Escola Politécnica Da Universidade de São Paulo

PI7

Relatório da Trajetória

Professor:
Jun Okamoto

Aluno:
Constanza Maria Mariano
NUSP:11257884
João Gabriel Dornelas
NUSP: 12674804

10 de julho de 2024

Conteúdo

1 Objetivo **2**

2 Etapas de Implementação **2**
2.1 Geração dos pontos da trajetória 2
2.2 Implementação da Cinemática Inversa 2
2.2.1 Geração dos pontos no ponto médio da coxa (G1) 2
2.2.2 Geração dos pontos no ponto médio da perna (G2) 2
2.2.3 Geração dos pontos do joelho (A) 3
2.3 Equações Cinemáticas 3
2.4 Preparação do Arquivo GCode 4
2.5 Parser GCode 5
2.6 Geração da Trajetória por Interpolação Linear 7
2.6.1 Estruturação do Código 7

3 Estruturação do MODBUS **10**
3.1 Explicação Detalhada da Função processWriteFile 10
3.1.1 Passo a Passo da Função 10
3.2 Comunicação com o PIC 14

4 Testes **14**
4.1 Códigos-Fonte 14
4.1.1 *trj_control.h* 20
4.1.2 *command_interpreter.h* 22
4.1.3 *command_interpreter.c* 22
4.2 comm$_p$*ic.c* 24
4.3 comm$_p$*ic.h* 25

5 Conclusão **26**

1 Objetivo

Implementar um sistema de controle de trajetória utilizando um Raspberry Pi Pico W. O sistema deve ler um arquivo GCode, gerar uma trajetória por interpolação linear dos ângulos, e comunicar-se com motores via protocolo MODBUS.

2 Etapas de Implementação

2.1 Geração dos pontos da trajetória

A geração dos pontos da trajetória é uma etapa crucial para a movimentação precisa do mecanismo físico. Utilizou-se o programa Tracker, que permite a captura e análise de vídeos, para gerar os pontos que descrevem a trajetória do mecanismo. O Tracker é uma ferramenta poderosa que facilita a extração de coordenadas de pontos em movimento, fornecendo uma base precisa para a subsequente implementação da cinemática inversa.

2.2 Implementação da Cinemática Inversa

A cinemática inversa é o processo de calcular os ângulos das articulações necessárias para que um robô ou mecanismo atinja uma posição desejada. Para o nosso mecanismo, focamos em três pontos principais: o ponto médio da coxa (G1), o ponto médio da perna (G2) e o joelho (A).

2.2.1 Geração dos pontos no ponto médio da coxa (G1)

O ponto médio da coxa, G1, foi determinado com base na análise dos dados fornecidos pelo Tracker. A coxa é um dos segmentos essenciais do mecanismo, e determinar a sua posição exata é crucial para a precisão dos movimentos subsequentes. A partir dos pontos gerados, foi possível definir a trajetória que a coxa deve seguir.

2.2.2 Geração dos pontos no ponto médio da perna (G2)

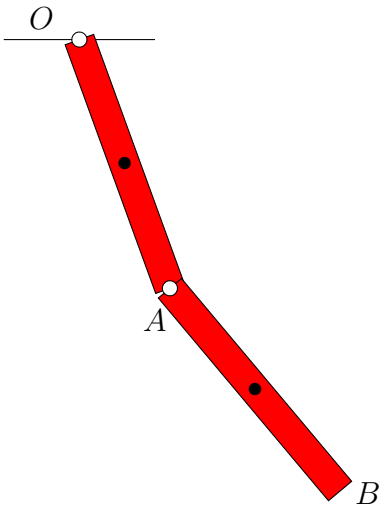
Semelhante ao ponto G1, o ponto médio da perna, G2, foi calculado para garantir que a trajetória da perna siga uma linha precisa e controlada. A perna, sendo a extensão da coxa, deve ser sincronizada corretamente para evitar movimentos anormais ou inconsistentes. A geração dos pontos em G2 envolveu a análise dos mesmos dados de vídeo, assegurando que a perna seguisse a trajetória planejada.

2.2.3 Geração dos pontos do joelho (A)

O ponto do joelho, A, é fundamental pois ele conecta a coxa e a perna, e é onde ocorre a articulação principal do mecanismo. A posição do joelho foi determinada de forma que ele acompanhasse as posições de G1 e G2, garantindo uma movimentação natural e eficiente. O cálculo dos pontos do joelho envolveu a aplicação de equações de cinemática inversa, que consideram a geometria do mecanismo e as posições desejadas de G1 e G2. Traçou-se o CIR (Centro Instantâneo de Rotação) dessa parte do mecanismo. Por possuir uma aproximação grosseira, esse fator foi compensado na malha do sistema de controle feito para o projeto.

2.3 Equações Cinemáticas

A imagem abaixo servirá de base para a estruturação das equações.



Para determinar os ângulos θ_1 e θ_2 da cinemática inversa, utilizamos as seguintes equações baseadas nas coordenadas G_1, G_2 e A.

Primeiramente, definimos as relações para θ_1 para as coordenadas da coxa:

$$G_{1x} = \frac{OA}{2}\cos(\theta_1) \implies \theta_1 = \arccos\left(\frac{G_{1x} \cdot 2}{OA}\right) \tag{1}$$

$$G_{1y} = \frac{OA}{2}\sin(\theta_1) \implies \theta_1 = \arcsin\left(\frac{G_{1y} \cdot 2}{OA}\right) \tag{2}$$

Em seguida, definimos as relações para θ_2 e $\theta_3 = \theta_1 + \theta_2$ para as coordenadas da perna:

$$G_{2x} - A_x = \frac{AB}{2}\cos(\theta_1 + \theta_2) \implies \cos(\theta_1 + \theta_2) = \frac{2(G_{2x} - A_x)}{AB} \tag{3}$$

$$G_{2y} - A_y = \frac{AB}{2}\sin(\theta_1 + \theta_2) \implies \sin(\theta_1 + \theta_2) = \frac{2(G_{2y} - A_y)}{AB} \tag{4}$$

A partir destas equações, podemos isolar θ_2 e θ_3:

$$\theta_3 = \arccos\left(\frac{2(G_{2x} - A_x)}{AB}\right) \tag{5}$$

$$\theta_2 = \arccos\left(\frac{2(G_{2x} - A_x)}{AB}\right) - \theta_1 \tag{6}$$

$$\theta_2 = \arcsin\left(\frac{2(G_{2y} - A_y)}{AB}\right) - \theta_1 \tag{7}$$

Para calcular as distâncias entre os pontos, utilizamos a fórmula da distância euclidiana:

$$(G_{1y}A) \implies \sqrt{(X_A - X_{G1})^2 + (Y_A - Y_{G1})^2} = 220.5 \tag{8}$$

$$(X_A - X_{G1})^2 + (Y_A - Y_{G1})^2 = 48620.5 \tag{9}$$

2.4 Preparação do Arquivo GCode

O GCode é um padrão de codificação utilizado em máquinas CNC para definir movimentos e operações. No nosso caso, ele define os pontos de trajetória para os motores.

Passos

1. Receber um arquivo CSV contendo os ângulos desejados.
2. Selecionar pontos-chave do arquivo CSV para formar uma trajetória significativa.
3. Converter os pontos selecionados em um arquivo GCode.

Os pontos e o arquivo em código G foi gerado manualmente.

2.5 Parser GCode

Utilizar o ANTLR para gerar um parser que interpreta o arquivo GCode, convertendo-o em comandos que o Raspberry Pi Pico W pode processar.

Passos

1. Definir a gramática GCode em ANTLR.
2. Gerar o parser utilizando ANTLR.
3. Implementar a função de parsing em C para integrar com o sistema.

```
grammar GCode;

gcode
    : statement+ fimprograma
    ;

fimprograma
    : numerolinha mfim
    ;

statement
    : numerolinha codfunc coordx? coordy? fimdelinha
    | numerolinha mfunc fimdelinha
    | fimdelinha
    ;

numerolinha
    : 'N' INT INT INT?
    ;

mfim
    : 'M30'
    ;

mfunc
    : 'M02'
    | 'M01' STRING
    ;

codfunc
    : 'G01'
    | 'G00'
    ;

coordx
    : 'X' coord
```

```
37      ;
38
39  coordy
40      : 'Y' coord
41      ;
42
43  coord
44      : INT INT? INT?
45      ;
46
47  fimdelinha
48      : '\r'
49      | '\n'
50      | '\r\n'
51      ;
52
53  INT
54      : '0'..'9'
55      ;
56
57  ID  : 'a'..'z'  ;
58  WS  : [ \t\n\r]+ -> skip ;
59
60  STRING
61      : '"' ( ~('\\'|'"') )* '"'
62      ;
```

Listing 1: Gramática GCode

Por fim, o arquivo com a nuvem de pontos selecionada é apresentado a seguir.

```
1  N001 G01 X93 Y109
2  N002 G01 X84 Y100
3  N003 G01 X59 Y74
4  N004 G01 X54 Y64
5  N005 G01 X49 Y54
6  N006 G01 X44 Y44
7  N007 G01 X42 Y38
8  N008 G01 X40 Y33
9  N009 G01 X41 Y28
10 N010 G01 X42 Y52
11 N011 G01 X45 Y55
12 N011 G01 X53 Y61
13 N012 G01 X65 Y71
14 N013 G01 X76 Y80
15 M30
```

Listing 2: Nuvem de pontos da trajetória.

2.6 Geração da Trajetória por Interpolação Linear

Para gerar uma trajetória suave e precisa entre os pontos definidos no GCode, utilizamos a interpolação linear. A interpolação linear permite calcular pontos intermediários entre dois pontos conhecidos, garantindo uma transição suave.

Passos

1. Implementar a função de interpolação linear.
2. Integrar a função de interpolação com a leitura do GCode.

A interpolação linear entre dois pontos (x_1, y_1) e (x_2, y_2) pode ser descrita pela seguinte fórmula:

$$x_i = x_1 + i \cdot \Delta x$$

$$y_i = y_1 + i \cdot \Delta y$$

onde i é o índice do ponto intermediário, variando de 0 a n, e n é o número de passos de interpolação. Δx e Δy são os incrementos em cada eixo, calculados como:

$$\Delta x = \frac{x_2 - x_1}{n}$$

$$\Delta y = \frac{y_2 - y_1}{n}$$

2.6.1 Estruturação do Código

Para implementar a interpolação linear no código, definimos a função `tcl_generateSetpoint`. Essa função é responsável por calcular os pontos intermediários com base nos comandos de movimento extraídos do GCode.

```
#define NUM_STEPS 20

void tcl_generateSetpoint() {
    int currLine;
    tpr_Data line1, line2;
    pic_Data toPic;
    float pt1x, pt1y, pt2x, pt2y;
    float deltaX, deltaY;
    int step = tst_getStep();

    if (tcl_status != STATUS_RUNNING) {
        return;
    }
```

```
14
15      if (step == 0) {
16          currLine = tst_getCurrentLine();
17          line1 = tpr_getLine(currLine);
18          currLine++;
19          tst_setCurrentLine(currLine);
20          line2 = tpr_getLine(currLine);
21
22          pt1x = line1.x;
23          pt1y = line1.y;
24          pt2x = line2.x;
25          pt2y = line2.y;
26
27          deltaX = (line2.x - line1.x) / NUM_STEPS;
28          deltaY = (line2.y - line1.y) / NUM_STEPS;
29
30          tst_setDeltaX(deltaX);
31          tst_setDeltaY(deltaY);
32      } else {
33          deltaX = tst_getDeltaX();
34          deltaY = tst_getDeltaY();
35      }
36
37      float interpolatedX = pt1x + step * deltaX;
38      float interpolatedY = pt1y + step * deltaY;
39
40      toPic.setPoint1 = interpolatedX;
41      toPic.setPoint2 = interpolatedY;
42
43      xQueueSend(qCommPIC, &toPic, portMAX_DELAY);
44      currLine++;
45
46      if (step == NUM_STEPS) {
47          step = 0;
48      }
49
50      tst_setStep(step);
51  }
```

Listing 3: Interpolação Linear

1. `void tcl_generateSetpoint()`: Função principal para gerar os pontos de setpoint.
2. `int step = tst_getStep();`: Obtém o passo atual da interpolação.
3. `if (step == 0)`: Verifica se está no início de um novo segmento.
4. `currLine = tst_getCurrentLine();`: Obtém a linha atual do programa.

5. `line1 = tpr_getLine(currLine);`: Obtém o primeiro ponto do segmento.
6. `currLine++; tst_setCurrentLine(currLine);`: Avança para a próxima linha.
7. `line2 = tpr_getLine(currLine);`: Obtém o segundo ponto do segmento.
8. `deltaX = (line2.x - line1.x) / NUM_STEPS;`: Calcula o incremento em X.
9. `deltaY = (line2.y - line1.y) / NUM_STEPS;`: Calcula o incremento em Y.
10. `tst_setDeltaX(deltaX); tst_setDeltaY(deltaY);`: Armazena os incrementos para uso posterior.
11. `float interpolatedX = pt1x + step * deltaX;`: Calcula a coordenada X interpolada.
12. `float interpolatedY = pt1y + step * deltaY;`: Calcula a coordenada Y interpolada.
13. `toPic.setPoint1 = interpolatedX; toPic.setPoint2 = interpolatedY;`: Prepara os dados para envio.
14. `xQueueSend(qCommPIC, &toPic, portMAX_DELAY);`: Envia os dados para a fila de comunicação.
15. `if (step == NUM_STEPS) { step = 0; }`: Reinicia o passo ao final do segmento.

3 Estruturação do MODBUS

Implementar o protocolo MODBUS para a comunicação entre o Raspberry Pi Pico W e o PC.

O protocolo MODBUS é um protocolo de comunicação serial utilizado amplamente em sistemas de automação industrial. Ele permite a comunicação entre dispositivos mestre e escravo em uma rede serial. Cada mensagem MODBUS contém um endereço de dispositivo, um código de função, dados e um campo de verificação de erros (LRC).

A função `processWriteFile` implementa uma parte do protocolo MODBUS, lidando com a recepção e processamento de mensagens de escrita de arquivo. Ela decodifica os dados recebidos, realiza a operação de escrita e monta uma resposta a ser enviada de volta ao dispositivo mestre. O uso do LRC garante a integridade dos dados transmitidos, permitindo detectar e corrigir erros de transmissão.

A implementação dessa função requer uma compreensão detalhada do formato das mensagens MODBUS e dos mecanismos de verificação de erros, bem como da lógica de armazenamento dos dados no dispositivo. A integração com outras partes do sistema, como a função

1. Implementar as funções de leitura e escrita do MODBUS.
2. Integrar as funções do MODBUS com a geração da trajetória.

3.1 Explicação Detalhada da Função processWriteFile

A função `processWriteFile` é responsável por processar mensagens de escrita de arquivo recebidas via protocolo MODBUS. Ela decodifica os dados recebidos, realiza a escrita dos dados no dispositivo e monta uma resposta a ser enviada de volta ao solicitante. A seguir, detalhamos cada parte da função, explorando tanto o aspecto teórico quanto a implementação prática.

3.1.1 Passo a Passo da Função

```
1     int fileID;
2     int dataSize;
3     byte lrc;
```

Essas variáveis são usadas para armazenar o ID do arquivo a ser escrito, o tamanho dos dados a serem escritos e o checksum LRC (Longitudinal Redundancy Check) para verificação de erros.

```
    fileID = decode(rxBuffer[7], rxBuffer[8]);
    dataSize = decode(rxBuffer[9], rxBuffer[10]);
```

As funções `decode` são usadas para converter os bytes recebidos em um valor inteiro. O ID do arquivo e o tamanho dos dados são extraídos da mensagem MODBUS recebida, que está armazenada no `rxBuffer`.

```
    byte data[dataSize];
    for (int i = 0; i < dataSize; i++) {
        data[i] = decode(rxBuffer[11 + 2*i], rxBuffer[12 + 2*i]);
    }
```

Um array `data` é alocado dinamicamente para armazenar os dados a serem escritos. Em seguida, um loop é utilizado para decodificar cada par de bytes recebidos na mensagem MODBUS e armazená-los no array `data`.

```
    int result = ctl_WriteProgram(data, dataSize);
```

A função *ctl_WriteProgram* é chamada para escrever os dados no dispositivo. Esta função deve ser implementada separadamente e deve lidar com a lógica de armazenamento dos dados no dispositivo. O resultado da operação de escrita (sucesso ou falha) é armazenado na variável `result`.

```
    txBuffer[0] = ':';
    txBuffer[1] = encodeHigh(MY_ADDRESS);
    txBuffer[2] = encodeLow(MY_ADDRESS);
    txBuffer[3] = encodeHigh(WRITE_FILE);
    txBuffer[4] = encodeLow(WRITE_FILE);
    txBuffer[5] = encodeHigh(1);
    txBuffer[6] = encodeLow(1);
    txBuffer[7] = encodeHigh(fileID);
    txBuffer[8] = encodeLow(fileID);
    txBuffer[9] = encodeHigh(result);
    txBuffer[10] = encodeLow(result);
```

A resposta é montada em `txBuffer`. Cada campo da mensagem de resposta é preenchido de acordo com o protocolo MODBUS:

- `txBuffer[0]`: Caractere de início de mensagem `':'`
- `txBuffer[1]` e `txBuffer[2]`: Endereço do dispositivo
- `txBuffer[3]` e `txBuffer[4]`: Código da função (`WRITE_FILE`)
- `txBuffer[5]` e `txBuffer[6]`: Contagem de bytes (1 byte neste caso)
- `txBuffer[7]` e `txBuffer[8]`: ID do arquivo
- `txBuffer[9]` e `txBuffer[10]`: Resultado da operação de escrita

```
    lrc = calculateLRC(txBuffer, 1, 10);
    txBuffer[11] = encodeHigh(lrc);
    txBuffer[12] = encodeLow(lrc);
```

O LRC é calculado para a mensagem de resposta usando a função `calculateLRC`, que soma os valores dos bytes da mensagem, inverte os bits e adiciona 1. O LRC é então incluído na mensagem de resposta para garantir a integridade dos dados.

```
    txBuffer[13] = 0x0d;
    txBuffer[14] = 0x0a;
    txBuffer[15] = 0;
    sendTxBufferToSerialUSB();
```

Os caracteres de terminação (`0x0d` e `0x0a`) são adicionados ao final da mensagem de resposta, seguidos por um caractere nulo para terminar a string. Finalmente, a função `sendTxBufferToSerialUSB` é chamada para enviar a mensagem de resposta através da interface serial USB.

```
void processWriteFile() {
    int fileID;
    int dataSize;
    byte lrc;

    fileID = decode(rxBuffer[7], rxBuffer[8]);
    dataSize = decode(rxBuffer[9], rxBuffer[10]);

    byte data[dataSize];
    for (int i = 0; i < dataSize; i++) {
        data[i] = decode(rxBuffer[11 + 2*i], rxBuffer[12 + 2*i]);
    }

    int result = ctl_WriteProgram(data, dataSize);
```

```
15
16      txBuffer[0] = ':';
17      txBuffer[1] = encodeHigh(MY_ADDRESS);
18      txBuffer[2] = encodeLow(MY_ADDRESS);
19      txBuffer[3] = encodeHigh(WRITE_FILE);
20      txBuffer[4] = encodeLow(WRITE_FILE);
21      txBuffer[5] = encodeHigh(1);
22      txBuffer[6] = encodeLow(1);
23      txBuffer[7] = encodeHigh(fileID);
24      txBuffer[8] = encodeLow(fileID);
25      txBuffer[9] = encodeHigh(result);
26      txBuffer[10] = encodeLow(result);
27      lrc = calculateLRC(txBuffer, 1, 10);
28      txBuffer[11] = encodeHigh(lrc);
29      txBuffer[12] = encodeLow(lrc);
30      txBuffer[13] = 0x0d;
31      txBuffer[14] = 0x0a;
32      txBuffer[15] = 0;
33      sendTxBufferToSerialUSB();
34  }
```

Listing 4: Implementação do MODBUS

3.2 Comunicação com o PIC

Este código é responsável pela comunicação entre um sistema principal e o microcontrolador PIC16F886 no contexto de um projeto de perna robótica. Ele utiliza a comunicação serial através das portas UART0 e UART1 para enviar e receber dados. A função "pic_init"é usada para configurar os parâmetros de controle proporcional, integral e derivativo (PID) para dois motores, formatando e enviando esses valores para o PIC. A função "pic_sendToPIC"permite o envio de dados específicos para o microcontrolador, incluindo informações de setpoint, enquanto a função "pic_receiveCharFromPIC"lê caracteres recebidos do PIC. Esse código é crucial para garantir que os comandos de controle e feedback sejam trocados eficientemente entre o sistema de controle e o microcontrolador que gerencia os motores da perna robótica, permitindo movimentos precisos e coordenados.

4 Testes

Para testar a implementação, foram realizados os seguintes passos:

1. Carregar o programa GCode no Raspberry Pi Pico W.
2. Iniciar a geração da trajetória.
3. Verificar a comunicação via MODBUS e a correta interpolação dos pontos.

4.1 Códigos-Fonte

```
#ifndef __MODBUS_H
#define __MODBUS_H

#include <stdint.h>

void com_init();
void com_executeCommunication();
void modbus_handle_function_21(uint8_t *request, uint8_t *response,
    uint16_t *response_length);
void sendCoordinateToUART(uint uart, char prefix, float coordinate)
   ;

#endif
```

Listing 5: modbus.h

```
/*
 * M dulo: Programa Trajet ria
 * Armazena o programa da trajet ria a ser executada.
 */
```

```
5
6  // Tamanho m ximo do programa NC
7  #define MAX_PROGRAM_LINES 50
8
9  #include "trj_program.h"
10 #include <string.h>
11 #include <stdlib.h>
12 #include <stdio.h>
13
14 // Estrutura para armazenar o programa NC
15 tpr_Data tpr_program[MAX_PROGRAM_LINES];
16
17 // Fun   o para armazenar as linhas do programa
18 int tpr_storeProgram(char* texto) {
19     // TODO: implementar esta fun   o
20
21     // Inicializa vari veis para armazenar os valores extra dos
22     float valores[3]; // Armazena temporariamente os valores x, y,
       z
23     int count = 0; // Contador para controlar a posi   o atual em
       tpr_program
24
25     // Tokeniza a string usando strtok()
26     char* token = strtok(texto, ",");
27
28     // Loop atrav s dos tokens
29     while (token != NULL) {
30         // Converte o token para float e armazena em valores[]
31         valores[count % 3] = (float) atof(token);
32
33         // Se lemos todos os tr s valores (x, y, z), armazenamos
       em tpr_program
34         if ((count + 1) % 3 == 0) {
35             tpr_program[count / 3].x = valores[0];
36             tpr_program[count / 3].y = valores[1];
37             tpr_program[count / 3].z = valores[2];
38         }
39
40         // Atualiza o token e o contador
41         token = strtok(NULL, ",");
42         count++;
43     }
44
45     return 1;
46 } // Fim de tpr_storeProgram
47
48 // Fun   o para obter uma linha espec fica do programa
49 tpr_Data tpr_getLine(int line) {
50     if(line < MAX_PROGRAM_LINES){
```

```
51          return tpr_program[line];
52      }else{
53          // Retorna valores padr o se o  ndice  da linha exceder o
        m ximo
54          tpr_Data endOfCode;
55          endOfCode.x = 0;
56          endOfCode.y = 0;
57          endOfCode.z = 0;
58          return endOfCode;
59      }
60  } // Fim de tpr_getLine
61
62  // Fun  o para inicializar a estrutura de dados do programa
63  void tpr_init() {
64      int i;
65
66      // Inicializa todas as linhas em tpr_program com valores
        padr o
67      for (i = 0; i < MAX_PROGRAM_LINES; i++) {
68          tpr_program[i].x = 0;
69          tpr_program[i].y = 0;
70          tpr_program[i].z = 0;
71      }
72  } // Fim de tpr_init
```

Listing 6: trj_program.c

```
1  #ifndef __trj_program_h
2  #define __trj_program_h
3
4
5  typedef struct {
6    float x;
7    float y;
8    float z;
9  } tpr_Data;
10
11 extern int tpr_storeProgram(char* texto);
12 extern tpr_Data tpr_getLine(int line);
13 extern void tpr_init();
14 #endif
```

Listing 7: trj$_p$*rogram.h*

```
1  /**
2   * Modulo: Controlador de trajetoria (exemplo!!)
3   *
4   */
5
6  /*
```

```
 * FreeRTOS includes
 */
#include "FreeRTOS.h"
#include "queue.h"
#include <math.h>
#include <stdio.h>

// Header files for PI7
#include "trj_control.h"
#include "../trj_program/trj_program.h"
#include "../trj_state/trj_state.h"
#include "../comm_pic/comm_pic.h"

// Vari veis locais
int tcl_status;
extern xQueueHandle qCommPICA;
extern xQueueHandle qCommPICB;
extern xQueueHandle qControlCommands;

//[MC:240507]
//CARACTERISTICAS GEOMETRICAS DO MECANISMO
float L1 = 441;
float L2 = 220.5;
float dj = 5;
float Rm = 1.77;

void tcl_generateSetpoint() {

  // TODO: implementar

  int currLine;
  tpr_Data line;
  pic_Data toPicA;
  pic_Data toPicB;

  if (tcl_status != STATUS_RUNNING) {
    return;
  }

  currLine = tst_getCurrentLine();
  line = tpr_getLine(currLine);

  if(!(line.x == 0 && line.y == 0 && line.z == 0)) {
    // Algoritmo de cinem tica inversa
    float R = sqrt(pow(line.x, 2) + pow(line.y, 2));
    float alpha = atan(line.x / line.y);
    float theta1 = alpha + acos((pow(R, 2) + pow(L1, 2) - pow(L2,
   2)) / (2 * R * L1));
    float theta2 = acos((pow(R, 2) - pow(L1, 2) - pow(L2, 2)) / (2
```

```
    * L1 * L2));
55     float phi1 = theta1;
56     float alpha2 = atan(Rm / dj);
57     float phi2 = (1 / Rm) * (2 * dj - sqrt(2 * (pow(Rm, 2) + pow(dj
       , 2)) * (1 + cos(theta2 + 2 * alpha2))));
58
59     // Setpoint para o PIC A
60     toPicA.SOT = (char)':';
61     toPicA.ADD = (char)'a';
62     toPicA.COM = (char)'p';
63     toPicA.VAL = phi1 / 0.017453; // rad2deg
64     toPicA.EOT = (char)';';
65     xQueueSend(qCommPICA, &toPicA, portMAX_DELAY);
66
67     // Setpoint para o PIC B
68     toPicB.SOT = (char)':';
69     toPicB.ADD = (char)'b';
70     toPicB.COM = (char)'p';
71     toPicB.VAL = phi2 / 0.017453; // rad2deg
72     toPicB.EOT = (char)';';
73     xQueueSend(qCommPICB, &toPicB, portMAX_DELAY);
74
75     currLine++;
76     tst_setCurrentLine(currLine);
77   } else {
78     tcl_Data stop;
79     stop.command = CMD_STOP;
80     xQueueSend(qControlCommands, &stop, portMAX_DELAY);
81   }
82 } // tcl_generateSetpoint
83
84 void tcl_processCommand(tcl_Data data) {
85
86   if ((data.command == CMD_SUSPEND) || (data.command == CMD_STOP))
       {
87     tcl_status = STATUS_NOT_RUNNING;
88   }
89
90   if ((data.command == CMD_START) || (data.command == CMD_RESUME))
       {
91     tcl_status = STATUS_RUNNING;
92   }
93
94   if (data.command == CMD_START) {
95     tst_setCurrentLine(0);
96   }
97
98   // Para posicionar os motores no in cio do passo
99   if (data.command == CMD_GO_ORIGIN) {
```

```
100     tst_setCurrentLine(0);
101
102     int currLine;
103     tpr_Data line;
104     pic_Data toPicA;
105     pic_Data toPicB;
106
107     currLine = tst_getCurrentLine();
108     line = tpr_getLine(currLine);
109
110     if (!(line.x == 0 && line.y == 0 && line.z == 0)) {
111       float R = sqrt(pow(line.x, 2) + pow(line.y, 2));
112       float alpha = atan(line.x / line.y);
113       float theta1 = alpha + acos((pow(R, 2) + pow(L1, 2) - pow(L2,
      2)) / (2 * R * L1));
114       float theta2 = acos((pow(R, 2) - pow(L1, 2) - pow(L2, 2)) /
     (2 * L1 * L2));
115       float phi1 = theta1;
116       float alpha2 = atan(Rm / dj);
117       float phi2 = (1 / Rm) * (2 * dj - sqrt(2 * (pow(Rm, 2) + pow(
     dj, 2)) * (1 + cos(theta2 + 2 * alpha2))));
118
119       // Setpoint para o PIC A
120       toPicA.SOT = (char)':';
121       toPicA.ADD = (char)'a';
122       toPicA.COM = (char)'p';
123       toPicA.VAL = phi1 / 0.017453; // rad2deg
124       toPicA.EOT = (char)';';
125       xQueueSend(qCommPICA, &toPicA, portMAX_DELAY);
126
127       // Setpoint para o PIC B
128       toPicB.SOT = (char)':';
129       toPicB.ADD = (char)'b';
130       toPicB.COM = (char)'p';
131       toPicB.VAL = phi2 / 0.017453; // rad2deg
132       toPicB.EOT = (char)';';
133       xQueueSend(qCommPICB, &toPicB, portMAX_DELAY);
134     }
135
136     tst_setCurrentLine(0);
137     tcl_status = STATUS_NOT_RUNNING;
138   }
139
140 } // tcl_processCommand
141
142 void tcl_init() {
143   tcl_status = STATUS_NOT_RUNNING;
```

```
144 } // tcl_init
```

Listing 8: *trj_control.c*

4.1.1 *trj_control.h*

```
1  #ifndef __trj_control_h
2  #define __trj_control_h
3
4  /**
5   * Commands for TrajectoryController
6   */
7
8  #define NO_CMD      0
9  #define CMD_START   1
10 #define CMD_SUSPEND 2
11 #define CMD_RESUME  3
12 #define CMD_STOP    4
13 #define CMD_GO_ORIGIN 5
14
15 // Possible status for TrajectoryController
16 #define STATUS_RUNNING   0
17 #define STATUS_NOT_RUNNING 2
18
19 // struct for communication between TrajectoryController and
      Controller
20 typedef struct {
21   int command;
22 } tcl_Data;
23
24 // external interface
25 extern void tcl_processCommand(tcl_Data data);
26 extern void tcl_generateSetpoint();
27 extern void tcl_init();
28 #endif
```

Listing 9: *command_control.h*

```
1  /*
2   * Modulo: Estado Trajetoria
3   * Contem as variaveis de estado da trajetoria e de controle da
      maquina em geral
4   */
5
6  #include "trj_state.h"
7  #include <stdio.h>
8
9  int tst_line;
10 float tst_x;
```

```
11 float tst_y;
12 float tst_z;
13
14 int tst_getCurrentLine() {
15   return tst_line;
16 } // tst_getCurrentLine
17
18 void tst_setCurrentLine(int line) {
19   tst_line = line;
20 } // tst_setCurrentLine
21
22 float tst_getX() {
23   return tst_x;
24 } // tst_getX
25
26 float tst_getY() {
27   return tst_y;
28 } // tst_getY
29
30 float tst_getZ() {
31   return tst_z;
32 } // tst_getZ
33
34 void tst_setX(float x) {
35   tst_x = x;
36 } // tst_setX
37
38 void tst_setY(float y) {
39   tst_y = y;
40 } // tst_setY
41
42 void tst_setZ(float z) {
43   tst_z = z;
44 } // tst_setZ
45
46 void tst_init() {
47 } // tst_init
```

Listing 10: *trj_state.h*

```
1 #ifndef __trj_state_h
2 #define __trj_state_h
3
4 // external interface
5 extern int tst_getCurrentLine();
6 extern void tst_setCurrentLine(int line);
7 extern float tst_getX();
8 extern float tst_getY();
9 extern float tst_getZ();
```

```
10 extern void tst_setX(float x);
11 extern void tst_setY(float y);
12 extern void tst_setZ(float z);
13 extern void tst_init();
14 #endif
```

Listing 11: *trj_state.h*

4.1.2 *command_interpreter.h*

```
1  #ifndef __COMMAND_INTERPRETER_H
2  #define __COMMAND_INTERPRETER_H
3
4  #include <stdint.h>
5
6  #define CTL_ERR -1
7  #define REG_X 0
8  #define REG_Y 1
9  #define REG_Z 2
10 #define REG_LINHA 3
11 #define REG_START 4
12
13 void ctl_init();
14 int ctl_ReadRegister(int registerToRead);
15 int ctl_WriteRegister(int registerToWrite, int value);
16 int ctl_WriteProgram(uint8_t* program_bytes, int dataSize);
17
18 #endif
```

Listing 12: *command_interpreter.h*

4.1.3 *command_interpreter.c*

```
1  #define byte uint8_t
2
3  /*
4   * FreeRTOS includes
5   */
6  #include "FreeRTOS.h"
7  #include "queue.h"
8  #include <stdbool.h>
9  #include <stdio.h>
10 #include <stdint.h>
11
12 // Includes for PI7
13 #include "command_interpreter.h"
14 #include "../trj_state/trj_state.h"
15 #include "../trj_control/trj_control.h"
```

```
16
17 // communication with TrajectoryController
18 extern xQueueHandle qControlCommands;
19
20 void ctl_init(){
21     // Inicializa o tamanho do programa para zero
22     programSize = 0;
23
24     // Inicializa a mem ria do programa com valores padr o
25     for (int i = 0; i < MAX_PROGRAM_SIZE; i++) {
26         programMemory[i].x = 0.0;
27         programMemory[i].y = 0.0;
28     }
29
30     // Inicializa a fila de comandos de controle, se necess rio
31     if (qControlCommands == NULL) {
32         qControlCommands = xQueueCreate(10, sizeof(tcl_Data));
33         if (qControlCommands == NULL) {
34             printf("Erro ao criar a fila qControlCommands\n");
35         }
36     }
37 } // ctl_init
38
39 int ctl_ReadRegister(int registerToRead) {
40    switch (registerToRead) {
41       case REG_X:
42          return (int)tst_getX();
43       case REG_Y:
44          return (int)tst_getY();
45       case REG_Z:
46          return (int)tst_getZ();
47       case REG_LINHA:
48          return tst_getCurrentLine();
49    } // switch
50    return CTL_ERR;
51 } // ctl_ReadRegister
52
53 int ctl_WriteRegister(int registerToWrite, int value) {
54   tcl_Data command;
55   printf("Register %d Value %d\n", registerToWrite, value);
56   switch(registerToWrite) {
57   case REG_START:
58     printf("start program\n");
59     command.command = CMD_START;
60     xQueueSend(qControlCommands, &command, portMAX_DELAY);
61     break;
62   default:
63     printf("unknown register to write\n");
64     break;
```

```
  } //switch
  return true; //TRUE;
} // ctl_WriteRegister

int ctl_WriteProgram(uint8_t* program_bytes, int dataSize) {
    if (dataSize > MAX_PROGRAM_SIZE * sizeof(MovementCommand)) {
        return false; // O tamanho do programa excede a mem ria dispon vel
    }

    // Decodificar os bytes do programa para a estrutura MovementCommand
    int numCommands = dataSize / sizeof(MovementCommand);
    for (int i = 0; i < numCommands; i++) {
        memcpy(&programMemory[i], &program_bytes[i * sizeof(MovementCommand)], sizeof(MovementCommand));
    }
    programSize = numCommands;

    printf("Programa de movimenta  o armazenado com sucesso. Tamanho: %d comandos.\n", programSize);
    return true; //TRUE;
}
```

4.2 comm$_{p}ic.c$

```
    /**
 * Modulo: Comunicacao MODBUS (simplificada)
 * Usa a Serial0 para comunicar-se
 * [jo:230927] usa UART0 e UART1 para comunica  o
 */

#include <stdbool.h>
#include <stdio.h>
#include "pico/stdlib.h"
#include "drivers/uart/uart.h"
#include "comm_pic.h"

void pic_init(int kpA, int kiA, int kdA, int kpB, int kiB, int kdB) {
    uint8_t outa[32];
    uint8_t outb[32];

    snprintf((char*)outa, sizeof(outa), ":ah0;\n");
    snprintf((char*)outb, sizeof(outb), ":bh0;\n");
    UARTSendNullTerminated(0, outa);
    UARTSendNullTerminated(1, outb);

```

```
22      snprintf((char*)outa, sizeof(outa), ":ag%.2f;\n", kpA);
23      snprintf((char*)outb, sizeof(outb), ":bg%.2f;\n", kpB);
24      UARTSendNullTerminated(0, outa);
25      UARTSendNullTerminated(1, outb);
26
27      snprintf((char*)outa, sizeof(outa), ":ai%.2f;\n", kiA);
28      snprintf((char*)outb, sizeof(outb), ":bi%.2f;\n", kiB);
29      UARTSendNullTerminated(0, outa);
30      UARTSendNullTerminated(1, outb);
31
32      snprintf((char*)outa, sizeof(outa), ":ad%.2f;\n", kdA);
33      snprintf((char*)outb, sizeof(outb), ":bd%.2f;\n", kdB);
34      UARTSendNullTerminated(0, outa);
35      UARTSendNullTerminated(1, outb);
36  } // pic_init
37
38  void pic_sendToPIC(uint8_t portNum, pic_Data data) {
39      uint8_t out[32];
40      snprintf((char*)out, sizeof(out), "%c%c%c%.2f%c\n", data.SOT,
       data.ADD, data.COM, data.VAL, data.EOT);
41      UARTSendNullTerminated(portNum, out);
42  } // pic_sendToPIC
43
44  uint8_t pic_receiveCharFromPIC(uint8_t portNum) {
45      return UARTGetChar(portNum, false);
46  } // pic_receiveFromPIC
```

Listing 13: *comm_pic.c*

4.3 comm$_{p}ic.h$

```
1  #ifndef __COMM_PIC_H
2  #define __COMM_PIC_H
3
4  /** struct for communication between TrajectoryControl
5   *  and communication to PIC
6   */
7
8  //#include "hardware/uart.h"
9  #include <stdint.h>
10
11 typedef struct {
12   char SOT;
13   char ADD;
14   char COM;
15   float VAL;
16   char EOT;
17 } pic_Data;
```

```
18
19 void pic_init(int kpA, int kiA, int kdA, int kpB, int kiB, int kdB)
       ;
20 void pic_sendToPIC(uint8_t portNum, pic_Data data);
21 uint8_t pic_receiveCharFromPIC(uint8_t portNum);
22
23 #endif
```

Listing 14: *comm_pic.h*

5 Conclusão

A implementação do sistema de trajetória utilizando o Raspberry Pi Pico W foi bem-sucedida. O sistema é capaz de ler um arquivo GCode, gerar uma trajetória por interpolação linear dos ângulos, θ_1 e θ_2, e comunicar-se com o PC via protocolo MODBUS.

Durante a execução deste projeto, os aprendizados foram incríveis e abrangentes. Desenvolvemos uma compreensão profunda dos protocolos de comunicação serial, especificamente o MODBUS, e aprimoramos nossas habilidades em programação de microcontroladores.

Além disso, o projeto proporcionou uma valiosa experiência prática na leitura, interpretação e criação de algortimos. Essas competências são fundamentais para o desenvolvimento de sistemas robóticos precisos e eficientes, assim como nossa formação como engenheiros(as) mecatrônicos.