目录

[第一章 设计思路 2](#_Toc164891512)

[1.1 任务要求 2](#_Toc164891513)

[1.2 设计思路及项目组成 2](#_Toc164891514)

[第二章 仿真流程介绍 3](#_Toc164891515)

[2.1 仿真总流程 3](#_Toc164891516)

[2.2 NC程序解析函数parse\_nc() 4](#_Toc164891517)

[2.3 定时器函数timer\_Tick() 12](#_Toc164891518)

[第三章 其余类的介绍 21](#_Toc164891519)

[3.1 axis\_item类 21](#_Toc164891520)

[3.2 milling\_item类 22](#_Toc164891521)

[3.3 status\_table类 23](#_Toc164891522)

[附录 25](#_Toc164891523)

# 第一章 设计思路

## 1.1 任务要求

对于给定的一段NC代码，采用VC、VS或QT编写程序解释、插补软件，在PC机上仿真数控装置，进行图形描绘、坐标值显示、步进电机控制模拟显示及信号输出、冷却液和主轴开关量控制模拟显示及信号输出。

## 1.2 设计思路及项目组成

由任务要求，抽象出GUI的组成为：图形描绘、状态栏、NC程序输入栏。将图形描绘分为两部分，一部分是路径图像显示，另一部分是图像显示背后的解析NC程序的逻辑，如图1.1所示。

项目组成

图形描绘

状态栏

NC程序输入栏

路径图像显示

解析NC程序

坐标轴类：axis\_item

渲染路径类：

milling\_item

状态栏类：

status\_table

解析类：mainwindow

图1.1 项目组成

# 第二章 仿真流程介绍

## 2.1 仿真总流程

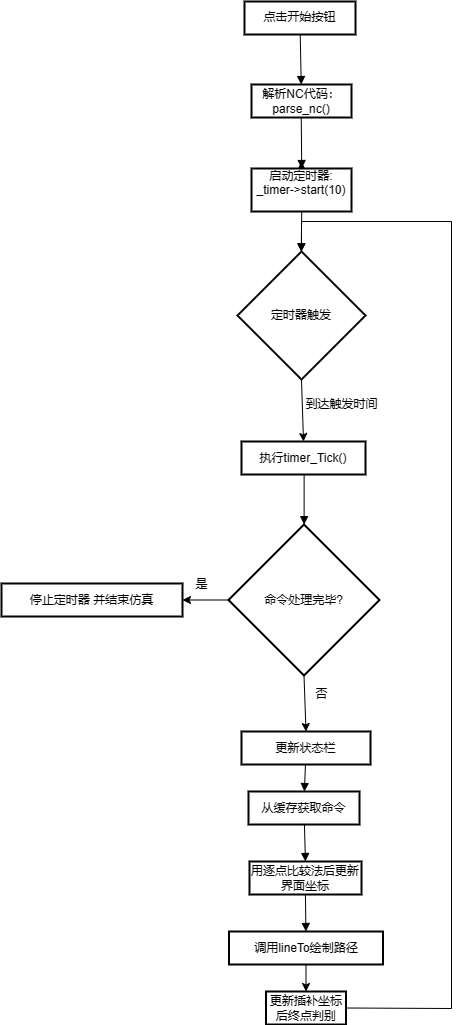
 仿真总流程如图2.1所示。

图2.1 仿真总流程

（1）在文本框中输入NC程序，点击开始

（2）调用parse\_nc()函数，解析NC程序

（3）每隔10ms启用定时器，一到时间调用timer\_Tick()函数

（4）判断是否处理完所有命令的仿真。若已处理完，停止计时器，结束仿真。若未处理完，先更新状态栏，之后从缓存获取命令，用逐点比较法前进一个单元

（5）调用lineto()函数绘制路径

（6）更新插补坐标后终点判别

以下介绍更详细的实现过程。

## 2.2 NC程序解析函数parse\_nc()

此函数涉及到mainwindow类中的一些私有变量，变量名及其作用如下：

1. *// NC代码解析相关数组，预定义长度为DEFAULT\_PT\_LEN，即100。*
2. float fXbeginArray[DEFAULT\_PT\_LEN] {}; *// 存储每条NC代码的起始X坐标。*
3. float fYbeginArray[DEFAULT\_PT\_LEN] {}; *// 存储每条NC代码的起始Y坐标。*
4. float fXendArray[DEFAULT\_PT\_LEN] {}; *// 存储每条NC代码的结束X坐标。*
5. float fYendArray[DEFAULT\_PT\_LEN] {}; *// 存储每条NC代码的结束Y坐标。*
6. float fFArray[DEFAULT\_PT\_LEN] {}; *// 存储每条NC代码的进给速率(F代码)。*
7. float fIArray[DEFAULT\_PT\_LEN] {}; *// 圆弧插补中I参数，表示圆弧中心相对于起始点的X偏移。*
8. float fJArray[DEFAULT\_PT\_LEN] {}; *// 圆弧插补中J参数，表示圆弧中心相对于起始点的Y偏移。*
9. double dRArray[DEFAULT\_PT\_LEN] {}; *// 存储计算出的圆弧半径值。*
10. int nDirArray[DEFAULT\_PT\_LEN] {}; *// 存储每条NC代码的旋转方向(M03顺时针，M04逆时针)。*
11. int nSArray[DEFAULT\_PT\_LEN] {}; *// 存储每条NC代码的主轴转速(S代码)。*
12. int nTArray[DEFAULT\_PT\_LEN] {}; *// 存储每条NC代码的工具编号(T代码)。*
13. int nCool1Array[DEFAULT\_PT\_LEN] {}; *// 存储每条NC代码冷却液一的状态(M08开，M09关)。*
14. int nCool2Array[DEFAULT\_PT\_LEN] {}; *// 存储每条NC代码冷却液二的状态(M07开，M09关)。*
15. int nStatusArray[DEFAULT\_PT\_LEN] {}; *// 存储每条NC代码的状态，例如G00快速定位，G01直线插补。*
16. int nLineNum = 0; *// 当前解析的NC程序行号或画第nLineNum行的图。*
17. int finish = 0; *// 解析完成标志，当所有NC代码行解析完毕后设为1。*
18. int G90[DEFAULT\_PT\_LEN] {}; *// 存储每条NC代码是否采用绝对编程(G90)或相对编程(G91)，默认为绝对编程。*
19. std::vector<std::string> \_cmd\_cache; *// 用于存储从文本编辑器中读取的原始NC代码行，过滤后用于解析。*

此函数的伪代码如下：

1. /**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\****\*\*\**\*
2. 函数名称：void parse\_nc()
3. 函数功能：解析NC代码
4. 输入参数：无
5. 输出参数：无
6. 返 回 值：无
7. 备    注：此函数用于从GUI中的文本编辑器读取并解析NC代码，
8. 并更新各种状态。
9. **\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***/
10. void parse\_nc()
11. {
12. // 清空命令缓存
13. 清空 \_cmd\_cache;
14. // 从UI的文本编辑器中读取所有文本并按行分割
15. 从UI中获取文本并分行;
16. // 对每一行进行遍历
17. 对于每一行:
18. 如果 行不为空且不包含注释("//")
19. 将行添加到 \_cmd\_cache;
20. // 初始化行号
21. 初始化 nLineNum 为 0;
22. // 处理每一行的NC代码
23. 当 nLineNum 小于 \_cmd\_cache 的大小时{
24. 读取当前行到 lineContent;
25. 去除 lineContent 中的所有空格;
26. // 初始化识别标志变量
27. 初始化各状态标志如 nX\_exist, nY\_exist 等;
28. // 解析行中的每个字符
29. 对于 lineContent 中的每个字符:
30. 如果 字符是数字、小数点或负号
31. 将字符追加到 str\_code;
32. 否则
33. 将 str\_code 存入 codeArray 并开始新的 str\_code;
34. 更新字符索引;
35. // 重置指令索引
36. 重置指令索引 nCodeNum 为 1;
37. // 解析并处理每个代码
38. 对于 codeArray 中的每个代码:
39. 提取第一个字符和后续文本;
40. 根据指令字符进行处理;
41. // 更新起始坐标
42. 更新起始坐标数组为当前行的结束坐标;
43. // 更新行号
44. nLineNum 自增;
45. }
46. // 设置完成标志
47. 设置 finish 为 1;
48. }

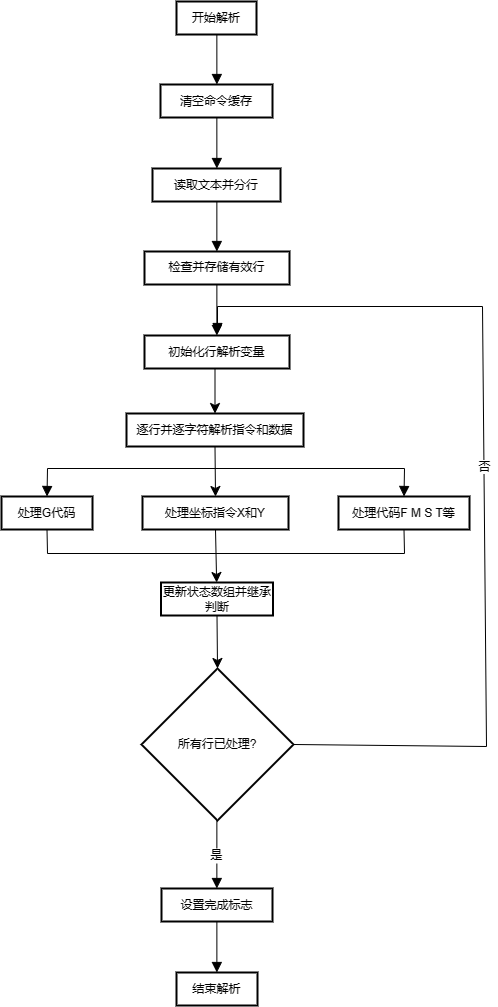
parse\_nc()函数的流程图如图2.2所示

图2.2 parse\_nc()函数的流程图

**（1）**首先调用vector库中的clear()函数清空**\_**cmd\_cache，，此步骤的目的是为了防止数据混淆，确保之前解析的数据不会与当前会话的数据混淆。

**（2）**通过 ui->edit\_cmd->toPlainText().split("\n") 方法，将NC程序输入栏中的内容按行分割，并过滤掉包含注释（"//"）或空行的内容，仅保留有效的NC代码行。通过获取\_cmd\_cache此容器的大小，得到总行数。

**（3）**逐行解析NC程序

**<3.1>**开始解析前，创建一些变量，其作用如下所示

1. int nX\_exist = 0, nY\_exist = 0;
2. int nM03\_exist = 0, nM04\_exist = 0, nM07\_exist = 0, nM08\_exist = 0,nM09\_exist = 0;
3. int nT\_exist = 0, nF\_exist = 0, nS\_exist = 0;
4. int nG00\_exist = 0, nG01\_exist = 0, nG02\_exist = 0, nG03\_exist = 0,nG90\_exist = 0;
5. *//后续的继承判断用*
6. int nCharNum = 0;  *// 一行字符串第nCharNum个字符*
7. int nCodeNum = 0;  *// 该行第nCodeNum个指令*
8. std::string str\_code = "";  *// 第nCodeNum个指令内容*
9. std::string codeArray[100];*//存储当前行解析出的所有代码，每个元素代表一个完整的代码指令，如"G01"或"X20"*

**<3.2>**获取当前行的字符串内容，并去除空格，代码如下：

1. auto lineContent = \_cmd\_cache[nLineNum];
2. lineContent.erase(
3. std::remove(lineContent.begin(), lineContent.end(), ' '),
4. lineContent.end());

**<3.3>**逐字符读这行字符串，总体来说分为两种情况：若是字母，说明是指令，则nCodeNum自增1，并将此指令字母存入到codeArray数组中；若是数字，小数点等等，那么拼接在codeArray数组中的字母后。

1. while (nCharNum <= lineContent.length() - 1) {
2. std::string str = lineContent.substr(nCharNum, 1);*//复制子字符串，要求从指定位置开始，并有指定长度1*
3. if (std::isdigit(str[0]) || str == "." || str == "-")*// 该字符为数字或小数追加在该指令后面*
4. {
5. str\_code = str\_code + str;
6. codeArray[nCodeNum] = str\_code;
7. } else {  *// 该字符为字母*
8. nCodeNum += 1;
9. str\_code = "";
10. str\_code = str\_code + str;
11. codeArray[nCodeNum] = str\_code;
12. }
13. nCharNum++;
14. }

比如:G01X100中的01就是对G命令的补充，此时0和1接连追加到codeArray[nCodeNum]后面。最终效果为codeArray[1]=G01，codeArray[2]=X100以此类推。

读完此行字符串后，将nCodeNum置为1继续解析。

**<3.4>**用while循环识别这行字符串中的所有指令。在上一步的最后，已将nCodeNum置为1，所以先提取此行第nCodeNum个指令（即第一个指令）的第1位字符，用变量firstchar表示，随后提取这个指令的内容，用变量txt表示。

1. *//复制子字符串，要求从指定位置开始，并有指定长度1*
2. std::string firstChar = codeArray[nCodeNum].substr(0, 1);
3. *//substr()中只有一个参数的话，会返回从那个位置之后的字符串，即返回字母后的数字*
4. std::string txt = codeArray[nCodeNum].substr(1);

**<3.5>**分类讨论firstchar的值，判断txt的内容，对应地将<3.1>中创建的变量置为1，并为mainwindow类中的一些私有变量赋值。若第1位字符是G代码：

1. if (firstChar == "G") {
2. switch (std::stoi(txt)) {
3. case 0:
4. nG00\_exist = 1;
5. nStatusArray[nLineNum] = 4;
6. break;
7. case 1:
8. nG01\_exist = 1;
9. nStatusArray[nLineNum] = 1;
10. break;
11. case 2:
12. nG02\_exist = 1;
13. nStatusArray[nLineNum] = 2;
14. break;
15. case 3:
16. nG03\_exist = 1;
17. nStatusArray[nLineNum] = 3;
18. break;
19. case 90:
20. nG90\_exist = 1;
21. G90[nLineNum] = 1;
22. break;
23. case 91:
24. nG90\_exist = 1;
25. G90[nLineNum] = 0;
26. break;
27. default:
28. break;
29. }
30. }

若第1位字符是X代码

1. if (firstChar == "X") {
2. nX\_exist = 1;
3. switch (G90[nLineNum]) {
4. case 1:
5. *// 绝对坐标模式下，直接将读取的值赋给终点X坐标*
6. fXendArray[nLineNum] = std::stof(txt);
7. break;
8. case 0:
9. *// 相对坐标模式下，将读取的值与起点X坐标相加*
10. fXendArray[nLineNum] = std::stof(txt) + fXbeginArray[nLineNum];
11. break;
12. default:
13. break;
14. }
15. }

其余的指令代码与上面例子同理，若第1位字符是R代码，则会设计到圆心的计算，原理如下：

已知（）与（）与R，求圆心（）

可知这两点满足与

分别展开，之后用上式减下式，可得

故有

令

故，将其代入进并展开可得

令A=，B=，C=;

那么上式就为A+B+C=0,此为一元二次方程。根据x起点坐标x终点坐标大小比较，顺时针圆弧，逆时针圆弧，R值符号来确定解的正负符号。如果输入的是I\_J\_这类格式，也可以计算出R

1. dRArray[nLineNum] =
2. sqrt(fIArray[nLineNum] \* fIArray[nLineNum]
3. + fJArray[nLineNum] \* fJArray[nLineNum]);

解析完此行的第一个指令后，nCodeNum自增1(此时变成2，说明下一步解析第二个指令)，进入到新一轮循环中。直到<3.4>中的循环条件不满足为止。

**（4）**继承判断，代码如下：

1. if (nX\_exist == 0) {
2. fXendArray[nLineNum] = fXbeginArray[nLineNum];
3. }
4. if (nY\_exist == 0) {
5. fYendArray[nLineNum] = fYbeginArray[nLineNum];
6. }
7. if ((nM03\_exist == 0 && nM04\_exist == 0) && (nLineNum != 0)) {
8. nDirArray[nLineNum] = nDirArray[nLineNum - 1];
9. }
10. if (nS\_exist == 0 && (nLineNum != 0)) {
11. nSArray[nLineNum] = nSArray[nLineNum - 1];
12. }
13. if (nM07\_exist == 0 && nM09\_exist == 0 && (nLineNum != 0)) {
14. nCool2Array[nLineNum] = nCool2Array[nLineNum - 1];
15. }
16. if (nM08\_exist == 0 && nM09\_exist == 0 && (nLineNum != 0)) {
17. nCool1Array[nLineNum] = nCool1Array[nLineNum - 1];
18. }
19. if (nT\_exist == 0 && (nLineNum != 0)) {
20. nTArray[nLineNum] = nTArray[nLineNum - 1];
21. }
22. if (nF\_exist == 0 && (nLineNum != 0)) {
23. fFArray[nLineNum] = fFArray[nLineNum - 1];
24. }
25. // 如果一行坐标前未出现插补方式，则默认为上一行插补方式
26. if ((nX\_exist != 0 || nY\_exist != 0)
27. && (nG00\_exist == 0 && nG01\_exist == 0 && nG02\_exist == 0
28. && nG03\_exist == 0))
29. {
30. nStatusArray[nLineNum] = nStatusArray[nLineNum - 1];
31. }
32. fXbeginArray[nLineNum + 1] = fXendArray[nLineNum];
33. fYbeginArray[nLineNum + 1] = fYendArray[nLineNum];

以X指令继承为例，第一行为：X100，第二行为Y50，在第二行中并未出现X指令，故nX\_exist == 0，那么第二行中X终点坐标即为第一行中X的终点坐标（在此例子中为100）。继承判断完毕后，nLineNum自增1，解析下一行的NC程序，循环直到解析完所有行的NC程序为止。

**（5）**解析完毕，将nLineNum置为0便于后续画图，并将完成标志finish置1。

## 2.3 定时器函数timer\_Tick()

此函数涉及除2.2中的私有变量外，还涉及mainwindow类中的剩下的一些私有变量：

1. int nQuadrant = 0;  *// 画圆象限*
2. int first = 1;  *// 画圆参数 根据圆的起点判断起点的象限*
3. int x0 {}, y0 {}, x1 {}, y1 {};  *// 画图参数 (x0,y0)--(x1,y1)*

此函数的伪代码如下：

1. /**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\****\*\*\**\*
2. 函数名称：void timer\_Tick()
3. 函数功能：定时器触发处理NC代码执行
4. 输入参数：无
5. 输出参数：无
6. 返 回 值：无
7. 备    注：此函数用于逐步执行NC代码，包括直线和圆弧插补，
8. 并根据不同的代码更新状态栏且路径显示。
9. **\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***/
10. void timer\_Tick()
11. {
12. // 检查仿真是否已完成所有命令
13. 如果 (finish == 1 且 nLineNum > \_cmd\_cache.size())
14. 停止计时器
15. 显示仿真完成消息
16. 重置 finish 和 nLineNum
17. 返回
18. // 处理当前行的命令
19. 如果 (finish == 1 且 nLineNum <= \_cmd\_cache.size())
20. 根据 nLineNum 读取状态信息
21. 更新机床状态和UI显示
22. // 检查是否需要直线插补或圆弧插补
23. 根据当前 G 代码 (存储在 nStatusArray 中)
24. 如果是 G00 或 G01
25. 执行直线插补计算
26. 更新位置信息
27. 如果是 G02 或 G03
28. 执行圆弧插补计算
29. 更新位置信息
30. // 检查是否达到每条命令的终点
31. 如果到达终点
32. 处理下一行
33. 否则
34. 下次定时器函数触发时继续当前命令的插补步骤
35. }

timer\_Tick()流程图如图2.3所示：

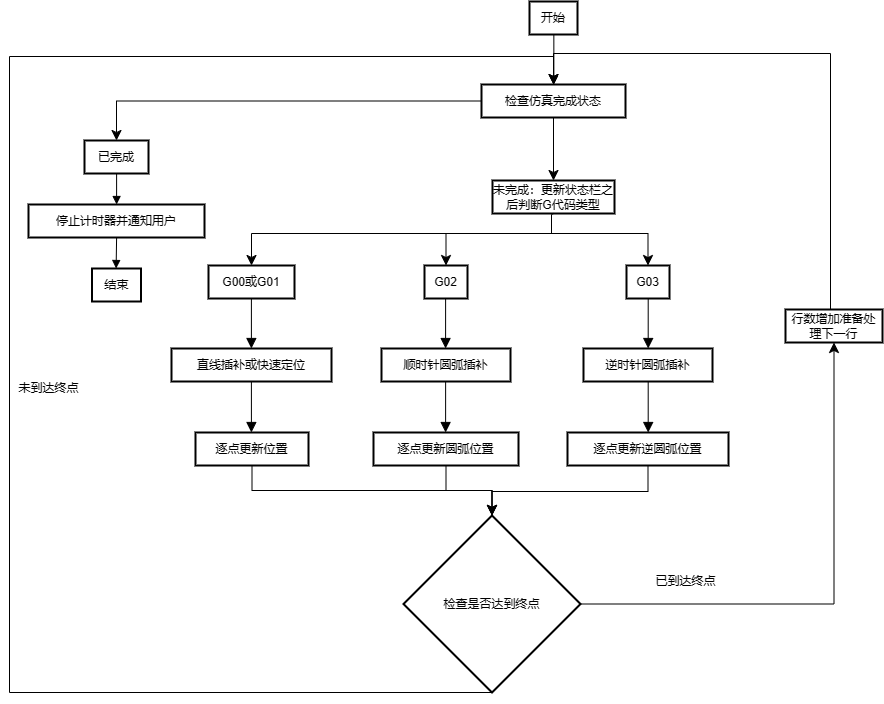


图2.3 timer\_Tick()函数的流程图

**（1）**首先检查nLineNum 是否大于\_cmd\_cache.size()，前者在此函数中代表着当前画图画到了第几行NC程序，后者可以理解为有多少行需要画。若大于，那么仿真状态完成，停止定时器并通知用户。

**（2）**若小于，先更新状态栏。具体来说，引入了status\_table类，此类中定义了一个结构体如下

1. struct Status
2. {
3. int knife\_count {};*//刀具*
4. int direction {};*//旋转方向*
5. int speed {};*//主轴速度*
6. int lubricant\_one {};*//冷却液1*
7. int lubricant\_two {};//冷却液2
8. QString supply\_speed {"0"};//进给速度
9. };

为Status结构体实例化对象，从缓存获取命令为变量赋值如

1. status.knife\_count = nTArray[nLineNum]*;*

这行代码即为获取当前NC程序行的刀具号，其余变量同理。之后调用status\_table类中的update\_status（），传入对象名，即可更新状态栏值。

**（3）**判断G代码类型，使用逐点比较法去更新画图点并画图。

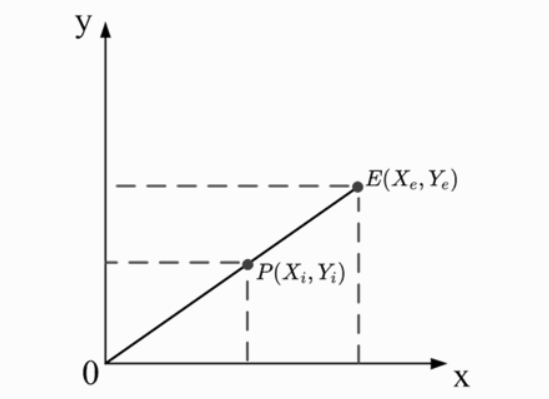
**<3.1>**若是G00或者G01，则为直线插补。以第一象限的直线插补为例。

图2.4 直线插补演示图

假设现在有一条如图2.4所示的第一象限直线OE，取O点为坐标原点，直线终点为E点且坐标已知，图中有一动点P为加工点。故用两点式方程可以得OE的方程为，设Fi为加工动点P与直线OE间的偏差，将Fi和点P坐标代入上式可得：，若此式大于等于0，表示动点P在直线OE上方，此时向+X方向进给一步；若此式小于0，表示动点P在直线OE下方，此时应该+Y方向进给一步。流程图如图2.6所示，其中的结束对应着图2.3中的已到达终点，之后行数增加准备处理下一行。

例子：（0,0）到（3,3），则插补直线路径为：

(0,0)-(1,0)-(1,1)-(2,1)-(2,2)-(3,2)-(3,3)。

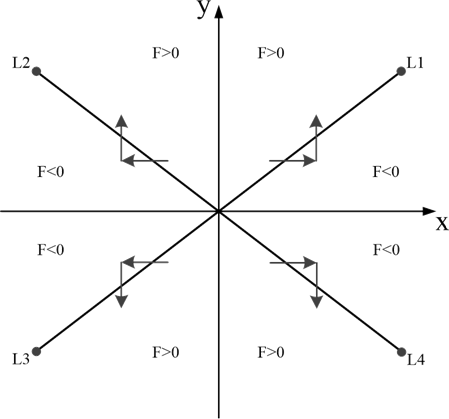
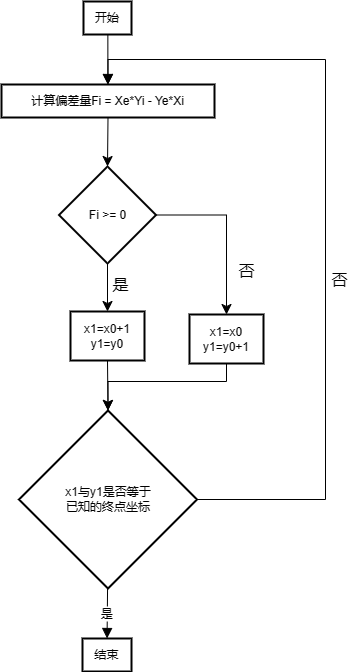
所有象限的F值的正负及对应的插补路线如图2.5所示

图2.5 所有象限的F值的正负及对应的插补路线

图2.6 直线插补流程图

代码如下：

1. if ((nStatusArray[nLineNum] == 4 || nStatusArray[nLineNum] == 1)
2. && fXbeginArray[nLineNum] < fXendArray[nLineNum]
3. && fYbeginArray[nLineNum] <= fYendArray[nLineNum])
4. {
5. // 根据偏差方程进行判断
6. if (((fXendArray[nLineNum] - fXbeginArray[nLineNum]) \* (y0 - fYbeginArray[nLineNum])
7. - (fYendArray[nLineNum] - fYbeginArray[nLineNum]) \* (x0 - fXbeginArray[nLineNum]))
8. >= 0)
9. {
10. x1 = x0 + 1;
11. y1 = y0;
12. }
13. else
14. {
15. y1 = y0 + 1;
16. x1 = x0;
17. }
18. // 实时坐标显示
19. update\_pos(x1, y1);
20. // 此段插补的终点作为下一小段插补的起点
21. x0 = x1;
22. y0 = y1;
23. // 终点判别
24. if (x1 == fXendArray[nLineNum] && y1 == fYendArray[nLineNum])
25. {
26. nLineNum ++;
27. }
28. }

其中update\_pos(int x, int y)的定义如下

1. /**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\****\*\*\**\*
2. 函数名称：void update\_pos()
3. 函数功能：更新刀具当前位置的显示
4. 输入参数：int x, int y 代表最终坐标
5. 输出参数：无
6. 返 回 值：无
7. 备    注：此函数更新用户界面上的刀具坐标位置显示，并在仿真场景中绘制从当前位置到新位置的直线。
8. **\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***/
9. void MainWindow::update\_pos(int x, int y)
10. {
11. \_label\_pos->setText("( " + QString::number(x) + "," + QString::number(y)+ " )");
12. \_milling\_item->lineTo(x, y);
13. }

lineTo函数位于milling\_item类中，由于milling\_item类继承于QGraphicsPathItem类，故可以通过QGraphicsPathItem类提供的绘制路径函数去绘制走刀路径。

**<3.2>**若是G02或者G03，则为圆弧插补。以逆时针插补G03为例。

逐点比较法的圆弧插补和直线插补一样，都分为偏差判别、坐标进给、偏差计算和终点判别四个步骤。以第一象限为示例，如图2.7，动点为（），半径为R，则偏差方程为，当F> 0，动点P在圆弧AB的外侧，应该向圆内即-X方向进给一步；F< 0，动点P在圆弧AB的内侧，应该向圆外即+Y方向进给一步。流程图如2.8所示。

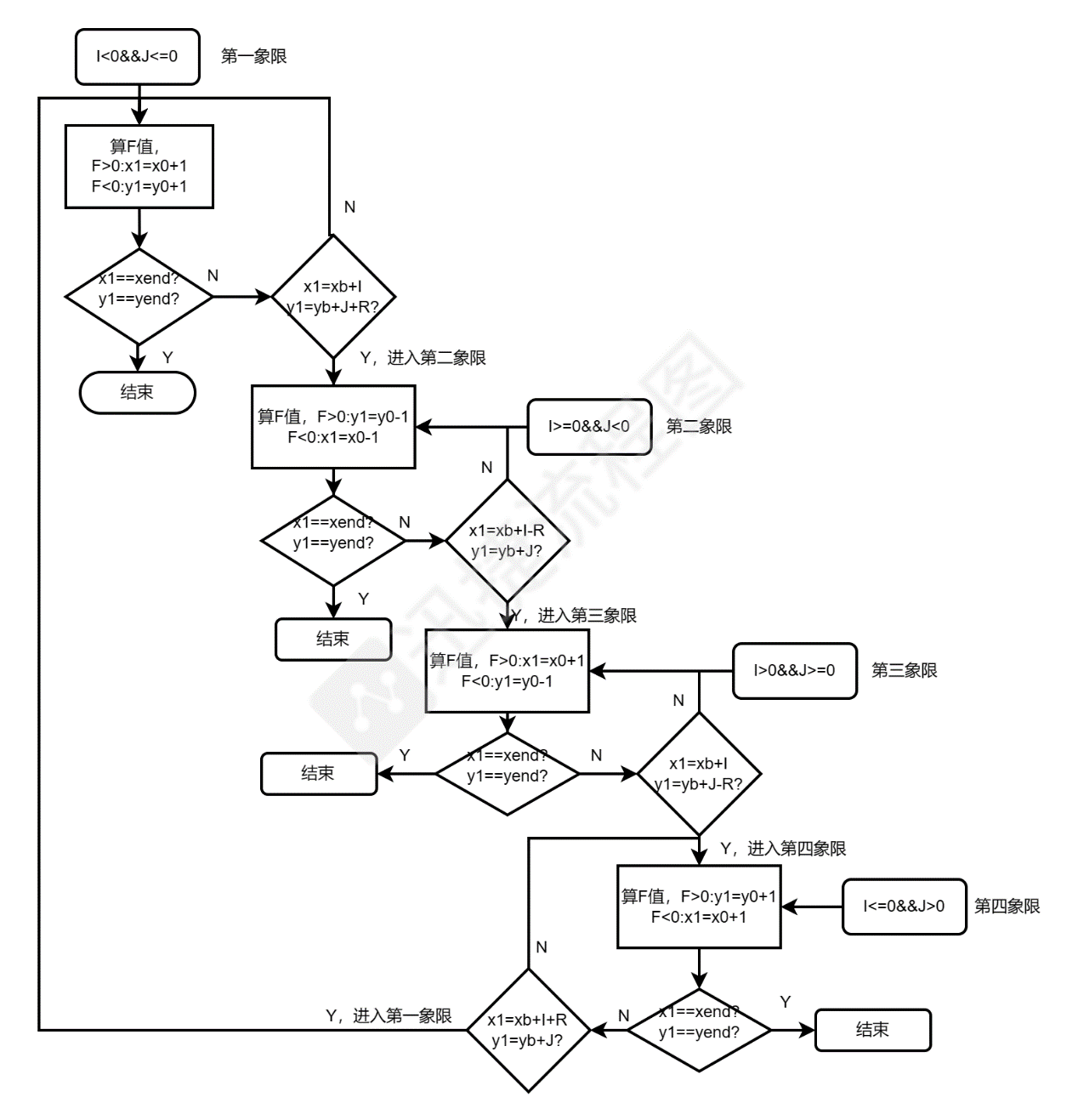
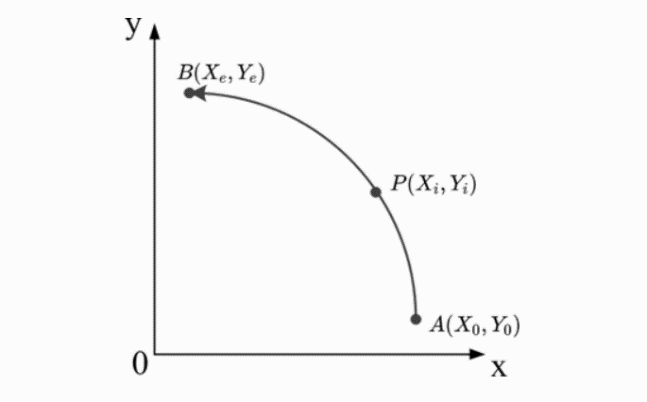
图2.7 逆时针圆弧插补演示图

图2.8 逆时针圆弧插补流程图

简单来说，首先判断是哪个象限，若为第一象限，代码如下：

1. if (nQuadrant == 1)  // 画第1象限
2. {
3. if (((x1 - fXbeginArray[nLineNum] - fIArray[nLineNum])
4. \* (x1 - fXbeginArray[nLineNum] - fIArray[nLineNum])
5. + (y1 - fYbeginArray[nLineNum] - fJArray[nLineNum])
6. \* (y1 - fYbeginArray[nLineNum] - fJArray[nLineNum])
7. - dRArray[nLineNum] \* dRArray[nLineNum])
8. > 0)
9. {
10. y1 = y0;
11. x1 = x0 - 1;
12. } else {
13. x1 = x0;
14. y1 = y0 + 1;
15. }
16. update\_pos(x1, y1);
17. y0 = y1;
18. x0 = x1;
19. if ((x1 == fXendArray[nLineNum])
20. && (y1 == fYendArray[nLineNum]))
21. {
22. nLineNum = nLineNum + 1;
23. first = 1;
24. }
25. if (x1 == (fXbeginArray[nLineNum] + fIArray[nLineNum])
26. && y1
27. == (fYbeginArray[nLineNum] + fJArray[nLineNum]
28. + dRArray[nLineNum]))
29. {
30. nQuadrant = 2;
31. }
32. }

其余象限也同理，与G00和G01不同，每次调用完update\_pos函数后，G02与G03不仅做终点判别，还得判断是否跨越象限。

图示

描述已自动生成顺时针G02及逆时针G03圆弧插补的F值及其进给方向如图2.9 图2.10所示

图示

描述已自动生成图2.9 G02圆弧插补的F值及其进给方向

图2.10 G03圆弧插补的F值及其进给方向

# 第三章 其余类的介绍

## 3.1 axis\_item类

此类继承自 Qt 的 QGraphicsItem，用于在图形场景中表示和绘制坐标轴。其实现方式涵盖以下几个主要部分：

1. 构造函数和析构函数：用于创建和销毁 AxisItem 对象。
2. boundingRect() 方法：重写基类的方法，提供了坐标轴的边界矩形，这是 Qt 绘制和事件处理系统所需的。
3. paint() 方法：也是重写基类的方法，它实际上负责绘制坐标轴本身，包括轴线和刻度。
4. setRange() 方法：允许外部设置坐标轴的范围，当范围改变时，触发图形项的更新和重绘。

其中最重要的paint()函数定义如下：

1. */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**
2. 函数名称：void paint()
3. 函数功能：在图形视图中绘制坐标轴及其刻度和标签
4. 输入参数：QPainter\* painter, QStyleOptionGraphicsItem\* option, QWidget\* widget
5. 输出参数：无
6. 返 回 值：无
7. 备    注：此函数负责绘制坐标轴的线条和刻度，并在刻度旁边添加数值标签，以便清楚地显示坐标轴的刻度。
8. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
9. void AxisItem::paint(QPainter\* painter,
10. const QStyleOptionGraphicsItem\* option,
11. QWidget\* widget)
12. {
13. Q\_UNUSED(option);*//用于避免编译器警告*
14. painter->setPen(QPen(Qt::black, 1));*//轴的颜色，像素*
15. const auto start\_x = -\_range.width /2;
16. const auto end\_x = \_range.width / 2;
17. const auto start\_y = -\_range.height / 2;
18. const auto end\_y = \_range.height / 2;
19. painter->drawLine(start\_x, 0, end\_x, 0);
20. painter->drawLine(0, start\_y, 0, end\_y);
21. *// 绘制x轴刻度*
22. const auto tick\_length = 10;
23. const auto step = 50.0;*//刻度的长度和间距*
24. for (int i = start\_x / step; i < end\_x / step; i++) {
25. const auto x = i \* step;
26. painter->drawLine(x, -tick\_length, x, tick\_length);
27. const auto text\_rect =
28. widget->fontMetrics().boundingRect(QString::number(x));*//将数字x转换为字符串，并传递给boundingRect()来计算这个字符串的边界矩形的宽高*
29. *// 轴上数字*
30. painter->drawText(
31. x - text\_rect.width() / 2,
32. - tick\_length - 10, QString::number(x));
33. *//第一个参数确定数字的水平位置，x为刻度线的位置，text\_rect.width() / 2 是文本宽度的一半，这样就可以将文本的中心对准刻度线*
34. *//第三个参数将x转成string类型*
35. }
36. *//y轴同理，在此处省略*
37. }

## 3.2 milling\_item类

此类是一个基于 QGraphicsPathItem 的自定义图形项类，用于在 Qt 的图形视图框架中利用QPainter绘制铣削路径。

1. class MillingItem : public QGraphicsPathItem
2. {
3. public:
4. MillingItem(QGraphicsItem\* parent = nullptr);
5. ~MillingItem();
6. void reset();*//清除路径，以便于重新开始铣削路径的绘制。*
7. void lineTo(int x, int y); *//接收新的坐标点，并将这个点添加到当前的路径中，形成一条新的线段*
8. protected:
9. *//绘制铣削路径*
10. void paint(QPainter\* painter,
11. const QStyleOptionGraphicsItem\* option,
12. QWidget\* widget) override;
13. private:
14. QPainterPath \_path;
15. };

其中的私有变量QPainterPath \_path; QPainterPath 用来描述和管理绘图路径的形状，QPainter用来实际的绘制操作，包括绘制 QPainterPath 描述的路径以及其他图形、文本等。

## 3.3 status\_table类

此类是基于 Qt 的 QAbstractTableModel 的一个自定义模型，用于在表格视图中显示和管理状态信息。继承 QAbstractTableModel 后，至少要实现三个纯虚函数才能进行实例化，在此类中实现的是rowCount()、columnCount()和data()。这个模型的主要作用和实现方式如下：

作用：

1管理和显示数控机床的状态，包括刀具数量、旋转方向、主轴速度、冷却液使用情况等。

2提供一个更新和刷新表格视图的界面，当数控机床状态发生变化时。

实现方式：

1重写了 rowCount() 和 columnCount() 方法来定义表格的行和列的数量。

2 data() 方法用于根据模型索引和角色返回数据，这是显示不同状态值的关键。

3 flags() 方法设置了表格项的属性，如是否可选中或可编辑。

4 update\_status() 方法用于更新表格模型的状态，这通常在状态发生变化时被调用，它会触发表格的重绘。

其中最关键的data函数定义如下：

1. */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**
2. 函数名称：QVariant data()
3. 函数功能：根据模型索引和角色返回状态表格中的数据
4. 输入参数：const QModelIndex& index, int role
5. 输出参数：无
6. 返 回 值：返回指定角色和索引对应的数据
7. 备    注：此函数是 Qt 表格模型的一部分，用于返回状态表格的内容。它处理文本对齐角色以确定列的对齐方式，并处理显示角色以返回状态表中每一行和列的具体内容。它用于控制如何在视图中显示每个单元格的数据，包括数值和对齐方式。
8. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
9. QVariant StatusTable::data(const QModelIndex& index, int role) const
10. {
11. switch (role) {
12. case Qt::TextAlignmentRole:
13. switch (index.column()) {
14. case KEY:
15. return Qt::AlignLeft;*//将“刀具”“冷却液”等等这几项居左对齐*
16. case VALUE:
17. return Qt::AlignCenter;*//将它们对应的值居中对齐*
18. }
19. case Qt::DisplayRole:
20. switch (index.column()) {
21. case KEY:
22. switch (Rows(index.row())) {
23. case Rows::KNIFE\_COUNT:
24. return "刀具";
25. case DIRECTION:
26. return "主轴旋转方向";
27. case SPEED:
28. return "主轴速度";
29. case Lubricant\_ONE:
30. return "切削液1";
31. case Lubricant\_TWO:
32. return "切削液2";
33. case SUPPLY\_SPEED:
34. return "进给速度";
35. case ROW\_COUNT:
36. break;
37. }
38. case VALUE:
39. switch (Rows(index.row())) {
40. case Rows::KNIFE\_COUNT:
41. return \_status.knife\_count ? \_ status.knife\_count: 1;//默认刀具号为1
42. case DIRECTION:
43. return \_status.direction;
44. case SPEED:
45. return \_status.speed;
46. case Lubricant\_ONE:
47. return \_status.lubricant\_one;
48. case Lubricant\_TWO:
49. return \_status.lubricant\_two;
50. case SUPPLY\_SPEED:
51. return \_status.supply\_speed;
52. case ROW\_COUNT:
53. break;
54. }
55. }
56. }
57. return {};
58. }

# 附录

流程图用mermaid代码实现，以下附上流程图的mermaid代码

1 图2.1

1. graph TD;
2. A[点击开始按钮] --> B[解析NC代码];
3. B --> C[启动定时器];
4. C --> D{定时器触发};
5. D --> E[执行timer\_Tick];
6. E --> F{命令处理完毕?};
7. F -- 是 --> G[停止定时器 并结束仿真];
8. F -- 否 --> H[从缓存获取命令];
9. H --> I[更新机器状态];
10. I --> J[更新界面坐标];
11. J --> K[调用lineTo绘制路径];
12. K --> L[场景更新];
13. L --> D;

2 图2.2

1. graph TD;
2. A[开始解析] --> B[清空命令缓存]
3. B --> C[读取文本并分行]
4. C --> D[检查并存储有效行]
5. D --> E[初始化行解析变量]
6. E --> F[逐字符解析指令和数据]
7. F --> G[处理G代码]
8. F --> H[处理坐标指令X和Y]
9. F --> I[处理代码F M S T]
10. G --> J[更新状态数组]
11. H --> J
12. I --> J
13. J --> K{所有行已处理?}
14. K -- 是 --> L[设置完成标志]
15. K -- 否 --> E
16. L --> M[结束解析]

3 图2.3

1. graph TD;
2. A[开始] --> B[检查仿真完成状态]
3. B --> C[已完成]
4. B --> D[未完成]
6. C --> E[停止计时器并通知用户]
7. D --> F[命令处理与状态更新]
9. F --> G[判断G代码类型]
10. G --> H[G00或G01]
11. G --> I[G02]
12. G --> J[G03]
13. H --> K[直线插补或快速定位]
14. I --> L[顺时针圆弧插补]
15. J --> M[逆时针圆弧插补]
16. K --> N[逐点更新位置]
17. L --> O[逐点更新圆弧位置]
18. M --> P[逐点更新逆圆弧位置]
20. N --> Q{检查是否达到终点}
21. O --> Q
22. P --> Q
24. Q -->|未到终点| B
25. Q -->|到终点| R[处理下一行]
26. R --> B
27. E --> S[结束]

4 图2.6

1. flowchart TD
2. A[开始] --> B[计算偏差量Fi = Xe\*Yi - Ye\*Xi]
3. B --> C{Fi >= 0}
4. C -- 是 -->D[x1=x0+1<br>y1=y0]
5. C -- 否 -->E[x1=x0<br>y1=y0+1]
6. D --> F{ x1与y1是否等于<br>已知的终点坐标}
7. E --> F
8. F -- 是 --> G[结束]
9. F -- 否 --> B

5 图2.8

1. flowchart TD
2. start[开始] --> check1{I<=0&&J<=0}
3. check1 -- "是" --> action1[更新坐标<br>F0:X1=X0+1<br>F0:Y1=Y0+1]
4. check1 -- "否" --> check2{I>=0&&J<0}
5. action1 --> checkEnd1{X1==Xend<br>Y1==Yend}
6. checkEnd1 -- "是" --> end[结束操作]
7. checkEnd1 -- "否" --> update1[更新坐标<br>X1=Xb+1<br>Y1=Yb+J?]
8. update1 --> action1
9. check2 -- "是" --> action2[更新坐标<br>F0:X1=X0-1<br>F0:Y1=Y0]
10. check2 -- "否" --> check3{I>0&&J>=0}
11. action2 --> checkEnd2{X1==Xend<br>Y1==Yend}
12. checkEnd2 -- "是" --> end
13. checkEnd2 -- "否" --> update2[更新坐标<br>X1=Xb+I-R<br>Y1=Yb+?]
14. update2 --> action2
15. check3 -- "是" --> action3[更新坐标<br>F0:X1=X0+1<br>F0:Y1=Y0-1]
16. check3 -- "否" --> check4{I<=0&&J>0}
17. action3 --> checkEnd3{X1==Xend<br>Y1==Yend}
18. checkEnd3 -- "是" --> end
19. checkEnd3 -- "否" --> update3[更新坐标<br>X1=Xb+I<br>Y1=Yb+J?]
20. update3 --> action3
21. check4 -- "是" --> action4[更新坐标<br>F0:X1=X0-1<br>F0:Y1=Y0+1]
22. action4 --> checkEnd4{X1==Xend<br>Y1==Yend}
23. checkEnd4 -- "是" --> end
24. checkEnd4 -- "否" --> update4[更新坐标<br>X1=Xb+I+R<br>Y1=Yb+?]
25. update4 --> action4