, TOOL\_POSITION P2)

{

double xn, yn, xnm1, ynm1; // points along the curve

double t, len = 0; // t = bezier parameter, len = length of bezier curve

size\_t n; // point index

const size\_t NP = 1000; // number of points.

// subdivide the curve into a number of point-to-point straight line segments.

// compute the length of each segment and add them up

// starting point

xnm1 = P0.x;

ynm1 = P0.y;

for (n = 1; n < NP; n++)

{

t = (double)n / (double)(NP - 1);

xn = (1.0 - t) \* (1.0 - t) \* P0.x + 2.0 \* (1.0 - t) \* t \* P1.x + t \* t \* P2.x;

yn = (1.0 - t) \* (1.0 - t) \* P0.y + 2.0 \* (1.0 - t) \* t \* P1.y + t \* t \* P2.y;

len += sqrt(pow(xn - xnm1, 2) + pow(yn - ynm1, 2));

xnm1 = xn;

ynm1 = yn;

}

return len;

}

//---------------------------------------------------------------------------------------------------------------------

// Tranforms the tool position coordinates based on the current tranformation matrix.

// INPUTS: TM: the 3x3 transform matrix

// RETURN: tranformed tool position

TOOL\_POSITION transform(const double TM[][3], TOOL\_POSITION tp)

{

TOOL\_POSITION tpt; // transformed tool position

// matrix multiply transformation

tpt.x = tp.x \* TM[0][0] + tp.y \* TM[0][1] + TM[0][2];

tpt.y = tp.x \* TM[1][0] + tp.y \* TM[1][1] + TM[1][2];

return tpt;

}

//---------------------------------------------------------------------------------------------------------------------

// Resets the tranform matrix to the unit matrix. x, y points will no longer be transformed in inverseKinematics

// INPUTS: the 3x3 transform matrix

// RETURN: nothing

void resetTransformMatrix(double TM[][3]) // resets to unit matrix

{

int r, c; // matrix row, column indexes

for (r = 0; r < 3; r++)

{

for (c = 0; c < 3; c++)

{

TM[r][c] = (r == c?1.0:0.0);

}

}

}

//---------------------------------------------------------------------------------------------------------------------

// Premultiplies the transform matrix by matrix M. M is the rotation matrix, translation matrix, or the scaling matrix

// INPUTS: TM: the 3x3 transform matrix, M the premultiplier matrix

// RETURN: nothing

void transformMatrixMultiply(double TM[][3], const double M[][3])

{

int r, c, cc; // row, column indexes

double TMM[3][3]; // temp matrix

for (r = 0; r < 3; r++)

{

for (c = 0; c < 3; c++)

{

TMM[r][c] = 0.0; // set element to zero

for (cc = 0; cc < 3; cc++) // accumulate the multiples

{

TMM[r][c] += M[r][cc] \* TM[cc][c];

}

}

}

for (r = 0; r < 3; r++) // copy temp matrix to TM

{

for (c = 0; c < 3; c++)

{

TM[r][c] = TMM[r][c];

}

}

}

//---------------------------------------------------------------------------------------------------------------------

// DESCRIPTION: parses a command string that contains CYCLE\_PEN\_COLORS and sends command to robot if data ok.

// ARGUMENTS: strLine: A file line string.

// RETURN VALUE: true if command sent to robot, false if not.

bool setCyclePenColors(char\* strLine)

{

char\* tok = NULL, \* nextTok = NULL; // for tokenizing the line string

char cmd[COMMAND\_STRING\_ARRAY\_SIZE]; // command string for sprintf\_s

tok = strtok\_s(strLine, seps, &nextTok); // CYCLE\_PEN\_COLORS keyword (discarded)

tok = strtok\_s(NULL, seps, &nextTok); // parameter should be "ON" or "OFF"

if (tok == NULL)

{

return false;

}

// Got token. Check if token is "ON" or "OFF"

if (strcmp(tok, "ON") != 0 && strcmp(tok, "OFF") != 0)

{

dsprintf("Invalid parameter for CYCLE\_PEN\_COLORS! Must be ON or OFF.\n\n");

return false;

}

// all good. Send command.

sprintf\_s(cmd, COMMAND\_STRING\_ARRAY\_SIZE, "CYCLE\_PEN\_COLORS %s\n", tok);

robot.Send(cmd);

return true;

}

//---------------------------------------------------------------------------------------------------------------------

// DESCRIPTION: will fined what index the command is

// ARGUMENTS: strLine: A file line string.

// RETURN VALUE: the index of the command

int getCommandIndex(const char\* strLine)

{

char strLine2[MAX\_LINE\_SIZE];

strcpy\_s(strLine2, MAX\_LINE\_SIZE, strLine);

// fine witch command maches the know commands

for (int n = 0; n < NUM\_COMMANDS; n++)

{

// is the given string the same as the N'th command

if (strstr(strLine, m\_Commands[n].strCommand) != NULL)

{

return m\_Commands[n].index;

}

}

// if the command was not found

return -1;

}

//---------------------------------------------------------------------------------------------------------------------

// DESCRIPTION: get or update current robot shoulder and elbow angles

// ARGUMENTS: pAngles: shoulder/joint angles.

// getOrUpdate: set to UPDATE\_CURRENT\_ANGLES to update the current angles

// set to GET\_CURRENT\_ANGLES to retrieve the current angles

// RETURN VALUE: none

void robotAngles(JOINT\_ANGLES\* pAngles, int getOrUpdate)

{

static JOINT\_ANGLES currentAngles = { 0.0, 0.0 }; // NOTE: robot must be in home position when program starts!

if (pAngles == NULL) // safety

{

dsprintf("NULL JOINT\_ANGLES pointer! (robotAngles)");

return;

}

if (getOrUpdate == UPDATE\_CURRENT\_ANGLES)

currentAngles = \*pAngles;

else if (getOrUpdate == GET\_CURRENT\_ANGLES)

\*pAngles = currentAngles;

else

dsprintf("Unknown value for getOrUpdate (robotAngles)");

}

//---------------------------------------------------------------------------------------------------------------------

// DESCRIPTION: Pauses the robot then clears everthing after user presses ENTER

// ARGUMENTS: none

// RETURN VALUE: none

void pauseRobotThenClear()

{

waitForEnterKey();

system("cls");

robot.Send("HOME\n");

robot.Send("CLEAR\_TRACE\n");

robot.Send("PEN\_COLOR 0 0 255\n");

robot.Send("CLEAR\_REMOTE\_COMMAND\_LOG\n");

robot.Send("CLEAR\_POSITION\_LOG\n");

}

//---------------------------------------------------------------------------------------------------------------------

// DESCRIPTION: Maps an angle in radians into a an equivalent angle understood by the robot (-PI <= ang <= +PI)

// ARGUMENTS: ang: the angle in radians

// RETURN VALUE: the mapped angle in radians

double mapAngle(double angRad)

{

angRad = fmod(angRad, 2.0 \* PI); // put in range -2\*PI <= ang <= +2\*PI

// map into range -PI <= ang <= +PI

if (angRad > PI)

angRad -= 2.0 \* PI;

else if (angRad < -PI)

angRad += 2.0 \* PI;

return angRad;

}

//---------------------------------------------------------------------------------------------------------------------

// DESCRIPTION: Returns angle in degrees from input angle in radian

// ARGUMENTS: angDeg: angle in degrees

// RETURN VALUE: angle in radians

double degToRad(double angDeg)

{

return (PI / 180.0) \* angDeg;

}

//---------------------------------------------------------------------------------------------------------------------

// DESCRIPTION: Returns angle in radians from input angle in degrees

// ARGUMENTS: angRad: angle in radians

// RETURN VALUE: angle in degrees

double radToDeg(double angRad)

{

return (180.0 / PI) \* angRad;

}

//---------------------------------------------------------------------------------------------------------------------

// DESCRIPTION: This function flushes the input buffer to avoid scanf issues

// ARGUMENTS: none

// RETURN VALUE: false if nothing or only '\n' in stdin. true if extra keystrokes precede the '\n'.

// Good for detecting left over garbage from scanf\_s in the input buffer

bool flushInputBuffer()

{

int ch; // temp character variable

bool bHasGarbage = false;

// exit loop when all characters are flushed

while ((ch = getchar()) != '\n' && ch != EOF)

{

if (!bHasGarbage) bHasGarbage = true;

}

return bHasGarbage;

}

//---------------------------------------------------------------------------------------------------------------------

// DESCRIPTION: Waits for user to press enter. flushes stdin if keystrokes precede enter

// ARGUMENTS: none

// RETURN VALUE: none

void waitForEnterKey()

{

int ch;

if ((ch = getchar()) != EOF && ch != '\n') flushInputBuffer();

}

//---------------------------------------------------------------------------------------------------------------------

// DESCRIPTION: computes nearest integer to given double

// ARGUMENTS: d: double value

// RETURN VALUE: nearest int

int nint(double d)

{

return (int)floor(d + 0.5);

}

//---------------------------------------------------------------------------------------------------------------------

// DESCRIPTION: print a solid line to the console

// ARGUMENTS: length of line in characters

// RETURN VALUE: none

void printHLine(int N)

{

int n;

for (n = 0; n < N; n++)

{

// can't use dsprintf because characters are different because code pages are different

// console = code page 437, file = code page 1252

printf("%c", HL);

fprintf(flog, "%c", FHL);

}

dsprintf("\n");

}

//---------------------------------------------------------------------------------------------------------------------

// DESCRIPTION: makes a string all upper case characters

// ARGUMENTS: str: the string memory address

// RETURN VALUE: none

void makeStringUpperCase(char\* str)

{

if (str == NULL) return; // safety!

for (size\_t i = 0; i < strlen(str); i++) str[i] = (char)toupper(str[i]);

}

//---------------------------------------------------------------------------------------------------------------------

// DESCRIPTION: prints to both a file and to the console

// ARGUMENTS: f: the file handle

// fmt, ...: for variable number of parameters

// RETURN VALUE: the number of characters printed

int dsprintf(char const\* fmt, ...)

{

va\_list args;

int n1 = -1, n2 = -1;

if (flog != NULL)

{

va\_start(args, fmt);

n1 = vfprintf(flog, fmt, args);

va\_end(args);

}

va\_start(args, fmt);

n2 = vfprintf(stdout, fmt, args);

va\_end(args);

if (n2 < n1) n1 = n2;

return n1;

}

//---------------------------------------------------------------------------------------------------------------------

// DESCRIPTION: Gets the number of points to check/draw a path based on the length of the path and a resolution value

// ARGUMENTS: len: The arc length of the path

// resolution: a parameter that determines the density of points on a path

// RETURN VALUE: the number of points

size\_t getNumPathPoints(double len, int resolution)

{

size\_t NP; // number of points used to check/draw the path points

if (resolution == RESOLUTION\_LOW)

NP = (size\_t)nint((len / 500.0) \* (double)LOW\_RESOLUTION\_POINTS\_PER\_500\_UNITS);

else if (resolution == RESOLUTION\_MEDIUM)

NP = (size\_t)nint((len / 500.0) \* (double)MEDIUM\_RESOLUTION\_POINTS\_PER\_500\_UNITS);

else

NP = (size\_t)nint((len / 500.0) \* (double)HIGH\_RESOLUTION\_POINTS\_PER\_500\_UNITS);

NP = max(NP, 2); //safety

return NP;

}