MECH 542: CAD/CAM Principles and Practice

Project #1 – CAD & CAM Postprocessor

Prepared by:

Colin Noort (82959396)

Rishabh Goel (81514283)

Adam Nguyen (47505102)

Submitted: December 23, 2021

Table of Contents

List of Figures

1 - Introduction

When preparing a 3D model for CNC machining, the starting point is to start with a simple design concept. The 3D model of this design concept is processed in a CAM (computer-aided-manufacturing) environment, and eventually, this piece is manufactured for use. One commonly used way to manufacture such a part is with a CNC machine, which cuts a stock piece of material into a physical model matching the virtual design. What happens in between involves what is called the post-processor. Post-processors are programs with algorithms to translate CL (current location) files into g-code files. CLSFs are generally the file types of output from CAD/CAM programs such as Siemens NX, while g-code files are used as directions for CNC machines to follow to produce the desired physical model. The CLSF contains all the information required to generate an accurate physical model of a part; however, it is not in the proper format for a CNC machine to read it.

A post-processor translates CLSFs into readable g-code configured for the specific CNC machine being used. This portion of the report focuses on the technical steps and setup required to develop and implement an accurate post-processor to translate a CLSF to a g-code file to be used on a Haas UMC-750 SS CNC machine. The UMC-750 is a 5 axis CNC machine capable of simultaneous machining. This machine can run at a spindle speed up to 15 thousand RPM and has a tool capacity of up to 50 [1]. This machine already has a developed, commercially available, and widely used post-processor. This post-processor is used with a CLSF output from Siemens NX to test our post-processor for accuracy. To verify the g-code generated through the post-processor, CG Tech's Vericut is used.

2 - Post-Processor Development

2.1 - Tools Used

The tools which were used in the development of the post-processor are as follows [2]:

1. Haas UMC-750 machine details and kinematics.
2. Vericut template of Haas UMC-750.
3. Library of general CL and G-code commands as well as their meanings.
4. Existing commercially developed Haas UMC-750 post-processor.
5. Siemens NX program programmed for project #2.

2.1.1 - Machine Details and Kinematics

Before programming the post-processor, it is essential to understand the machine kinematics of the UMC-750. This information was found in the operator's manual for this machine [3]. The 5 axes and their definitions can be seen in Figure 1.

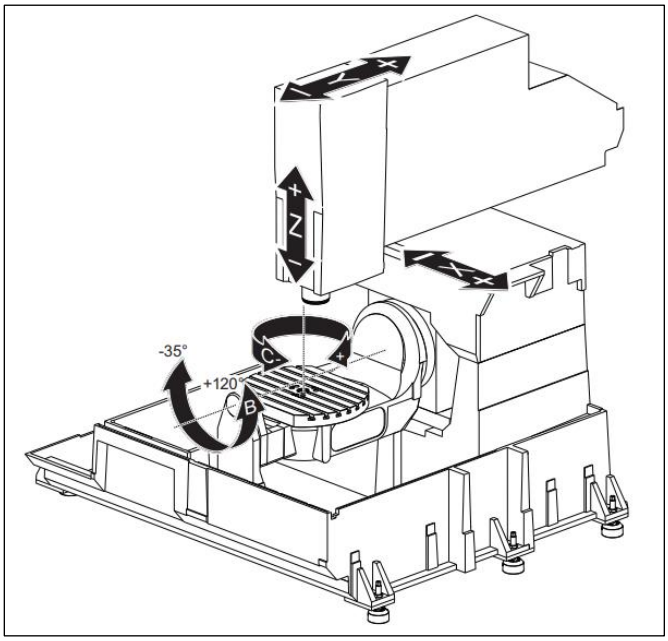

Figure 1 - Haas UMC 750-SS Axis Definitions [3]

As can be seen, there are three translational axes labeled X, Y, and Z and two rotational axes labeled B and C. The axes limits and further specifications for this machine can be found in appendix A of this report. The home position of the machine with respect to the machine coordinate system is [0, 0, 1]. This represents the i, j, and k coordinates of the machine. These coordinates change throughout the course of the operation of the machine according to the g-code instructions given to the machine.

2.1.2 - Haas UMC-750 Vericut Template

As mentioned in the introduction, to effectively verify the accuracy of the post-processor being developed, the output g-code must be tested using a virtual model of the Haas UMC 750, which was done with Vericut. The template of the Vericut model was provided and can be seen in Figure 2.

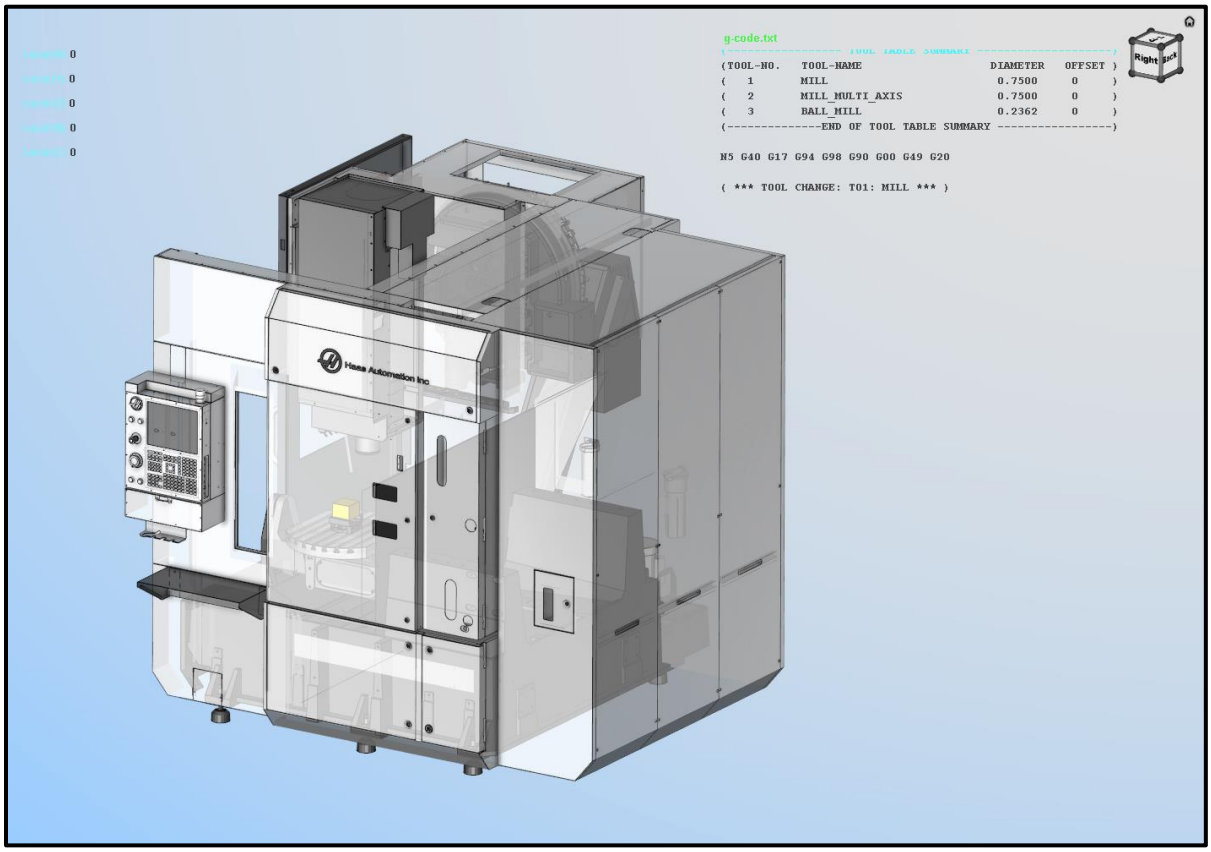

Figure 2 - Haas UMC-750 SS Vericut Model [4]

This template file includes a digital representation of the physical CNC machine model and the kinematic limits and machine parameters of the machine. A vice and a stock piece of material were added to this Vericut template model before testing our g-code. Additionally, to ensure accuracy, the tools to be used were imported into Vericut and listed in the upper right corner of Figure 2.

2.1.3 - Library of CL and G-Code Commands

One of the first steps in effectively converting a CLSF to a g-code file is understanding the relationships between the 2 languages. Once this is accomplished, the post-processor can translate any CLSF into a g-code file using a set of rules. To understand the CLSF (current location source file) file language, a Siemens NX webpage titled "The CLSF" was used in conjunction with the g-code file output using the developed Haas UMC-750 post-processor.

2.1.4 - Existing Haas UMC-750 Post-Processor

To compare the g-code developed with our post-processor, the g-code was output from Siemens NX using a commercially developed Haas UMC-750 post-processor, allowing for direct and effective comparison between our developed g-code and the "exact" g-code.

2.1.5 - Siemens NX Program Programmed for Project #2

The g-code used to compare post-processors was output using Siemens NX. Various machining operations, tool and machine parameters, and workpiece geometries were set up in Siemens NX to ensure accurate g-code was output. The project 2 report covers this in more detail. The g-code produced using Siemens NX was verified in Vericut and produced satisfactory results. Please see report 2 for additional details on the Siemens NX g-code produced and Vericut verification using this g-code.

3 - Post-Processor

Python was used to develop an effective post-processor. This post-processor was explicitly designed to convert CLSFs output from Siemens NX to readable g-code for the Haas UMC-750 machine. This program can be run from the command line of any computer or directly through any other IDE (integrated development environment) capable of running Python code.

3.1 - Main Function Overview

The post-processor works by first looking in the Python script's folder for the CLSF. The post-processor opens this file, and at the same time, creates a file called "g-code.txt." This file is where the g-code output from the post-processor is placed. Within the main program, a class called "CLSF_to_Gode" is created. This class contains all significant variables and functions used to translate the CL data to g-code data. The CLSF is first stored in its entirety; then, it is parsed into all relevant lines for the g-code conversion. Once this is complete, each line of the CLSF is read, translated, and output to the g-code.txt file using various sets of rules. This file can then be uploaded to any CAM verification software (e.g., Vericut) to verify the tool path output.

3.2 - Reading and Parsing CLSF Data

To effectively read and parse the data from the CLSF, the function "parse_CLSF" was created. First, this function opens the CLSF, strips the lines of CLSF data from the file, and then removes any unneeded information from the CLSF. This unneeded information was found to be any line of code starting with "paint/" or "$$." The lines containing "paint/" are lines from Siemens NX, which describe how a tool is displayed in NX [5]. The lines of code in the CLSF containing "$$" were found to be comments and thus can also be ignored. This portion of the "parse_CLSF" function can be seen in Figure 3.

```
# Parameters:
# CLSF_path (String) : Path of the CLSF File
def parse_CLSF(self, CLSF_path):

    # Debugging - comment this out after
    CLSF_path = 'cls.txt'

    # Open and turn CLSF file into list of strings (by line)
    with open(CLSF_path) as CLSF_File:
        unstripped_CLSF = CLSF_File.readlines()
        CLSF_to_GCode.CLSF = [line.strip() for line in unstripped_CLSF]

    CLSF_to_GCode.CLSF = [line for line in CLSF_to_GCode.CLSF if not 'PAINT' in line]
    CLSF_to_GCode.CLSF = [line.split('$')[0] for line in CLSF_to_GCode.CLSF]
```

Figure 3 - parse_CLSF Function

3.3 - Translating from CL to G-Code

Once this is done, the function moves onto the actual translation of the CLSF into g-code. First, every separate machining operation from the CLSF file is found. Every generated tool path contains a tool path header statement. These lines of code are distinguished by "TOOL PATH" at the beginning of the line. This statement delineates one tool path from another and is used by the system for various purposes, such as tool display, tool path transformations, optimizing the CLSF, etc. [5]. This "TOOL PATH" delineation was searched for in the CLSF file, and a library of operations was created. This library contains all information relating to the specific operation. The program stores all information provided from the CLSF file for each tool path, including the operation name, tool, and tool name.

Following this "TOOL PATH" delineation in the CLSF is the "TLDATA" delineation. This line contains all the specific cutting tool information related to the tool path and varies depending on the operation type. Our post-processor pulls out each tool's diameter, lower radius, taper angle, and tip angle related to each tool path. A tool number and operation number are associated with each of these 2 lines and saved for later use in the program. The loop used to accomplish this in our program can be seen in Figure 4.

```
# Scan and index all tools and operations ----------------------------------------------
operation_count = 0
line_count = 0
tool_count = 1
tool_name_to_number = {}

for line in CLSF_to_GCode.CLSF:
    # If we are changing tool...
    if 'TOOL PATH' in line:
        operation_count += 1
        # Make the tool object
        tool = Tool(line_count)
        tool.line_start = line_count

        # Add to tool dictionary
        if tool.tool_name not in tool_name_to_number:
            tool_name_to_number[tool.tool_name] = tool_count
            tool.tool_number = tool_count
            self.tools[tool_count] = tool
            tool_count += 1

        # Adding Tool Number to tool
        if tool.tool_name in tool_name_to_number:
            tool.tool_number = tool_name_to_number[tool.tool_name]

        # Index the new operation with that tool number
        self.operations[operation_count] = tool

    self.total_operations = operation_count
    line_count += 1
    # self.CLSF_line_count = line_count
```

Figure 4 - Finding each Tool and Operation

Next, the program searches through the remaining lines of the CLSF and distinguishes the type of command on each line by specific delimiters. These delimiters are "TOOL PATH," "LOAD/TOOL," "GOTO," and "CIRCLE." Each one of these delimiters describes a different type of operation. The "TOOLPATH" delimiter was already described. The "LOAD/TOOL" delimiter describes a tool change, the "GOTO" delimiter describes a linear interpolation movement, and the "CIRCLE" command describes a circular interpolation. This portion of code in the "parse_CLSF" function also writes the g-code to the "g-code.txt" file and does so differently depending on the type of command on each line. To do this effectively, a different function was written for each delimiter. This can be seen in Figure 5.

```
    # ------------------------------------------------------------------------------

    skip = 0

    for line in self.CLSF:

        for key in self.dictionary:
            if skip:
                skip -= 1
                break

            if key in line:
                skip = self.dictionary[key](self)

        self.CLSF_line_count = self.CLSF_line_count + 1

# Commands Dictionary
dictionary = {}
dictionary['TOOL PATH'] = new_operation
dictionary['LOAD/TOOL'] = load_tool
dictionary['GOTO'] = go_to
dictionary['CIRCLE'] = circular
```

Figure 5 - Delimiter Functions in Program

The "new_operation" function here translates code lines relating to any operation. This is any line starting with "TOOL PATH" as described above. This function writes all operation data to the g-code file depending on the operation. It also creates the tool table summary at the top of the g-code produced. This table can be seen in Figure 6.

```
(----------------- TOOL TABLE SUMMARY --------------------)
(TOOL-NO.   TOOL-NAME                   DIAMETER   OFFSET )
(   1       MILL                         0.7500     0     )
(   2       MILL_MULTI_AXIS              0.7500     0     )
(   3       BALL_MILL                    0.2362     0     )
(--------------END OF TOOL TABLE SUMMARY -----------------)
```

Figure 6 - Tool Table Summary

As mentioned, this "new_operation" function also prints out the corresponding g-code for each new operation to file. These lines of code are related to the coordinate system used for each operation and the initial position for the tool for each operation. The "new_operation" and "tool_table" functions can be seen in Figure 7.

```
def tool_table(self):
    self.g_code.append("(----------------- TOOL TABLE SUMMARY --------------------)")
    self.g_code.append(f"({'TOOL-NO.':<11}{'TOOL-NAME':<28}{'DIAMETER':<11}{'OFFSET':<7})")
    for key in self.tools:
        self.g_code.append(f"(
        {key:<8}{self.tools[key].tool_name:<29}{self.tools[key].diameter:<11.4f}{self.tools[key].offset:<6})")
    self.g_code.append("(--------------END OF TOOL TABLE SUMMARY -----------------)")


def new_operation(self):

    self.current_operation += 1

    tool = self.operations[self.current_operation]

    self.first_operation_move = True

    # Add these G-Code commands if this is the first opreation
    if self.current_operation == 1:
        self.tool_table()
        self.g_code.append("")
        self.g_code.append(f"N{self.n_index_return()} G40 G17 G94 G98 G90 G00 G49 G20")

    # For every new operation:
    self.g_code.append("")
    self.g_code.append(f"( *** TOOL CHANGE: T{tool.tool_number:02d}: {tool.tool_name} *** )")
    self.g_code.append("")
    self.g_code.append(f"( OPER: {tool.tool_path} )")

    # Add these G-Code commands if this is the first operation
    if self.current_operation == 1:
        self.g_code.append(f"N{self.n_index_return()} G53 G00 Z0.0")
        self.g_code.append(f"N{self.n_index_return()} G91 G28 X0.0 Y0.0")
        self.g_code.append(f"N{self.n_index_return()} G91 G28 B0.0 C0.0")
        self.g_code.append(f"N{self.n_index_return()} G90")

    return tool.line_skip
```

Figure 7 - "new_operation" and "tool_table" Functions

The next delimiter for which a function was written is the "LOAD/TOOL" delimiter. This function was named "load_tool." This is a simple program that writes the g-code for any tool change. The g-code related to a tool change is T_ and M06. As such, the tool number associated with the operation is pulled from the pre-defined operation library and appended to the T_, followed by an M06. This effectively tells the machine to change the current tool to the specified tool. It is essential to note also that M01, S_, and M03 are required to tell the machine to stop running the spindle and start running the spindle in the forward direction at a certain spindle speed once again the tool is changed. The "load_tool" function can be seen in Figure 8.

```
def load_tool(self):
    # current_line = self.CLSF_line_count
    # line = CLSF_to_GCode.CLSF[current_line].split(',')

    current_tool_number = self.operations[self.current_operation].tool_number
    current_tool_speed = self.operations[self.current_operation].speed
    self.g_code.append(f"N{self.n_index_return()} T{current_tool_number} M06")

    if self.current_operation + 1 in self.operations:
        operation_total_count = len(self.operations)

        for n in range(self.current_operation + 1, operation_total_count + 1):
            next_tool_number = self.operations[n].tool_number

            if next_tool_number != current_tool_number:
                self.g_code.append(f"N{self.n_index_return()} T{next_tool_number}")
                break

    self.g_code.append(f"N{self.n_index_return()} M01")
    self.g_code.append(f"N{self.n_index_return()} G53 G00 Z0.0")
    self.g_code.append(f"N{self.n_index_return()} S{current_tool_speed} M03")
    self.g_code.append(f"N{self.n_index_return()} G17 G54 G90")
```

Figure 8 - "load_tool" Function

The next delimiter for which a function was written is the "GOTO" delimiter. The function written for this delimiter is the "go_to" function. These lines in the CLSF file will either refer to a three-axis GOTO statement or a five-axis GOTO statement. In the three-axis GOTO statement, the corresponding coordinates are simply x, y, and z. In the five-axis GOTO statement, x, y, and z coordinates are defined, along with a tool axis vector in the form of i, j, and k unit vector coordinates. In this way, the location and orientation of the machine tool can be fully defined at every line of code.

One major difficulty with the i, j, k vectors involved here is that to properly capture the rotation the machine needs to satisfy (in g-code) rotation matrices must be implemented. The process here of calculating the rotation vector (i, j, k) is as follows:

1. Take the target coordinates from the CLSF files GOTO statement.
2. Send these coordinates into a rotation function.
3. Multiply by a y and z rotation matrix.
4. Send the rotation coordinates back to main function to be printed to file.

This can be seen in the following function seen in Figure 9:

```
def rotate(self, target_coord):
    beta = 90
    gamma = 90
    r2d = 180/math.pi

    if target_coord[3] == 0 and target_coord[5] == 0:
        gamma = 90
        beta = 90

    elif target_coord[3] == 0:
        gamma = 90
        try:
            beta = abs(math.atan(target_coord[4]/target_coord[5])) * r2d
        except:
            print(CLSF_to_GCode.CLSF[self.CLSF_line_count])
            raise("Please don't let me come here")

        if target_coord[4] < 0:
            gamma = -90


    else:
        # we transform from [0,0,1] to target to find rotation
        # First rotate C (about the Z-Axis) - we align the x axis to xy direction of target vector
        gamma = math.atan(target_coord[4]/target_coord[3]) * r2d

        xy = math.sqrt(target_coord[3]**2 + target_coord[4]**2)
        beta_prime = abs(math.atan(target_coord[5]/xy) * r2d)
        beta = 90 - beta_prime

        if target_coord[3] < 0:
            beta = -beta

    if beta < -35 or beta > 120:
        gamma = gamma - 180
        beta = -beta


    return beta, gamma
```

Figure 9 - Rotation Function

It is important to note that the rotations must remain within the machine limits of -35 and 120 degrees in B. Additionally, the CLSF file only provides rotations between 180 and -180 degrees, and so conditional rules were written to ensure the proper angles were being output. The rotation function itself outputs a beta and gamma value to be written to the g-code file.

We can see from Figure 10 that the rotation function works as intended. Figure 10 shows a side-by-side comparison of the "exact" g-code produced by the Haas post processor, and the g-code produced by our post-processor. We can see that the rotation values are the same in both cases.

```
1386  ( OPER: POCKETING_SIDE_OVAL )
1387  N6785 G255
1388  N6790 G53 G00 Z0.0
1389  N6795 S1146 M03
1390  N6800 M11 ( C-AXIS UNLOCK )
1391  N6805 M13 ( B-AXIS UNLOCK )
1392  N6810 G00 B82.875 C-180.
```

```
1309  ( OPER: POCKETING_SIDE_OVAL )
1310  N6465 X-1.9222 Y-2.3215 Z0.2772 B82.8750 C-180.0000
1311  N6470 X-1.9222 Y-2.5254 Z0.2665 F9.8425
1312  N6475 X-1.9452 Y-2.5648 Z0.2641
1313  N6480 X-1.9846 Y-2.5876 Z0.2617
1314  N6485 X-2.0314 Y-2.5875 Z0.2593
1315  N6490 X-2.0706 Y-2.5646 Z0.2568
```

Figure 10 - Demonstration of Rotation Matrices Functionality

Line N6810 of the "exact" g-code shows B82.875 C-180, and Line N6465 shows B82.8750 and C-180.0000, so we know that our rotation matrices and rotation function are functioning as intended. The 2 rotation matrices functions involved in achieving these results can be seen in Figure 11 and Figure 12.

```
def rotate_z_transform(self, target_coord, gamma):

    target_coord_result = target_coord[:]

    # while gamma < 0:
    #     gamma = gamma + 360

    theta = math.radians(gamma)

    rotation_matrix_z = [[math.cos(theta),  -math.sin(theta),       0,      0],
                         [math.sin(theta),  math.cos(theta),        0,      0],
                         [0,                0,                      1,      0],
                         [0,                0,                      0,      1]]

    rotation_matrix_z = np.array(rotation_matrix_z)

    target_coord_temp = [   [target_coord[0]],
                            [target_coord[1]],
                            [target_coord[2]],
                            [1]]

    target_coord_temp = np.array(target_coord_temp)

    rotated_coord = np.matmul(rotation_matrix_z, target_coord_temp)
    rotated_coord = rotated_coord.T

    target_coord_result[0] = rotated_coord[0][0]
    target_coord_result[1] = rotated_coord[0][1]
    target_coord_result[2] = rotated_coord[0][2]

    return target_coord_result
```

Figure 11 - Z Rotation Matrix Function

```
def rotate_y_transform(self, target_coord, beta):

        target_coord_result = target_coord[:]

        # while beta < 0:
        #     beta = beta + 360

        theta = math.radians(beta)

        rotation_matrix_y = [[math.cos(theta),      0,      math.sin(theta),    0],
                             [0,                    1,      0,                  0],
                             [-math.sin(theta),     0,      math.cos(theta),    0],
                             [0,                    0,      0,                  1]]

        rotation_matrix_y = np.array(rotation_matrix_y)

        target_coord_temp = [   [target_coord[0]],
                                [target_coord[1]],
                                [target_coord[2]],
                                [1]]

        target_coord_temp = np.array(target_coord_temp)

        rotated_coord = np.matmul(rotation_matrix_y, target_coord_temp)
        rotated_coord = rotated_coord.T

        target_coord_result[0] = rotated_coord[0][0]
        target_coord_result[1] = rotated_coord[0][1]
        target_coord_result[2] = rotated_coord[0][2]

        return target_coord_result
```

Figure 12 - Y Rotation Matrix Function

These rotation matrices functions employ the standard rotation matrix values and have the same form as a 3D rotation matrix.

The CLSF also contains information on how quickly the tool should move to the location specified by the GOTO command. This is designated by either "RAPID" or by nothing. If "RAPID" is specified, this is related to rapid transit g-code (G00), and if nothing is specified, the machine is to move to the new location at the current feed rate (G01). It is also important to note that the GOTO command's feed rate for the translation is specified in the line before the GOTO command in the CLSF file. This line is delimited by the "FEDRAT" delimiter. If our "go_to" function finds this, the function will write this feed rate to the g-code file in the form of an F_ command. The "go_to" function can be seen in Figure 13.

```
def go_to(self):
    current_line = self.CLSF_line_count
    previous_line = current_line - 1
    # next_line = current_line + 1

    try:
        target_coord = CLSF_to_GCode.CLSF[current_line].split('/')
        target_coord = target_coord[1].split(',')
        target_coord = [float(i) for i in target_coord]
    except:
        print(CLSF_to_GCode.CLSF[current_line])

    rapid = False
    feed = None
    # circle = False


    if 'RAPID' in CLSF_to_GCode.CLSF[previous_line]:
        rapid = True

    if 'FEDRAT' in CLSF_to_GCode.CLSF[previous_line]:
        line = CLSF_to_GCode.CLSF[previous_line].split(',')
        feed = float(line[1])

    # if not circle:
    self.linear(rapid, feed, target_coord)
```

Figure 13 - "go_to" Function

The last delimiter to address with a function is the "CIRCLE" delimiter. These lines specify either helical or circular motion. These commands are output in the CLSF in the following format [5]:

CIRCLE/x, y, z, i, j, k, r, t, f, d, e [TIMES,n]

x, y, and z are the center of the curve or helix in the MCS (machine coordinate system). i, j, and k, are circle or helix axis vector in the MCS. r is the radius of the circle or helix. t is the tolerance, f is the ratio, d is the tool diameter, and e is the corner radius of the tool. Our post-processor will read each line delimited by the "CIRCLE" delimiter and output the required g-code lines associated with it. This includes the centre of the circle, the direction (clockwise or counter clockwise) of rotation, and the i and j coordinates of the centre of the circle. Each circular interpolation command comes in a pair of 3 lines in CLSF files. The first line starts with a GOTO delimiter, then there is a "CIRCLE" delimiter, and then another "GOTO" delimiter. An example of this can be seen in Figure 14.

```
GOTO/2.7950,2.7950,1.7341
CIRCLE/2.7423,2.7199,1.7341,0.0000000,0.0000000,-1.0000000,0.0919,0.0063,0.5000,0.7500,0.0000
GOTO/2.7364,2.8115,1.7341
```

Figure 14 - CLSF Circular Interpolation

The first line describes the starting point of the curve being traced by the circular interpolation "CIRCLE" command. The third line describes the end point of this curve. The second line is a little more complex but follows the layout described above (x, y, z, i, j, k, r, t, f, d, e). All this information is translated to g-code by our post-processor.

One significant difference between the way our post-processor works vs how the Haas post-processor works is in how helixes are handled. If for instance, the 1st and 3rd line in the code shown

in Figure 14 from the CLSF file contains differences in the z coordinate (3rd number), the Haas post-processor interprets this as a helix, and writes g-code accordingly. The Haas post-processor splits the helical command into multiple linear commands as can be seen on the left-hand side of Figure 15 below. In our case however, the post-processor writes a circular interpolation as g-code and decreases the z value separately. This can be seen in the right-hand side of Figure 15 below.

```
43  N95 G01 X2.2701 Y2.3302 Z3.0671 F9.8
44  N100 X2.1471 Y2.3384 Z3.0342
45  N105 X2.0293 Y2.3022 Z3.0013
46  N110 X1.9321 Y2.2265 Z2.9684
47  N115 X1.8681 Y2.1212 Z2.9355
48  N120 X1.849 Y2. Z2.9026
```

```
24  N65 G01 Z2.9026 F9.8425
25  N70 G03 X1.8490 Y2.0000 I-0.1938 J-0.2763
26  N75 G01 Y1.8490
27  N80 X2.1510
28  N85 Y2.1510
29  N90 X1.8490
```

Figure 15 - Helical Comparison between Post-Processors

It is important to note here also, that the CLSF file itself does not contain explicit information regarding the direction of rotation (G03 vs G02). This rule is calculated by the post processor, and so in our case, had to be hard coded in a rule for when to circularly interpolate clockwise (G02) and when to circularly interpolate counter clockwise (G03). The process for calculating this is as follows:

1. Categorize the quadrant which the machine tool vector is approaching a circle (1,2,3,4).
2. Calculate the radius of the circle used for interpolation based on x and y starting, and x and y ending coordinates.
3. Calculate the angle between the entry and exit machine tool vector from the circle.
4. If the exit machine tool vector angle minus the starting machine tool vector angle is greater then or equal to 180, interpolate clockwise. If it is less then 180, interpolate counter clockwise.

A portion of these calculations can be seen captured in code in Figure 16.

```
x_end_angle = math.degrees(math.acos(ratio))

if end_quadrant == 1:
    x_end_angle = x_end_angle
if end_quadrant == 2:
    x_end_angle = x_end_angle
if end_quadrant == 3:
    x_end_angle = x_end_angle + 90
if end_quadrant == 4:
    x_end_angle == 360 - x_end_angle

# if x_end_angle < 0:
#     x_end_angle + 360

# if x_start_angle < 0:
#     x_start_angle + 360

if x_end_angle - x_start_angle >= 180:
    return True
else:
    return False
```

Figure 16 - Circular Interpolation Calculations

We ensured that our calculations were accurate by checking our g-code file against the “exact” g-code file once again and the results of this can be seen in Figure 17 below.

```
62  N190 G03 X.8154 Y.8153 I.0932 J-.0067
63  N195 G03 X.8729 Y.799 I.0517 J.0725
64  N200 G01 X3.1246
65  N205 G03 X3.1847 Y.8154 I.0067 J.0932
66  N210 G03 X3.201 Y.8729 I-.0725 J.0517
67  N215 G01 Y3.1246
68  N220 G03 X3.1845 Y3.1847 I-.0931 J.0067
69  N225 G03 X3.1272 Y3.201 I-.0516 J-.0723
```

```
39  N140 G03 X0.8154 Y0.8153 I0.0931 J-0.0067
40  N145 G03 X0.8729 Y0.7990 I0.0518 J0.0726
41  N150 G01 X3.1246 Y0.7990
42  N155 G03 X3.1847 Y0.8154 I0.0067 J0.0931
43  N160 G03 X3.2010 Y0.8729 I-0.0726 J0.0518
44  N165 G01 X3.2010 Y3.1246
45  N170 G03 X3.1845 Y3.1847 I-0.0931 J0.0067
46  N175 G03 X3.1272 Y3.2010 I-0.0516 J-0.0724
```

Figure 17 - Demonstration of Circular Interpolation Functionality

Closing Remarks

This was an incredibly rewarding project to work on. Although it was very time consuming to write the post-processor and troubleshoot issues along the way, the process of writing a post-processing program for a CNC machine based on CLSF data was an invaluable experience. The post-processor code written is more than 800 lines and took over 50 hours to program. We are very satisfied with our results, and we believe the post-processor is an exemplary example of an effective application of programming in post-processor and CNC applications.

Appendix A – Haas UMC 750 Specifications

Travels		
	S.A.E	**Metric**
X Axis	30"	762 mm
Y Axis	20"	508 mm
Z Axis	20"	508 mm
C-Axis Rotation	360° Rotation	
B-Axis Tilt	-35° to +120°	
Spindle Nose to Table (~ min.)	4"	102 mm
Spindle Nose to Table (~ max.)	24"	610 mm
For detailed machine dimensions, including work envelope information, refer to the UMC-750 Machine Layout Drawing on www.haascnc.com.		

Platter		
	S.A.E	**Metric**
Platter Diameter	19.7"	500 mm
T-Slot Width	5/8"	16 mm
T-Slot Center Distance	2.48"	63 mm
Number of Standard T-Slots	7	
Max. Weight on Table (evenly distributed)	660 lb	300 kg

T1.5: General Requirements

General Requirements		
	S.A.E	**Metric**
Air Required	4 scfm, 100 psi	113 L/min, 6.9 bar
Coolant Capacity	75 gal	284 L
Power Requirement, Low Voltage	195-260 VAC / 100A	
Power Requirement, High Voltage	354-488 VAC / 50A	
Machine Weight	18,000 lb	8165 kg

T1.6: Standard Features

Standard Features
Tool Center Point Control (TCPC), Dynamic Work Offsets (DWO), Remote Jog Handle*, Second Home*, Macros*, Spindle Orientation (SO)*, Coordinate Rotation and Scaling (COORD)*, TSC-Ready, Wireless Intuitive Probing System (WIPS) *Refer to the Mill Operator's Manual (96-8210) for information on these features.